ALGORITHMS
OF INFORMATICS

Volume 3

AnTonCom
Budapest, 2011

This electronic book was prepared in the framework of project Eastern Hungarian Informatics Books Repository no. TÁMOP-4.1.2-08/1/A-2009-0046.
This electronic book appeared with the support of European Union and with the co-financing of European Social Fund.

Editor: Antal Iványi

Authors of Volume 3: Mira-Cristiana Anisiu (Chapter 26) Béla Vizvári (Chapter 24), Antal Iványi (Chapter 25), Zoltán Kása (Chapter 25), Ferenc Szidarovszky (Chapter 10), László Szirmay-Kalos (Chapter 28),

Validators of Volume 3: György Kovács (Chapter 24) Zoltán Kása (Chapter 25), Antal Iványi (Chapter 26), Anna Iványi (Bibliography)

Contents

Introduction

The first volume of the book *Informatikai algoritmusok* appeared in 2004 in Hungarian [?], and the second volume of the book appeared in 2005 [?]. Two volumes contained 31 chapters: 23 chapters of the present book, and further chapters on clustering, frequent elements in data bases, geoinformatics, inner-point methods, number theory, Petri-nets, queuing theory, scheduling.

The Hungarian version of the first volume contains those chapters which were finished until May of 2004, and the second volume contains the chapters finished until April of 2005.

The printed English version contains the chapters submitted until April of 2007. Volume 1 [?] contains the chapters belonging to the fundamentals of informatics, while the second volume [?] contains the chapters having closer connection with some applications.

The given book is the extended and corrected electronic version of the printed book written is English.

The chapters of the first volume are divided into three parts. The chapters of Part 1 are connected with automata: *Automata and Formal Languages* (written by Zoltán Kása, Sapientia Hungarian University of Transylvania), *Compilers* (Zoltán Csörnyei, Eötvös Loránd University), *Compression and Decompression* (Ulrich Tamm, Chemnitz University of Technology Commitment), *Reliable Computations* (Péter Gács, Boston University).

The chapters of Part 2 have algebraic character: here are the chapters *Algebra* (written by Gábor Ivanyos, and Lajos Rónyai, Budapest University of Technology and Economics), *Computer Algebra* (Antal Járai, Attila Kovács, Eötvös Loránd University), further *Cryptology* and *Complexity Theory* (Jörg Rothe, Heinrich Heine University).

The chapters of the third part have numeric character: *Competitive Analysis* (Csanád Imreh, University of Szeged), *Game Theory* and *Risk Analysis* (Ferenc Szidarovszky, The University of Arizona) and *Scientific Computations* (Aurél Galántai, András Jeney, University of Miskolc).

The second volume is also divided into three parts. The chapters of Part 4 are connected with computer networks: *Distributed Algorithms* (Burkhard Englert, California State University; Dariusz Kowalski, University of Liverpool; Grzegorz Malewicz, University of Alabama; Alexander Allister Shvartsman, University of Connecticut), *Parallel Algorithms* (Claudia Fohry, University of Kassel and Antal Iványi, Eötvös Loránd University;), *Network Simulation* (Tibor Gyires, Illinois State University) and *Systolic Systems* (Eberhard Zehendner, Friedrich Schiller University).

The chapters of Part 5 are *Relational Databases* and *Query in Relational Databases* (János Demetrovics, Eötvös Loránd University; Attila Sali, Alfréd Rényi Institute of Mathematics), *Semistructured Data Bases* (Attila Kiss, Eötvös Loránd University) and *Memory Management* (Ádám Balog, Antal Iványi, Eötvös Loránd University).

The chapters of the third part of the second volume have close connections with biology: *Bioinformatics* (István Miklós, Rényi Institute of Mathematics), *Human-Computer Interactions* (Ingo Althöfer, Stefan Schwarz, Friedrich Schiller University), and *Computer Graphics* (László Szirmay-Kalos, Budapest University of Technology and Economics).

The chapters are validated by Gábor Ivanyos,qnevindexIvanyos, Gábor Lajos Rónyai, András Recski, and Tamás Szántai (Budapest University of Technology and Economics), Sándor Fridli, János Gonda, and Béla Vizvári (Eötvös Loránd University), Pál Dömösi, and Attila Pethő (University of Debrecen), Zoltán FülöpqnevindexFülöp, Zoltán (University of Szeged), Anna GálqnevindexGál, Anna (University of Texas), János Mayer (University of Zürich).

The book contains verbal description, pseudocode and analysis of over 200 algorithms, and over 350 figures and 120 examples illustrating how the algorithms work. Each section ends with exercises and each chapter ends with problems. In the book you can find over 330 exercises and 70 problems.

We have supplied an extensive bibliography, in the section *Chapter Notes* of each chapter. The web site of book contains the maintained living version of the bibliography in which the names of authors, journals and publishers are usually links to the corresponding web site.

The LaTeX style file was written by Viktor Belényesi, Zoltán Csörnyei and László Domoszlai. The figures was drawn or corrected by Kornél Locher. Anna Iványi transformed the bibliography into hypertext.

The publication of the printed book was supported by Department of Mathematics of Hungarian Academy of Science, and the electronic version received support from ?????????????????????????????????????

We plan to publish the corrected and extended version of this book in printed and electronic form too. This book has a web site: http://elek.inf.elte.hu/EnglishBooks. You can use this website to obtain a list of known errors, report errors, or make suggestions (using the data of the colofon page you can contact with any of the creators of the book). The website contains the maintaned PDF version of the bibliography in which the names of the authors, journals and publishers are usually active links to the corresponding web sites (the living elements are underlined in the printed bibliography). We welcome ideas for new exercises and problems.

Budapest, Szeptember 2010

Antal Iványi (tony@compalg.inf.elte.hu)

24. The Branch and Bound Method

It has serious practical consequences if it is known that a combinatorial problem is NP-complete. Then one can conclude according to the present state of science that no simple combinatorial algorithm can be applied and only an enumerative-type method can solve the problem in question. Enumerative methods are investigating many cases only in a non-explicit, i.e. implicit, way. It means that huge majority of the cases are dropped based on consequences obtained from the analysis of the particular numerical problem. The three most important enumerative methods are (i) implicit enumeration, (ii) dynamic programming, and (iii) branch and bound method. This chapter is devoted to the latter one. Implicit enumeration and dynamic programming can be applied within the family of optimization problems mainly if all variables have discrete nature. Branch and bound method can easily handle problems having both discrete and continuous variables. Further on the techniques of implicit enumeration can be incorporated easily in the branch and bound frame. Branch and bound method can be applied even in some cases of nonlinear programming.
The *Branch and Bound* (abbreviated further on as B&B) method is just a frame of a large family of methods. Its substeps can be carried out in different ways depending on the particular problem, the available software tools and the skill of the designer of the algorithm.

Boldface letters denote vectors and matrices; calligraphic letters are used for sets. Components of vectors are denoted by the same but non-boldface letter. Capital letters are used for matrices and the same but lower case letters denote their elements. The columns of a matrix are denoted by the same boldface but lower case letters.

Some formulae with their numbers are repeated several times in this chapter. The reason is that always a complete description of optimization problems is provided. Thus the fact that the number of a formula is repeated means that the formula is *identical* to the previous one.

24.1. An example: the Knapsack Problem

In this section the branch and bound method is shown on a numerical example. The problem is a sample of the binary knapsack problem which is one of the easiest problems of integer programming but it is still NP-complete. The calculations are carried out in a brute force way to illustrate all features of B&B. More intelligent calculations, i.e. using implicit enumeration techniques will be discussed only at the end of the section.

24.1.1. The Knapsack Problem

There are many different knapsack problems. The first and classical one is the binary knapsack problem. It has the following story. A tourist is planning a tour in the mountains. He has a lot of objects which may be useful during the tour. For example ice pick and can opener can be among the objects. We suppose that the following conditions are satisfied.

- Each object has a positive value and a positive weight. (E.g. a balloon filled with helium has a negative weight. See Exercises 24.1-1 and 24.1-2) The value is the degree of contribution of the object to the success of the tour.
- The objects are independent from each other. (E.g. can and can opener are not independent as any of them without the other one has limited value.)
- The knapsack of the tourist is strong and large enough to contain all possible objects.
- The strength of the tourist makes possible to bring only a limited total weight.
- But within this weight limit the tourist want to achieve the maximal total value.

The following notations are used to the mathematical formulation of the problem:

n	the number of objects;
j	the index of the objects;
w_j	the weight of object j;
v_j	the value of object j;
b	the maximal weight what the tourist can bring.

For each object j a so-called *binary* or *zero-one* decision variable, say x_j, is introduced:

$$x_j = \begin{cases} 1 & \text{if object } j \text{ is present on the tour} \\ 0 & \text{if object } j \text{ isn't present on the tour.} \end{cases}$$

Notice that

$$w_j x_j = \begin{cases} w_j & \text{if object } j \text{ is present on the tour,} \\ 0 & \text{if object } j \text{ isn't present on the tour} \end{cases}$$

is the weight of the object in the knapsack.

Similarly $v_j x_j$ is the value of the object on the tour. The total weight in the knapsack is

$$\sum_{j=1}^{n} w_j x_j$$

which may not exceed the weight limit. Hence the mathematical form of the problem is

$$\max \sum_{j=1}^{n} v_j x_j \tag{24.1}$$

$$\sum_{j=1}^{n} w_j x_j \le b \tag{24.2}$$

$$x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n\,. \tag{24.3}$$

The difficulty of the problem is caused by the integrality requirement. If constraint (24.3) is substituted by the relaxed constraint, i.e. by

$$0 \le x_j \le 1, \quad j = 1, \ldots, n\,, \tag{24.4}$$

then the Problem (24.1), (24.2), and (24.4) is a linear programming problem. (24.4) means that not only a complete object can be in the knapsack but any part of it. Moreover it is not necessary to apply the simplex method or any other LP algorithm to solve it as its optimal solution is described by

Theorem 24.1 *Suppose that the numbers v_j, w_j $(j = 1, \ldots, n)$ are all positive and moreover the index order satisfies the inequality*

$$\frac{v_1}{w_1} \ge \frac{v_2}{w_2} \cdots \ge \frac{v_n}{w_n}\,. \tag{24.5}$$

Then there is an index p $(1 \le p \le n)$ and an optimal solution $\mathbf{x}^*$ such that

$$x_1^* = x_2^* = \cdots = x_{p-1}^* = 1,\; x_{p+1}^* = x_{p+2}^* = \cdots = x_{p+1}^* = 0\,.$$

Notice that there is only at most one non-integer component in $\mathbf{x}^*$. This property will be used at the numerical calculations.

From the point of view of B&B the relation of the Problems (24.1), (24.2), and (24.3) and (24.1), (24.2), and (24.4) is very important. Any feasible solution of the first one is also feasible in the second one. But the opposite statement is not true. In other words the set of feasible solutions of the first problem is a proper subset of the feasible solutions of the second one. This fact has two important consequences:

- The optimal value of the Problem (24.1), (24.2), and (24.4) is an upper bound of the optimal value of the Problem (24.1), (24.2), and (24.3).
- If the optimal solution of the Problem (24.1), (24.2), and (24.4) is feasible in the Problem (24.1), (24.2), and (24.3) then it is the optimal solution of the latter problem as well.

These properties are used in the course of the branch and bound method intensively.

24.1.2. A numerical example

The basic technique of the B&B method is that it divides the set of feasible solutions into smaller sets and tries to fathom them. The division is called ***branching*** as new branches are created in the enumeration tree. A subset is fathomed if it can be determined exactly if it contains an optimal solution.

To show the logic of B&B the problem

$$\begin{array}{rrrrrrrrrrrl} \max & 23x_1 & + & 19x_2 & + & 28x_3 & + & 14x_4 & + & 44x_5 & & \\ & 8x_1 & + & 7x_2 & + & 11x_3 & + & 6x_4 & + & 19x_5 & \leq & 25 \\ & & & & & \multicolumn{5}{c}{x_1, x_2, x_3, x_4, x_5 = 0 \text{ or } 1} & & \end{array} \tag{24.6}$$

will be solved. The course of the solution is summarized on Figure 24.1.2.

Notice that condition (24.5) is satisfied as

$$\frac{23}{8} = 2.875 > \frac{19}{7} \approx 2.714 > \frac{28}{11} \approx 2.545 > \frac{14}{6} \approx 2.333 > \frac{44}{19} \approx 2.316\,.$$

The set of the feasible solutions of (24.6) is denoted by $\mathcal{F}$, i.e.

$$\mathcal{F} = \{\mathbf{x} \mid 8x_1 + 7x_2 + 11x_3 + 6x_4 + 19x_5 \leq 25;\ x_1, x_2, x_3, x_4, x_5 = 0 \text{ or } 1\}.$$

The continuous relaxation of (24.6) is

$$\begin{array}{rrrrrrrrrrrl} \max & 23x_1 & + & 19x_2 & + & 28x_3 & + & 14x_4 & + & 44x_5 & & \\ & 8x_1 & + & 7x_2 & + & 11x_3 & + & 6x_4 & + & 19x_5 & \leq & 25 \\ & & & & & \multicolumn{5}{c}{0 \leq x_1, x_2, x_3, x_4, x_5 \leq 1\,.} & & \end{array} \tag{24.7}$$

The set of the feasible solutions of (24.7) is denoted by $\mathcal{R}$, i.e.

$$\mathcal{R} = \{\mathbf{x} \mid 8x_1 + 7x_2 + 11x_3 + 6x_4 + 19x_5 \leq 25;\ 0 \leq x_1, x_2, x_3, x_4, x_5 \leq 1\}.$$

Thus the difference between (24.6) and (24.7) is that the value of the variables must be either 0 or 1 in (24.6) and on the other hand they can take any value from the closed interval $[0, 1]$ in the case of (24.7).

Because Problem (24.6) is difficult, (24.7) is solved instead. The optimal solution according to Theorem 24.1 is

$$x_1^* = x_2^* = 1, x_3^* = \frac{10}{11}, x_4^* = x_5^* = 0\,.$$

As the value of x_3^* is non-integer, the optimal value 67.54 is just an upper bound of the optimal value of (24.6) and further analysis is needed. The value 67.54 can be rounded down to 67 because of the integrality of the coefficients in the objective function.

The key idea is that the sets of feasible solutions of both problems are divided into two parts according the two possible values of x_3. The variable x_3 is chosen as its value is non-integer. The importance of the choice is discussed below.

Let

$$\mathcal{F}_0 = \mathcal{F},\ \mathcal{F}_1 = \mathcal{F}_0 \cap \{\mathbf{x} \mid x_3 = 0\},\ \mathcal{F}_2 = \mathcal{F}_0 \cap \{\mathbf{x} \mid x_3 = 1\}$$

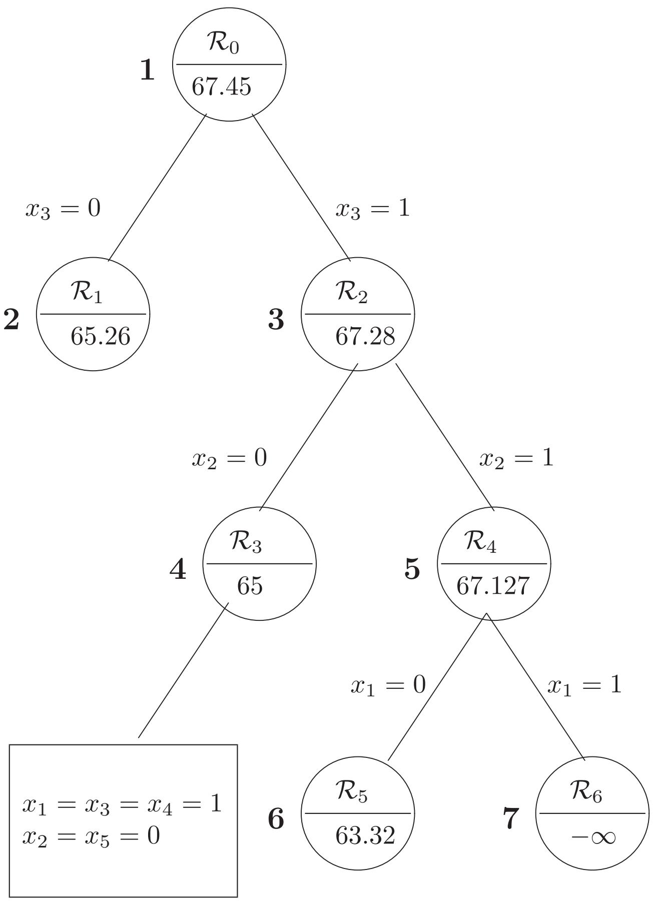


Figure 24.1 The first seven steps of the solution

and

$$\mathcal{R}_0 = \mathcal{R},\ \mathcal{R}_1 = \mathcal{R}_0 \cap \{\mathbf{x} \mid x_3 = 0\},\ \mathcal{R}_2 = \mathcal{R}_0 \cap \{\mathbf{x} \mid x_3 = 1\}\,.$$

Obviously

$$\mathcal{F}_1 \subseteq \mathcal{R}_1 \text{ and } \mathcal{F}_2 \subseteq \mathcal{R}_2\,.$$

Hence the problem

$$\begin{aligned} \max 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ \mathbf{x} \in \mathcal{R}_1 \end{aligned} \tag{24.8}$$

is a relaxation of the problem

$$\begin{aligned} \max 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ \mathbf{x} \in \mathcal{F}_1 \,. \end{aligned} \tag{24.9}$$

Problem (24.8) can be solved by Theorem 24.1, too, but it must be taken into consideration that the value of x_3 is 0. Thus its optimal solution is

$$x_1^* = x_2^* = 1,\ x_3^* = 0,\ x_4^* = 1,\ x_5^* = \frac{4}{19}.$$

The optimal value is 65.26 which gives the upper bound 65 for the optimal value of Problem (24.9). The other subsets of the feasible solutions are immediately investigated. The optimal solution of the problem

$$\begin{aligned} \max 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ \mathbf{x} \in \mathcal{R}_2 \end{aligned} \tag{24.10}$$

is

$$x_1^* = 1,\ x_2^* = \frac{6}{7},\ x_3^* = 1,\ x_4^* = x_5^* = 0$$

giving the value 67.28. Hence 67 is an upper bound of the problem

$$\begin{aligned} \max 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ \mathbf{x} \in \mathcal{F}_2 \ . \end{aligned} \tag{24.11}$$

As the upper bound of (24.11) is higher than the upper bound of (24.9), i.e. this branch is more promising, first it is fathomed further on. It is cut again into two branches according to the two values of x_2 as it is the non-integer variable in the optimal solution of (24.10). Let

$$\begin{aligned} \mathcal{F}_3 &= \mathcal{F}_2 \cap \{\mathbf{x} \mid x_2 = 0\} \,, \\ \mathcal{F}_4 &= \mathcal{F}_2 \cap \{\mathbf{x} \mid x_2 = 1\} \,, \\ \mathcal{R}_3 &= \mathcal{R}_2 \cap \{\mathbf{x} \mid x_2 = 0\} \,, \\ \mathcal{R}_4 &= \mathcal{R}_2 \cap \{\mathbf{x} \mid x_2 = 1\} \,. \end{aligned}$$

The sets $\mathcal{F}_3$ and $\mathcal{R}_3$ are containing the feasible solution of the original problems such that x_3 is fixed to 1 and x_2 is fixed to 0. In the sets $\mathcal{F}_4$ and $\mathcal{R}_4$ both variables are fixed to 1. The optimal solution of the first relaxed problem, i.e.

$$\begin{aligned} \max 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ \mathbf{x} \in \mathcal{R}_3 \end{aligned}$$

is

$$x_1^* = 1,\ x_2^* = 0,\ x_3^* = 1,\ x_4^* = 1,\ x_5^* = 0 \,.$$

As it is integer it is also the optimal solution of the problem

$$\begin{aligned} \max 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ \mathbf{x} \in \mathcal{F}_3 \,. \end{aligned}$$

The optimal objective function value is 65. The branch of the sets $\mathcal{F}_3$ and $\mathcal{R}_3$ is completely fathomed, i.e. it is not possible to find a better solution in it.

The other new branch is when both x_2 and x_3 are fixed to 1. If the objective function is optimized on $\mathcal{R}_4$ then the optimal solution is

$$x_1^* = \frac{7}{8},\; x_2^* = x_3^* = 1,\; x_4^* = x_5^* = 0\,.$$

Applying the same technique again two branches are defined by the sets

$$\mathcal{F}_5 = \mathcal{F}_4 \cap \{\mathbf{x} \mid x_1 = 0\},\; \mathcal{F}_6 = \mathcal{F}_4 \cap \{\mathbf{x} \mid x_1 = 1\},$$

$$\mathcal{R}_5 = \mathcal{R}_4 \cap \{\mathbf{x} \mid x_2 = 0\},\; \mathcal{R}_6 = \mathcal{R}_4 \cap \{\mathbf{x} \mid x_2 = 1\}\,.$$

The optimal solution of the branch of $\mathcal{R}_5$ is

$$x_1^* = 0,\; x_2^* = x_3^* = x_4^* = 1,\; x_5^* = \frac{1}{19}\,.$$

The optimal value is 63.32. It is strictly less than the objective function value of the feasible solution found in the branch of $\mathcal{R}_3$. Therefore it cannot contain an optimal solution. Thus its further exploration can be omitted although the best feasible solution of the branch is still not known. The branch of $\mathcal{R}_6$ is infeasible as objects 1, 2, and 3 are overusing the knapsack. Traditionally this fact is denoted by using $-\infty$ as optimal objective function value.

At this moment there is only one branch which is still unfathomed. It is the branch of $\mathcal{R}_1$. The upper bound here is 65 which is equal to the objective function value of the found feasible solution. One can immediately conclude that this feasible solution is optimal. If there is no need for alternative optimal solutions then the exploration of this last branch can be abandoned and the method is finished. If alternative optimal solutions are required then the exploration must be continued. The non-integer variable in the optimal solution of the branch is x_5. The subbranches referred later as the 7th and 8th branches, defined by the equations $x_5 = 0$ and $x_5 = 1$, give the upper bounds 56 and 61, respectively. Thus they do not contain any optimal solution and the method is finished.

24.1.3. Properties in the calculation of the numerical example

The calculation is revisited to emphasize the general underlying logic of the method. The same properties are used in the next section when the general frame of B&B is discussed.

Problem (24.6) is a difficult one. Therefore the very similar but much easier Problem (24.7) has been solved instead of (24.6). A priori it was not possible to exclude the case that the optimal solution of (24.7) is the optimal solution of (24.6) as well. Finally it turned out that the optimal solution of (24.7) does not satisfy all constraints of (24.6) thus it is not optimal there. But the calculation was not useless, because an upper bound of the optimal value of (24.6) has been obtained. These properties are reflected in the definition of *relaxation* in the next section.

As the relaxation did not solved Problem (24.6) therefore it was divided into

Subproblems (24.9) and (24.11). Both subproblems have their own optimal solution and the better one is the optimal solution of (24.6). They are still too difficult to be solved directly, therefore relaxations were generated to both of them. These problems are (24.8) and (24.10). The nature of (24.8) and (24.10) from mathematical point of view is the same as of (24.7).

Notice that the union of the sets of the feasible solutions of (24.8) and (24.10) is a proper subset of the relaxation (24.7), i.e.

$$\mathcal{R}_1 \cup \mathcal{R}_2 \subset \mathcal{R}_0 \,.$$

Moreover the two subsets have no common element, i.e.

$$\mathcal{R}_1 \cap \mathcal{R}_2 = \emptyset \,.$$

It is true for all other cases, as well. The reason is that the branching, i.e. the determination of the Subproblems (24.9) and (24.11) was made in a way that the optimal solution of the relaxation, i.e. the optimal solution of (24.7), was cut off.

The branching policy also has consequences on the upper bounds. Let $\nu(\mathcal{S})$ be the optimal value of the problem where the objective function is unchanged and the set of feasible solutions is $\mathcal{S}$. Using this notation the optimal objective function values of the original and the relaxed problems are in the relation

$$\nu(\mathcal{F}) \leq \nu(\mathcal{R}) \,.$$

If a subset $\mathcal{R}_k$ is divided into $\mathcal{R}_p$ and $\mathcal{R}_q$ then

$$\nu(\mathcal{R}_k) \geq \max\{\nu(\mathcal{R}_p), \nu(\mathcal{R}_q)\} \,. \tag{24.12}$$

Notice that in the current Problem (24.12) is always satisfied with strict inequality

$$\begin{aligned}
\nu(\mathcal{R}_0) &> \max\{\nu(\mathcal{R}_1), \nu(\mathcal{R}_2)\} \,, \\
\nu(\mathcal{R}_1) &> \max\{\nu(\mathcal{R}_7), \nu(\mathcal{R}_8)\} \,, \\
\nu(\mathcal{R}_2) &> \max\{\nu(\mathcal{R}_3), \nu(\mathcal{R}_4)\} \,, \\
\nu(\mathcal{R}_4) &> \max\{\nu(\mathcal{R}_5), \nu(\mathcal{R}_6)\} \,.
\end{aligned}$$

(The values $\nu(\mathcal{R}_7)$ and $\nu(\mathcal{R}_8)$ were mentioned only.) If the upper bounds of a certain quantity are compared then one can conclude that the smaller the better as it is closer to the value to be estimated. An equation similar to (24.12) is true for the non-relaxed problems, i.e. if $\mathcal{F}_k = \mathcal{F}_p \cup \mathcal{F}_q$ then

$$\nu(\mathcal{F}_k) = \max\{\nu(\mathcal{F}_p), \nu(\mathcal{F}_q)\} \,, \tag{24.13}$$

but because of the difficulty of the solution of the problems, practically it is not possible to use (24.13) for getting further information.

A subproblem is fathomed and no further investigation of it is needed if either

- its integer (non-relaxed) optimal solution is obtained, like in the case of $\mathcal{F}_3$, or
- it is proven to be infeasible as in the case of $\mathcal{F}_6$, or

- its upper bound is not greater than the value of the best known feasible solution (cases of $\mathcal{F}_1$ and $\mathcal{F}_5$).

If the first or third of these conditions are satisfied then all feasible solutions of the subproblem are enumerated in an implicit way.

The subproblems which are generated in the same iteration, are represented by two branches on the enumeration tree. They are siblings and have the same parent. Figure 24.1 visualize the course of the calculations using the parent–child relation.

The enumeration tree is modified by constructive steps when new branches are formed and also by reduction steps when some branches can be deleted as one of the three above-mentioned criteria are satisfied. The method stops when no subset remained which has to be still fathomed.

24.1.4. How to accelerate the method

As it was mentioned in the introduction of the chapter, B&B and implicit enumeration can co-operate easily. Implicit enumeration uses so-called tests and obtains consequences on the values of the variables. For example if x_3 is fixed to 1 then the knapsack inequality immediately implies that x_5 must be 0, otherwise the capacity of the tourist is overused. It is true for the whole branch 2.

On the other hand if the objective function value must be at least 65, which is the value of the found feasible solution then it possible to conclude in branch 1 that the fifth object must be in the knapsack, i.e. x_5 must be 1, as the total value of the remaining objects 1, 2, and 4 is only 56.

Why such consequences accelerate the algorithm? In the example there are 5 binary variables, thus the number of possible cases is $32 = 2^5$. Both branches 1 and 2 have 16 cases. If it is possible to determine the value of a variable, then the number of cases is halved. In the above example it means that only 8 cases remain to be investigated in both branches. This example is a small one. But in the case of larger problems the acceleration process is much more significant. E.g. if in a branch there are 21 free, i.e. non-fixed, variables but it is possible to determine the value of one of them then the investigation of 1 048 576 cases is saved. The application of the tests needs some extra calculation, of course. Thus a good trade-off must be found.

The use of information provided by other tools is further discussed in Section 24.5.

Exercises

24.1-1 What is the suggestion of the optimal solution of a Knapsack Problem in connection of an object having (a) negative weight and positive value, (b) positive weight and negative value?

24.1-2 Show that an object of a knapsack problem having negative weight and negative value can be substituted by an object having positive weight and positive value such that the two knapsack problems are equivalent. (*Hint.* Use complementary variable.)

24.1-3 Solve Problem (24.6) with a branching strategy such that an integer valued variable is used for branching provided that such a variable exists.

24.2. The general frame of the B&B method

The aim of this section is to give a general description of the B&B method. Particular realizations of the general frame are discussed in later sections.

B&B is based on the notion of ***relaxation.*** It has not been defined yet. As there are several types of relaxations the first subsection is devoted to this notion. The general frame is discussed in the second subsection.

24.2.1. Relaxation

Relaxation is discussed in two steps. There are several techniques to define relaxation to a particular problem. There is no rule for choosing among them. It depends on the design of the algorithm which type serves the algorithm well. The different types are discussed in the first part titled "Relaxations of a particular problem". In the course of the solution of Problem (24.6) subproblems were generated which were still knapsack problems. They had their own relaxations which were not totally independent from the relaxations of each other and the main problem. The expected common properties and structure is analyzed in the second step under the title "Relaxation of a problem class".

Relaxations of a particular problem The description of Problem (24.6) consists of three parts: (1) the objective function, (2) the algebraic constraints, and (3) the requirement that the variables must be binary. This structure is typical for optimization problems. In a general formulation an optimization problem can be given as

$$\max f(\mathbf{x}) \tag{24.14}$$

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{b} \tag{24.15}$$

$$\mathbf{x} \in \mathcal{X} \,. \tag{24.16}$$

Relaxing the non-algebraic constraints The underlying logic of generating relaxation (24.7) is that constraint (24.16) has been substituted by a looser one. In the particular case it was allowed that the variables can take any value between 0 and 1. In general (24.16) is replaced by a requirement that the variables must belong to a set, say $\mathcal{Y}$, which is larger than $\mathcal{X}$, i.e. the relation $\mathcal{X} \subseteq \mathcal{Y}$ must hold. More formally the relaxation of Problem (24.14)-(24.16) is the problem

$$\max f(\mathbf{x}) \tag*{(24.14)}$$

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{b} \tag*{(24.15)}$$

$$\mathbf{x} \in \mathcal{Y}. \tag{24.17}$$

This type of relaxation can be applied if a large amount of difficulty can be eliminated by changing the nature of the variables.

Relaxing the algebraic constraints There is a similar technique such that (24.16) the inequalities (24.15) are relaxed instead of the constraints. A natural way of this type of relaxation is the following. Assume that there are m inequalities in (24.15). Let $\lambda_i \geq 0$ $(i = 1, \ldots, m)$ be fixed numbers. Then any $\mathbf{x} \in \mathcal{X}$ satisfying (24.15) also satisfies the inequality

$$\sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) \leq \sum_{i=1}^{m} \lambda_i b_i \,. \tag{24.18}$$

Then the relaxation is the optimization of the (24.14) objective function under the conditions (24.18) and (24.16). The name of the inequality (24.18) is ***surrogate constraint.***

The problem

$$\begin{array}{lcccccccccccc} \max & 23x_1 & + & 19x_2 & + & 28x_3 & + & 14x_4 & + & 44x_5 & & \\ & 5x_1 & + & 4x_2 & + & 6x_3 & + & 3x_4 & + & 5x_5 & \leq & 14 \\ & 2x_1 & - & 2x_2 & - & 3x_3 & + & 5x_4 & + & 6x_5 & \leq & 4 \\ & 1x_1 & + & 5x_2 & + & 8x_3 & - & 2x_4 & + & 8x_5 & \leq & 7 \\ & & & & & \multicolumn{5}{c}{x_1, x_2, x_3, x_4, x_5 = 0 \text{ or } 1} & & \end{array} \tag{24.19}$$

is a general zero-one optimization problem. If $\lambda_1 = \lambda_2 = \lambda_3 = 1$ then the relaxation obtained in this way is Problem (24.6). Both problems belong to NP-complete classes. However the knapsack problem is significantly easier from practical point of view than the general problem, thus the relaxation may have sense. Notice that in this particular problem the optimal solution of the knapsack problem, i.e. (1,0,1,1,0), satisfies the constraints of (24.19), thus it is also the optimal solution of the latter problem.

Surrogate constraint is not the only option in relaxing the algebraic constraints. A region defined by nonlinear boundary surfaces can be approximated by tangent planes. For example if the feasible region is the unit circuit which is described by the inequality

$$x_1^2 + x_2^2 \leq 1$$

can be approximated by the square

$$-1 \leq x_1,\, x_2 \leq 1\,.$$

If the optimal solution on the enlarged region is e.g. the point (1,1) which is not in the original feasible region then a *cut* must be found which cuts it from the relaxed region but it does not cut any part of the original feasible region. It is done e.g. by the inequality

$$x_1 + x_2 \leq \sqrt{2}\,.$$

A new relaxed problem is defined by the introduction of the cut. The method is similar to one of the method relaxing of the objective function discussed below.

Relaxing the objective function In other cases the difficulty of the problem is caused by the objective function. If it is possible to use an easier objective function, say $h(\mathbf{x})$, but to obtain an upper bound the condition

$$\forall\, \mathbf{x} \in \mathcal{X} :\; h(\mathbf{x}) \geq f(\mathbf{x}) \tag{24.20}$$

must hold. Then the relaxation is

$$\max h(\mathbf{x}) \tag{24.21}$$

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{b} \tag{24.15}$$

$$\mathbf{x} \in \mathcal{X}\,. \tag{24.16}$$

This type of relaxation is typical if B&B is applied in (continuous) nonlinear optimization. An important subclass of the nonlinear optimization problems is the so-called convex programming problem. It is again a relatively easy subclass. Therefore it is reasonable to generate a relaxation of this type if it is possible. A Problem (24.14)-(24.16) is a convex programming problem, if $\mathcal{X}$ is a convex set, the functions $g_i(\mathbf{x})$ $(i = 1, \ldots, m)$ are convex and the objective function $f(\mathbf{x})$ is concave. Thus the relaxation can be a convex programming problem if only the last condition is violated. Then it is enough to find a concave function $h(\mathbf{x})$ such that (24.20) is satisfied.

For example the single variable function $f(x) = 2x^2 - x^4$ is not concave in the interval $[\frac{-\sqrt{3}}{3}, \frac{\sqrt{3}}{3}]$.[1] Thus if it is the objective function in an optimization problem it might be necessary that it is substituted by a concave function $h(x)$ such that $\forall x \in [\frac{-\sqrt{3}}{3}, \frac{\sqrt{3}}{3}] : f(x) \leq h(x)$. It is easy to see that $h(x) = \frac{8}{9} - x^2$ satisfies the requirements.

Let $\mathbf{x}^*$ be the optimal solution of the relaxed problem (24.21), (24.15), and (24.16). It solves the original problem if the optimal solution has the same objective function value in the original and relaxed problems, i.e. $f(\mathbf{x}^*) = h(\mathbf{x}^*)$.

Another reason why this type of relaxation is applied that in certain cases the objective function is not known in a closed form, however it can be determined in any given point. It might happen even in the case if the objective function is concave. Assume that the value of $f(\mathbf{x})$ is known in the points $\mathbf{y}_1, \ldots, \mathbf{y}_k$. If $f(\mathbf{x})$ concave then it is smooth, i.e. its gradient exists. The gradient determines a tangent plane which is above the function. The equation of the tangent plane in point $\mathbf{y_p}$ is[2]

$$\nabla(f(\mathbf{y_p}))(\mathbf{x} - \mathbf{y_p}) = 0.$$

Hence in all points of the domain of the function $f(\mathbf{x})$ we have that

$$h(\mathbf{x}) = \min\left\{f(\mathbf{y_p}) + \nabla(f(\mathbf{y_p}))(\mathbf{x} - \mathbf{y_p}) \mid p = 1, \ldots, k\right\} \geq f(\mathbf{x}).$$

Obviously the function $h(\mathbf{x})$ is an approximation of function $f(\mathbf{x})$.

[1]A continuous function is concave if its second derivative is negative. $f^{''}(x) = 4 - 12x^2$ which is positive in the open interval $\left(\frac{-\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right)$.

[2]The gradient is considered being a row vector.

The idea if the method is illustrated on the following numerical example. Assume that an “unknown” concave function is to be maximized on the [0,5] closed interval. The method can start from any point of the interval which is in the feasible region. Let 0 be the starting point. According to the assumptions although the closed formula of the function is not known, it is possible to determine the values of function and its derivative. Now the values $f(0) = -4$ and $f^{'}(0) = 4$ are obtained. The general formula of the tangent line in the point $(x_0, f(x_0))$ is

$$y = f^{'}(x_0)(x - x_0) + f(x_0).$$

Hence the equation of the first tangent line is $y = 4x-4$ giving the first optimization problem as

$$\begin{gathered} \max h \\ h \leq 4x - 4 \\ x \in [0, 5]. \end{gathered}$$

As $4x - 4$ is a monotone increasing function, the optimal solution is $x = 5$. Then the values $f(5) = -9$ and $f^{'}(5) = -6$ are provided by the method calculating the function. The equation of the second tangent line is $y = -6x + 21$. Thus the second optimization problem is

$$\begin{gathered} \max h \\ h \leq 4x - 4, \quad h \leq -6x + 21 \\ x \in [0, 5]. \end{gathered}$$

As the second tangent line is a monotone decreasing function, the optimal solution is in the intersection point of the two tangent lines giving $x = 2.5$. Then the values $f(2.5) = -0.25$ and $f^{'}(2.5) = -1$ are calculated and the equation of the tangent line is $y = -x + 2.25$. The next optimization problem is

$$\begin{gathered} \max h \\ h \leq 4x - 4, \quad h \leq -6x + 21, \quad h \leq -x + 2.25 \\ x \in [0, 5]. \end{gathered}$$

The optimal solution is $x = 1.25$. It is the intersection point of the first and third tangent lines. Now both new intersection points are in the interval [0,5]. In general some intersection points can be infeasible. The method goes in the same way further on. The approximated “unknow” function is $f(x) = -(x - 2)^2$.

The Lagrange Relaxation Another relaxation called ***Lagrange relaxation.*** In that method both the objective function and the constraints are modified. The underlying idea is the following. The variables must satisfy two different types of constraints, i.e. they must satisfy both (24.15) and (24.16). The reason that the constraints are written in two parts is that the nature of the two sets of constraints is different. The difficulty of the problem caused by the requirement of *both* constraints. It is significantly easier to satisfy only one type of constraints. *So what about to eliminate one of them?*

Assume again that the number of inequalities in (24.15) is m. Let $\lambda_i \geq 0$ $(i = 1, \ldots, m)$ be fixed numbers. The Lagrange relaxation of the problem (24.14)- (24.16) is

$$\max\ f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i (b_i - g_i(\mathbf{x})) \tag{24.22}$$

$$\mathbf{x} \in \mathcal{X}\,. \tag{24.16}$$

Notice that the objective function (24.22) penalizes the violation of the constraints, e.g. trying to use too much resources, and rewards the saving of resources. The first set of constraints disappeared from the problem. In most of the cases the Lagrange relaxation is a much easier one than the original problem. In what follows Problem (24.14)- (24.16) is also denoted by (P) and the Lagrange relaxation is referred as $(L(\lambda))$. The notation reflects the fact that the Lagrange relaxation problem depends on the choice of λ_i's. The numbers λ_i's are called Lagrange multipliers.

It is not obvious that $(L(\lambda))$ is really a relaxation of (P). This relation is established by

Theorem 24.2 *Assume that both (P) and $(L(\lambda))$ have optimal solutions. Then for any nonnegative λ_i $(i = 1, \ldots, m)$ the inequality*

$$\nu(L(\lambda)) \geq \nu(P)$$

holds.

Proof The statement is that the optimal value of $(L(\lambda))$ is an upper bound of the optimal value of (P). Let $\mathbf{x}^*$ be the optimal solution of (P). It is obviously feasible in both problems. Hence for all i the inequalities $\lambda_i \geq 0$, $b_i \geq g_i(\mathbf{x}^*)$ hold. Thus $\lambda_i(b_i - g_i(\mathbf{x}^*)) \geq 0$ which implies that

$$f(\mathbf{x}^*) \leq f(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i (b_i - g_i(\mathbf{x}^*)).$$

Here the right-hand side is the objective function value of a feasible solution of $(L(\lambda))$, i.e.

$$\nu(P) = f(\mathbf{x}^*) \leq f(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i (b_i - g_i(\mathbf{x}^*)) \leq \nu(L(\lambda))\,.$$

■

There is another connection between (P) and $(L(\lambda))$ which is also important from the point of view of the notion of relaxation.

Theorem 24.3 *Let $\mathbf{x}_L$ be the optimal solution of the Lagrange relaxation. If*

$$\mathbf{g}(\mathbf{x}_L) \leq \mathbf{b} \tag{24.23}$$

and

$$\sum_{i=1}^{m} \lambda_i (b_i - g_i(\mathbf{x}_L)) = 0 \tag{24.24}$$

then $\mathbf{x}_L$ *is an optimal solution of* (P).

Proof (24.23) means that $\mathbf{x}_L$ is a feasible solution of (P). For any feasible solution $\mathbf{x}$ of (P) it follows from the optimality of $\mathbf{x}_L$ that

$$f(\mathbf{x}) \leq f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i (b_i - g_i(\mathbf{x})) \leq f(\mathbf{x}_L) + \sum_{i=1}^{m} \lambda_i (b_i - g_i(\mathbf{x}_L)) = f(\mathbf{x}_L)\,,$$

i.e. $\mathbf{x}_L$ is at least as good as $\mathbf{x}$. ■

The importance of the conditions (24.23) and (24.24) is that they give an optimality criterion, i.e. if a point generated by the Lagrange multipliers satisfies them then it is optimal in the original problem. The meaning of (24.23) is that the optimal solution of the Lagrange problem is feasible in the original one and the meaning of (24.24) is that the objective function values of $\mathbf{x}_L$ are equal in the two problems, just as in the case of the previous relaxation. It also indicates that the optimal solutions of the two problems are coincident in certain cases.

There is a practical necessary condition for being a useful relaxation which is that the relaxed problem is easier to solve than the original problem. The Lagrange relaxation has this property. It can be shown on Problem (24.19). Let $\lambda_1 = 1$, $\lambda_2 = \lambda_3 = 3$. Then the objective function (24.22) is the following

$$\begin{aligned}
&(23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5) + (14 - 5x_1 - x_2 - 6x_3 - 3x_4 - 5x_5)\\
&+3(4 - 2x_1 - x_2 + 3x_3 - 5x_4 - 6x_5) + 3(7 - x_1 - 5x_2 - 8x_3 + 2x_4 - 8x_5)\\
&\quad = 47 + (23 - 5 - 6 - 3)x_1 + (19 - 1 - 3 - 15)x_2 + (28 - 6 + 9 - 24)x_3\\
&\qquad + (14 - 3 - 15 + 5)x_4 + (44 - 5 - 18 - 24)x_5\\
&\quad = 47 + 9x_1 + 0x_2 + 7x_3 + x_4 - 3x_5\,.
\end{aligned}$$

The only constraint is that all variables are binary. It implies that if a coefficient is positive in the objective function then the variable must be 1 in the optimal solution of the Lagrange problem, and if the coefficient is negative then the variable must be 0. As the coefficient of x_2 is zero, there are two optimal solutions: (1,0,1,1,0) and (1,1,1,1,0). The first one satisfies the optimality condition thus it is an optimal solution. The second one is infeasible.

What is common in all relaxation? They have three common properties.

1. All feasible solutions are also feasible in the relaxed problem.
2. The optimal value of the relaxed problem is an upper bound of the optimal value of the original problem.

3. There are cases when the optimal solution of the relaxed problem is also optimal in the original one.

The last property cannot be claimed for all particular case as then the relaxed problem is only an equivalent form of the original one and needs very likely approximately the same computational effort, i.e. it does not help too much. Hence the first two properties are claimed in the definition of the relaxation of a particular problem.

Definition 24.4 *Let f, h be two functions mapping from the n-dimensional Euclidean space into the real numbers. Further on let $\mathcal{U}, \mathcal{V}$ be two subsets of the n-dimensional Euclidean space. The problem*

$$\max\{h(\mathbf{x}) \mid \mathbf{x} \in \mathcal{V}\} \tag{24.25}$$

is a relaxation of the problem

$$\max\{f(\mathbf{x}) \mid \mathbf{x} \in \mathcal{U}\} \tag{24.26}$$

if
(i) $\mathcal{U} \subset \mathcal{V}$ and
(ii) it is known a priori, *i.e. without solving the problems that $\nu(24.25) \geq \nu(24.26)$.*

Relaxation of a problem class No exact definition of the notion of *problem class* will be given. There are many problem classes in optimization. A few examples are the knapsack problem, the more general zero-one optimization, the traveling salesperson problem, linear programming, convex programming, etc. In what follows problem class means only an infinite set of problems.

One key step in the solution of (24.6) was that the problem was divided into subproblems and even the subproblems were divided into further subproblems, and so on.

The division must be carried out in a way such that the subproblems belong to the same problem class. By fixing the value of a variable the knapsack problem just becomes another knapsack problem of lesser dimension. The same is true for almost all optimization problems, i.e. a restriction on the value of a single variable (introducing either a lower bound, or upper bound, or an exact value) creates a new problem in the same class. But restricting a single variable is not the only possible way to divide a problem into subproblems. Sometimes special constraints on a set of variables may have sense. For example it is easy to see from the first constraint of (24.19) that at most two out of the variables x_1, x_3, and x_5 can be 1. Thus it is possible to divide it into two subproblems by introducing the new constraint which is either $x_1 + x_3 + x_5 = 2$, or $x_1 + x_3 + x_5 \leq 1$. The resulted problems are still in the class of binary optimization. The same does not work in the case of the knapsack problem as it must have only one constraint, i.e. if a second inequality is added to the problem then the new problem is out of the class of the knapsack problems.

The division of the problem into subproblems means that the set of feasible solutions is divided into subsets not excluding the case that one or more of the subsets turn out to be empty set. $\mathcal{R}_5$ and $\mathcal{R}_6$ gave such an example.

Another important feature is summarized in formula (24.12). It says that the upper bound of the optimal value obtained from the undivided problem is at most as accurate as the upper bound obtained from the divided problems.

Finally, the further investigation of the subset $\mathcal{F}_1$ could be abandoned as $\mathcal{R}_1$ was not giving a higher upper bound as the objective function value of the optimal solution on $\mathcal{R}_3$ which lies at the same time in $\mathcal{F}_3$, too, i.e. the subproblem defined on the set $\mathcal{F}_3$ was solved.

The definition of the relaxation of a problem class reflects the fact that relaxation and defining subproblems (branching) are not completely independent. In the definition it is assumed that the branching method is a priori given.

Definition 24.5 *Let $\mathcal{P}$ and $\mathcal{Q}$ be two problem classes. Class $\mathcal{Q}$ is a relaxation of class $\mathcal{P}$ if there is a map R with the following properties.*

1. *R maps the problems of $\mathcal{P}$ into the problems of $\mathcal{Q}$.*
2. *If a problem $(P) \in \mathcal{P}$ is mapped into $(Q) \in \mathcal{Q}$ then (Q) is a relaxation of (P) in the sense of Definition 24.4.*
3. *If (P) is divided into (P_1),…,(P_k) and these problems are mapped into (Q_1),…,(Q_k), then the inequality*
$$\nu(Q) \geq \max\{\nu(Q_1), \ldots, \nu(Q_k)\} \tag{24.27}$$
holds.
4. *There are infinite many pairs (P), (Q) such that an optimal solution of (Q) is also optimal in (P).*

24.2.2. The general frame of the B&B method

As the Reader has already certainly observed B&B divides the problem into subproblems and tries to fathom each subproblem by the help of a relaxation. A subproblem is fathomed in one of the following cases:

1. The optimal solution of the relaxed subproblem satisfies the constraints of the unrelaxed subproblem and its relaxed and non-relaxed objective function values are equal.
2. The infeasibility of the relaxed subproblem implies that the unrelaxed subproblem is infeasible as well.
3. The upper bound provided by the relaxed subproblem is less (in the case if alternative optimal solution are sought) or less or equal (if no alternative optimal solution is requested) than the objective function value of the best known feasible solution.

The algorithm can stop if all subsets (branches) are fathomed. If nonlinear programming problems are solved by B&B then the finiteness of the algorithm cannot be always guaranteed.

In a typical iteration the algorithm executes the following steps.

- It selects a leaf of the branching tree, i.e. a subproblem not divided yet into further subproblems.
- The subproblem is divided into further subproblems (branches) and their relaxations are defined.
- Each new relaxed subproblem is solved and checked if it belongs to one of the above-mentioned cases. If so then it is fathomed and no further investigation is needed. If not then it must be stored for further branching.
- If a new feasible solution is found which is better than the so far best one, then even stored branches having an upper bound less than the value of the new best feasible solution can be deleted without further investigation.

In what follows it is supposed that the relaxation satisfies definition 24.5.

The original problem to be solved is

$$\max f(\mathbf{x}) \tag{24.14}$$

$$\mathbf{g}(\mathbf{x}) \le \mathbf{b} \tag{24.15}$$

$$\mathbf{x} \in \mathcal{X}. \tag{24.16}$$

Thus the set of the feasible solutions is

$$\mathcal{F} = \mathcal{F}_0 = \{\mathbf{x} \mid \mathbf{g}(\mathbf{x}) \le \mathbf{b}; \mathbf{x} \in \mathcal{X}\}. \tag{24.28}$$

The relaxed problem satisfying the requirements of definition 24.5 is

$$\max h(\mathbf{x})$$

$$\mathbf{k}(\mathbf{x}) \le \mathbf{b}$$

$$\mathbf{x} \in \mathcal{Y},$$

where $\mathcal{X} \subseteq \mathcal{Y}$ and for all points of the domain of the objective functions $f(\mathbf{x}) \le h(\mathbf{x})$ and for all points of the domain of the constraint functions $\mathbf{k}(\mathbf{x}) \le h(\mathbf{x})$. Thus the set of the feasible solutions of the relaxation is

$$\mathcal{R} = \mathcal{R}_0 = \{\mathbf{x} \mid \mathbf{k}(\mathbf{x}) \le \mathbf{b}; \mathbf{x} \in \mathcal{Y}\}.$$

Let $\mathcal{F}_k$ be a previously defined subset. Suppose that it is divided into the subsets $\mathcal{F}_{t+1}, \ldots, \mathcal{F}_{t+p}$, i.e.

$$\mathcal{F}_k = \bigcup_{l=1}^{p} \mathcal{F}_{t+l}.$$

Let $\mathcal{R}_k$ and $\mathcal{R}_{t+1}, \ldots, \mathcal{R}_{t+p}$ be the feasible sets of the relaxed subproblems. To satisfy the requirement (24.27) of definition 24.5 it is assumed that

$$\mathcal{R}_k \supseteq \bigcup_{l=1}^{p} \mathcal{R}_{t+l}.$$

The subproblems are identified by their sets of feasible solutions. The unfathomed subproblems are stored in a list. The algorithm selects a subproblem from the list for further branching. In the formal description of the general frame of B&B the following notations are used.

$\hat{z}$	the objective function value of the best feasible solution found so far
$\mathcal{L}$	the list of the unfathomed subsets of feasible solutions
t	the number of branches generated so far
$\mathcal{F}_0$	the set of all feasible solutions
r	the index of the subset selected for branching
$p(r)$	the number of branches generated from $\mathcal{F}_r$
$\mathbf{x}_i$	the optimal solution of the relaxed subproblem defined on $\mathcal{R}_i$
z_i	the upper bound of the objective function on subset $\mathcal{F}_i$
$\mathcal{L} + \mathcal{F}_i$	the operation of adding the subset $\mathcal{F}_i$ to the list $\mathcal{L}$
$\mathcal{L} - \mathcal{F}_i$	the operation of deleting the subset $\mathcal{F}_i$ from the list $\mathcal{L}$

Note that $y_i = \max\{h(\mathbf{x}) \mid \mathbf{x} \in \mathcal{R}_i\}$.

The frame of the algorithms can be found below. It simply describes the basic ideas of the method and does not contain any tool of acceleration.

Branch-and-Bound

```
 1  $\hat{z} \leftarrow -\infty$
 2  $\mathcal{L} \leftarrow \{ \mathcal{F}_0 \}$
 3  $t \leftarrow 0$
 4  while $\mathcal{L} \neq \emptyset$
 5        do determination of $r$
 6           $\mathcal{L} \leftarrow \mathcal{L} - \mathcal{F}_r$
 7           determination of $p(r)$
 8           determination of branching $\mathcal{F}_r \subset \mathcal{R}_1 \cup \ldots \cup \mathcal{R}_{p(r)}$
 9           for $i \leftarrow 1$ to $p(r)$ do
10               $\mathcal{F}_{t+i} \leftarrow \mathcal{F}_r \cap \mathcal{R}_i$
11               calculation of $(\mathbf{x}_{t+i}, z_{t+i})$
12               if $z_{t+i} > \hat{z}$
13                  then if $\mathbf{x}_{t+i} \in \mathcal{F}$
14                          then $\hat{z} \leftarrow z_{t+i}$
15                  else $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{F}_{t+i}$
16           $t \leftarrow t + p(r)$
17           for $i \leftarrow 1$ to $t$ do
18               if $z_i \leq \hat{z}$
19                  then $\mathcal{L} \leftarrow \mathcal{L} - \mathcal{F}_i$
20  return $\mathbf{x}$
```

The operations in rows 5, 7, 8, and 11 depend on the particular problem class and on the skills of the designer of the algorithm. The relaxed subproblem is solved in row 14. A detailed example is discussed in the next section. The handling of the list needs also careful consideration. Section 24.4 is devoted to this topic.

The loop in rows 17 and 18 can be executed in an implicit way. If the selected subproblem in row 5 has a low upper bound, i.e. $z_r \leq \hat{z}$ then the subproblem is fathomed and a new subproblem is selected.

However the most important issue is the number of required operations including the finiteness of the algorithm. The method is not necessarily finite. Especially nonlinear programming has infinite versions of it. Infinite loop may occur even in the case if the number of the feasible solutions is finite. The problem can be caused by an incautious branching procedure. A branch can belong to an empty set. Assume that that the branching procedure generates subsets from $\mathcal{F}_r$ such that one of the subsets $\mathcal{F}_{t+1}, ..., \mathcal{F}_{t+p(r)}$ is equal to $\mathcal{F}_r$ and the other ones are empty sets. Thus there is an index i such that

$$\mathcal{F}_{t+i} \ = \ \mathcal{F}_r, \ \ \mathcal{F}_{t+1} = ... = \mathcal{F}_{t+i-1} = \mathcal{F}_{t+i+1} = ... = \mathcal{F}_{t+p(r)} = \emptyset \,. \qquad (24.29)$$

If the same situation is repeated at the branching of $\mathcal{F}_{t+i}$ then an infinite loop is possible.

Assume that a zero-one optimization problem of n variables is solved by B&B and the branching is made always according to the two values of a free variable. Generally it is not known that how large is the number of the feasible solutions. There are at most 2^n feasible solutions as it is the number of the zero-one vectors. After the first branching there are at most 2^{n-1} feasible solutions in the two first level leaves, each. This number is halved with each branching, i.e. in a branch on level k there are at most 2^{n-k} feasible solutions. It implies that on level n there is at most $2^{n-n} = 2^0 = 1$ feasible solution. As a matter of fact on that level there is exactly 1 zero-one vector and it is possible to decide whether or not it is feasible. Hence after generating all branches on level n the problem can be solved. This idea is generalized in the following finiteness theorem. While formulating the statement the previous notations are used.

Theorem 24.6 *Assume that*
(i) The set $\mathcal{F}$ is finite.
(ii) There is a finite set $\mathcal{U}$ such that the following conditions are satisfied. If a subset $\hat{\mathcal{F}}$ is generated in the course of the branch and bound method then there is a subset $\hat{\mathcal{U}}$ of $\mathcal{U}$ such that $\hat{\mathcal{F}} \subseteq \hat{\mathcal{U}}$. Furthermore if the branching procedure creates the cover $\mathcal{R}_1 \cup \ldots \cup \mathcal{R}_p \supseteq \hat{\mathcal{F}}$ then $\hat{\mathcal{U}}$ has a partitioning such that

$$\hat{\mathcal{U}} = \hat{\mathcal{U}}_1 \cup \cdots \cup \hat{\mathcal{U}}_p, \; \hat{\mathcal{U}}_i \cap \hat{\mathcal{U}}_j = \emptyset (i \neq j)$$
$$\hat{\mathcal{F}} \cap \hat{\mathcal{R}} j \subseteq \hat{\mathcal{U}}_j (j = 1, \ldots, p)$$

and moreover

$$1 \leq \mid \hat{\mathcal{U}}_j \mid < \mid \hat{\mathcal{U}} \mid \quad (j = 1, \ldots, p) \,. \qquad (24.30)$$

(iii) If a set $\hat{\mathcal{U}}$ belonging to set $\hat{\mathcal{F}}$ has only a single element then the relaxed subproblem solves the unrelaxed subproblem as well.

Then the Branch-and-Bound *procedure stops after finite many steps. If $\hat{z} = -\infty$ then there is no feasible solution. Otherwise $\hat{z}$ is equal to the optimal objective function value.*

Remark. Notice that the binary problems mentioned above with $\hat{\mathcal{U}}_j$'s of type

$$\hat{\mathcal{U}}_j = \{\mathbf{x} \in \{0,1\}^n \mid x_k = \delta_{kj},\ k \in I_j\},$$

where $I_j \subset \{1,2,\dots,n\}$ is the set of fixed variables and $\delta_{kj} \in \{0,1\}$ is a fixed value, satisfy the conditions of the theorem.

Proof Assume that the procedure Branch-and-Bound executes infinite many steps. As the set $\mathcal{F}$ is finite it follows that there is at least one subset of $\mathcal{F}$ say $\mathcal{F}_r$ such that it defines infinite many branches implying that the situation described in (24.29) occurs infinite many times. Hence there is an infinite sequence of indices, say $r_0 = r < r_1 < \cdots$, such that $\mathcal{F}_{r_{j+1}}$ is created at the branching of $\mathcal{F}_{r_j}$ and $\mathcal{F}_{r_{j+1}} = \mathcal{F}_{r_j}$. On the other hand the parallel sequence of the $\mathcal{U}$ sets must satisfy the inequalities

$$|\,\mathcal{U}_{r_0}\,| > |\,\mathcal{U}_{r_1}\,| > \cdots \geq 1\,.$$

It is impossible because the $\mathcal{U}$s are finite sets.

The finiteness of $\mathcal{F}$ implies that optimal solution exists if and only if $\mathcal{F}$ is nonempty, i.e. the problem cannot be unbounded and if feasible solution exist then the supremum of the objective function is its maximum. The initial value of $\hat{z}$ is $-\infty$. It can be changed only in Row 18 of the algorithm and if it is changed then it equals to the objective function value of a feasible solution. Thus if there is no feasible solution then it remains $-\infty$. Hence if the second half of the statement is not true, then at the end of the algorithm $\hat{z}$ equal the objective function value of a non-optimal feasible solution or it remains $-\infty$.

Let r be the maximal index such that $\mathcal{F}_r$ still contains the optimal solution. Then

$$z_r \geq \text{optimal value} > \hat{z}\,.$$

Hence it is not possible that the branch containing the optimal solution has been deleted from the list in the loop of Rows 22 and 23, as $z_r > \hat{z}$. It is also sure that the subproblem

$$\max\{f(\mathbf{x}) \mid \mathbf{x} \in \mathcal{F}_r\}$$

has not been solved, otherwise the equation $z_r = \hat{z}$ should hold. Then only one option remained that $\mathcal{F}_r$ was selected for branching once in the course of the algorithm. The optimal solution must be contained in one of its subsets, say $\mathcal{F}_{t+i}$ which contradicts the assumption that $\mathcal{F}_r$ has the highest index among the branches containing the optimal solution. ■

If an optimization problem contains only bounded integer variables then the sets $\mathcal{U}$s are the sets the integer vectors in certain boxes. In the case of some scheduling problems where the optimal order of tasks is to be determined even the relaxations have combinatorial nature because they consist of permutations. Then $\mathcal{U} = \mathcal{R}$ is also possible. In both of the cases Condition (iii) of the theorem is fulfilled in a natural way.

Exercises

24.2-1 Decide if the Knapsack Problem can be a relaxation of the Linear Binary Optimization Problem in the sense of Definition 24.5. Explain your solution regardless that your answer is YES or NO.

24.3. Mixed integer programming with bounded variables

Many decisions have both continuous and discrete nature. For example in the production of electric power the discrete decision is to switch on or not an equipment. The equipment can produce electric energy in a relatively wide range. Thus if the first decision is to switch on then a second decision must be made on the level of the produced energy. It is a continuous decision. The proper mathematical model of such problems must contain both discrete and continuous variables.

This section is devoted to the mixed integer linear programming problem with bounded integer variables. It is assumed that there are n variables and a subset of them, say $\mathcal{I} \subseteq \{1, \ldots, n\}$ must be integer. The model has m linear constraints in equation form and each integer variable has an explicit integer upper bound. It is also supposed that all variables must be nonnegative. More formally the mathematical problem is as follows.

$$\max \mathbf{c}^T\mathbf{x} \tag{24.31}$$

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{24.32}$$

$$\forall\, j \in \mathcal{I}: \; x_j \le g_j \tag{24.33}$$

$$x_j \ge 0 \;\; j = 1, \ldots, n \tag{24.34}$$

$$\forall\, j \in \mathcal{I}: \; x_j \;\text{ is integer }, \tag{24.35}$$

where $\mathbf{c}$ and $\mathbf{x}$ are n-dimensional vectors, $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{b}$ is an m-dimensional vector and finally all g_j $(j \in \mathcal{I})$ is a positive integer.

In the mathematical analysis of the problem below the the explicit upper bound constraints (24.33) will not be used. The Reader may think that they are formally included into the other algebraic constraints (24.32).

There are technical reasons that the algebraic constraints in (24.32) are claimed in the form of equations. Linear programming relaxation is used in the method. The linear programming problem is solved by the simplex method which needs this form. But generally speaking equations and inequalities can be transformed into one another in an equivalent way. Even in the numerical example discussed below inequality form is used.

First a numerical example is analyzed. The course of the method is discussed from geometric point of view. Thus some technical details remain concealed. Next simplex method and related topics are discussed. All technical details can be described only in the possession of them. Finally some strategic points of the algorithm are analyzed.

24.3.1. The geometric analysis of a numerical example

The problem to be solved is

$$\begin{array}{rrcrcrcr} \max & x_0 & = & 2x_1 & + & x_2 & & \\ & & & 3x_1 & - & 5x_2 & \leq & 0 \\ & & & 3x_1 & + & 5x_2 & \leq & 15 \\ & & & \multicolumn{5}{c}{x_1,\, x_2 \geq 0} \\ & & & \multicolumn{5}{c}{x_1,\, x_2 \text{ is integer}\,.} \end{array} \tag{24.36}$$

To obtain a relaxation the integrality constraints are omitted from the problem. Thus a linear programming problem of two variables is obtained.

The branching is made according to a non-integer variable. Both x_1 and x_2 have fractional values. To keep the number of branches as low as possible, only two new branches are created in a step.

The numbering of the branches is as follows. The original set of feasible solutions is No. 1. When the two new branches are generated then the branch belonging to the smaller values of the branching variable has the smaller number. The numbers are positive integers started at 1 and not skipping any integer. Branches having no feasible solution are numbered, too.

The optimal solution of the relaxation is $x_1 = 2.5$, $x_2 = 1.5$, and the optimal value is $\frac{13}{2}$ as it can be seen from figure 24.2. The optimal solution is the intersection point the lines determined by the equations

$$3x_1 - 5x_2 = 0$$

and

$$3x_1 + 5x_2 = 15\,.$$

If the branching is based on variable x_1 then they are defined by the inequalities

$$x_1 \leq 2 \quad \text{and} \quad x_1 \geq 3\,.$$

Notice that the maximal value of x_1 is 2.5. In the next subsection the problem is revisited. Then this fact will be observed from the simplex tableaux. Variable x_2 would create the branches

$$x_2 \leq 1 \quad \text{and} \quad x_2 \geq 2\,.$$

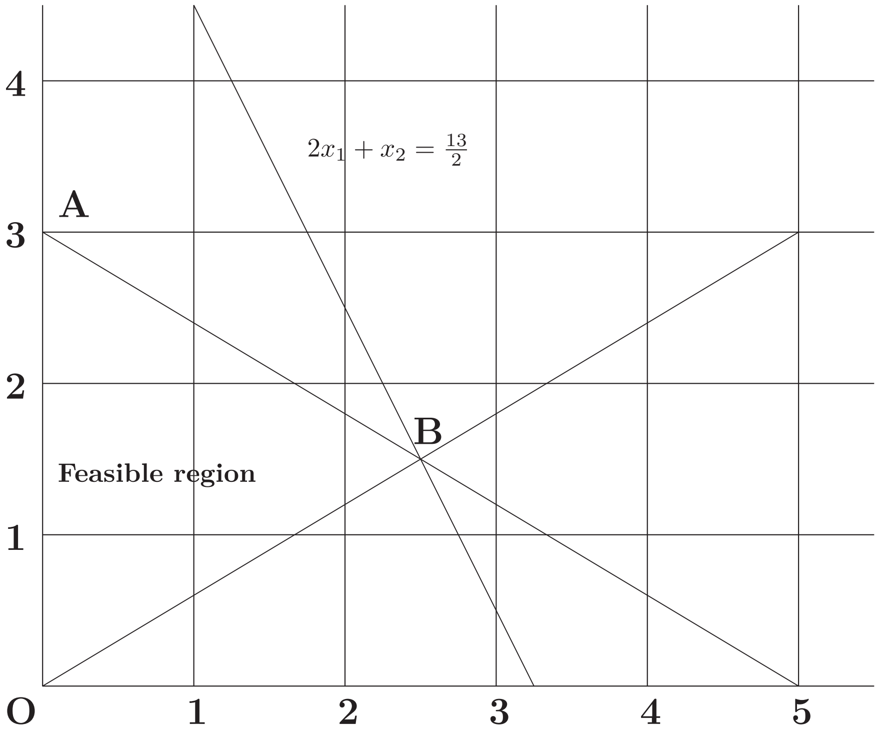


Figure 24.2 The geometry of linear programming relaxation of Problem (24.36) including the feasible region (triangle $\overline{\text{OAB}}$), the optimal solution ($x_1 = 2.5$, $x_2 = 1.5$), and the optimal level of the objective function represented by the line $2x_1 + x_2 = \frac{13}{2}$.

None of them is empty. Thus it is more advantageous the branch according to x_1. Geometrically it means that the set of the feasible solutions in the relaxed problem is cut by the line $x_1 = 2$. Thus the new set becomes the quadrangle $\overline{OACD}$ on Figure 24.3. The optimal solution on that set is $x_1 = 2$, $x_2 = 1.8$. It is point C on the figure.

Now branching is possible according only to variable x_2. Branches 4 and 5 are generated by the cuts $x_2 \leq 1$ and $x_2 \geq 2$, respectively. The feasible regions of the relaxed problems are $\overline{OHG}$ of Branch 4, and $\overline{AEF}$ of Branch 5. The method continues with the investigation of Branch 5. The reason will be given in the next subsection when the quickly calculable upper bounds are discussed. On the other hand it is obvious that the set $\overline{AEF}$ is more promising than $\overline{OHG}$ if the Reader takes into account the position of the contour, i.e. the level line, of the objective function on Figure 24.3. The algebraic details discussed in the next subsection serve to realize the decisions in higher dimensions what is possible to see in 2-dimension.

Branches 6 and 7 are defined by the inequalities $x_1 \leq 1$ and $x_1 \geq 2$, respectively. The latter one is empty again. The feasible region of Branch 6 is $\overline{AIJF}$. The optimal solution in this quadrangle is the Point I. Notice that there are only three integer points in $\overline{AIJF}$ which are (0,3), (0,2), and (1,2). Thus the optimal integer solution of

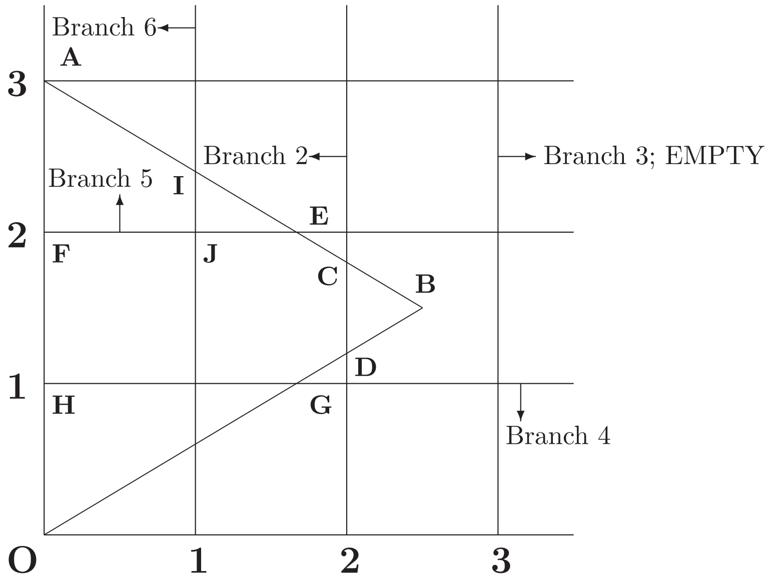


Figure 24.3 The geometry of the course of the solution. The co-ordinates of the points are: O=(0,0), A=(0,3), B=(2.5,1.5), C=(2,1.8), D=(2,1.2), E=($\frac{5}{3}$,2), F=(0,2), G=($\frac{5}{3}$,1), H=(0,1), I=(1,2.4), and J=(1,2). The feasible regions of the relaxation are as follows. Branch 1: $\overline{OAB}$, Branch 2: $\overline{OACD}$, Branch 3: empty set, Branch 4: $\overline{OHG}$, Branch 5: $\overline{AEF}$, Branch 6: $\overline{AIJF}$, Branch 7: empty set (not on the figure). Point J is the optimal solution.

this branch is (1,2). There is a technique which can help to leap over the continuous optimum. In this case it reaches directly point J, i.e. the optimal integer solution of the branch as it will be seen in the next section, too. Right now assume that the integer optimal solution with objective function value 4 is uncovered.

At this stage of the algorithm the only unfathomed branch is Branch 4 with feasible region $\overline{OHG}$. Obviously the optimal solution is point G=($\frac{5}{3}$,1). Its objective function value is $\frac{13}{3}$. Thus it cannot contain a better feasible solution than the known (1,2). Hence the algorithm is finished.

24.3.2. The linear programming background of the method

The first ever general method solving linear programming problems were discovered by George Dantzig and called *simplex method*. There are plenty of versions of the simplex method. The main tool of the algorithm is the so-called *dual simplex method*. Although simplex method is discussed in a previous volume, the basic knowledge is summarized here.

Any kind of simplex method is a so-called pivoting algorithm. *An important property of the pivoting algorithms is that they generate equivalent forms of the equation system and – in the case of linear programming – the objective function.* Practically it means that the algorithm works with equations. As many variables as many linearly independent equations exist are expressed with other variables and

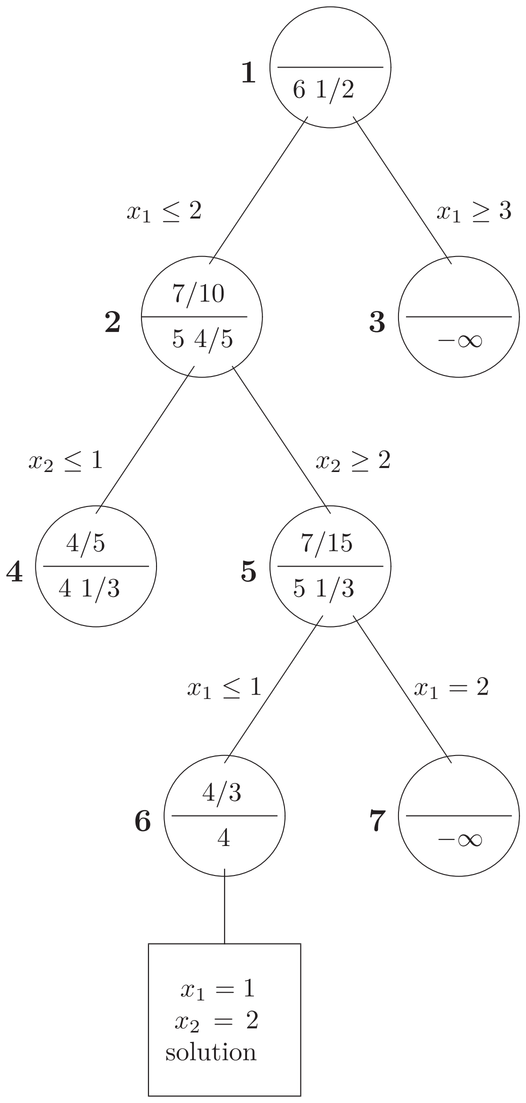


Figure 24.4 The course of the solution of Problem (24.36). The upper numbers in the circuits are explained in subsection 24.3.2. They are the corrections of the previous bounds obtained from the first pivoting step of the simplex method. The lower numbers are the (continuous) upper bounds obtained in the branch.

further consequences are drawn from the current equivalent form of the equations.

If there are inequalities in the problem then they are reformulated by introducing nonnegative slack variables. E.g. in case of LP-relaxation of Problem (24.36) the equivalent form of the problem is

$$\begin{array}{llccccccccc} \max & x_0 & = & 2x_1 & + & x_2 & + & 0x_3 & + & 0x_4 & & \\ & & & 3x_1 & - & 5x_2 & + & x_3 & + & 0x_4 & = & 0 \\ & & & 3x_1 & + & 5x_2 & + & 0x_3 & + & x_4 & = & 15 \\ & & & & & x_1,\ x_2\, x_3,\ x_4 \geq 0\,. \end{array} \tag{24.37}$$

Notice that all variables appear in all equations including the objective function, but it is allowed that some coefficients are zeros. The current version (24.37) can be considered as a form where the variables x_3 and x_4 are expressed by x_1 and x_2 and the expression is substituted into the objective function. If $x_1 = x_2 = 0$ then $x_3 = 0$ and $x_4 = 15$, thus the solution is feasible. Notice that the value of the objective function is 0 and if it is possible to increase the value of any of x_1 and x_2 and still getting a feasible solution then a better feasible solution is obtained. It is true, because the method uses equivalent forms of the objective function. The method obtains better feasible solution by pivoting. Let x_1 and x_2 be the two expressed variables. Skipping some pivot steps the equivalent form of (24.37) is

$$\begin{array}{llccccccccccc} \max & x_0 & = & 0x_1 & + & 0x_2 & - & \frac{7}{30}x_3 & - & \frac{13}{30}x_4 & + & \frac{13}{2} \\ & & & x_1 & + & 0x_2 & + & \frac{1}{6}x_3 & + & \frac{1}{6}x_4 & = & \frac{5}{2} \\ & & & 0x_1 & + & x_2 & - & \frac{1}{10}x_3 & + & \frac{1}{10}x_4 & = & \frac{3}{2} \\ & & & & & x_1,\ x_2\, x_3,\ x_4 \geq 0\,. \end{array} \tag{24.38}$$

That form has two important properties. First if $x_3 = x_4 = 0$ then $x_1 = \frac{5}{2}$ and $x_2 = \frac{3}{2}$, thus the solution is feasible, similarly to the previous case. Generally this property is called *primal feasibility*. Secondly, the coefficients of the non-expressed variables are negative in the objective function. It is called *dual feasibility*. It implies that if any of the non-expressed variables is positive in a feasible solution then that is worse than the current one. It is true again, because the current form of the objective function is equivalent to the original one. Thus the current value of the objective function which is $\frac{13}{2}$, is optimal.

In what follows the sign of maximization and the nonnegativity of the variables will be omitted from the problem but they are kept in mind.

In the general case it may be assumed without loss of generality that all equations are independent. Simplex method uses the form of the problem

$$\max x_0 = \mathbf{c}^T\mathbf{x} \tag{24.39}$$

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{24.40}$$

$$\mathbf{x} \geq \mathbf{0}\,, \tag{24.41}$$

where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{c}$ and $\mathbf{x}$ are n-dimensional vectors, and $\mathbf{b}$ is an m-dimensional vector. According to the assumption that all equations are independent, $\mathbf{A}$ has m linearly independent columns. They form a basis of the m-dimensional linear space. They also form an $m \times m$ invertible submatrix. It is denoted by $\mathbf{B}$. The inverse of $\mathbf{B}$ is $\mathbf{B}^{-1}$. Matrix $\mathbf{A}$ is partitioned into the basic and non-basic parts:

$\mathbf{A} = (\mathbf{B}, \mathbf{N})$ and the vectors $\mathbf{c}$ and $\mathbf{x}$ are partitioned accordingly. Hence

$$\mathbf{A}\mathbf{x} = \mathbf{B}\mathbf{x}_B + \mathbf{N}\mathbf{x}_N = \mathbf{b}\,.$$

The expression of the basic variables is identical with the multiplication of the equation by $\mathbf{B}^{-1}$ from left

$$\mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \mathbf{B}^{-1}\mathbf{B}\mathbf{x}_B + \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N = \mathbf{I}\mathbf{x}_B + \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N = \mathbf{B}^{-1}\mathbf{b},$$

where $\mathbf{I}$ is the unit matrix. Sometimes the equation is used in the form

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N\,. \tag{24.42}$$

The objective function can be transformed into the equivalent form

$$\mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N$$

$$\mathbf{c}_B^T(\mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N) + \mathbf{c}_N^T\mathbf{x}_N = \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{b} + (\mathbf{c}_N^T - \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{N})\mathbf{x}_N\,.$$

Notice that the coefficients of the basic variables are zero. If the non-basic variables are zero valued then the value of the basic variables is given by the equation

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}\,.$$

Hence the objective function value of the basic solution is

$$\mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N = \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{b} + \mathbf{c}_N^T\mathbf{0} = \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{b}\,. \tag{24.43}$$

Definition 24.7 *A vector* $\mathbf{x}$ *is a* **solution** *of Problem (24.39)-(24.41) if it satisfies the equation (24.40). It is a* **feasible solution** *or equivalently a* **primal feasible solution** *if it satisfies both (24.40) and (24.41). A solution* $\mathbf{x}$ *is a* **basic solution** *if the columns of matrix* $\mathbf{A}$ *belonging to the non-zero components of* $\mathbf{x}$ *are linearly independent. A basic solution is a* **basic feasible** *or equivalently a* **basic primal feasible** *solution if it is feasible. Finally a basic solution is* **basic dual feasible solution** *if*

$$\mathbf{c}_N^T - \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{N} \le \mathbf{0}^T\,. \tag{24.44}$$

The primal feasibility of a basic feasible solution is equivalent to

$$\mathbf{B}^{-1}\mathbf{b} \ge \mathbf{0}\,.$$

Let $\mathbf{a}_1, \ldots, \mathbf{a}_n$ be the column vectors of matrix $\mathbf{A}$. Further on let $\mathcal{I}_B$ and $\mathcal{I}_N$ be the set of indices of the basic and non-basic variables, respectively. Then componentwise reformulation of (24.44) is

$$\forall\, j \in \mathcal{I}_N :\; c_j - \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{a}_j \le 0\,.$$

The meaning of the dual feasibility is this. The current value of the objective function given in (24.43) is an upper bound of the optimal value as all coefficients in the equivalent form of the objective function is nonpositive. Thus if any feasible, i.e. nonnegative, solution is substituted in it then value can be at most the constant term, i.e. the current value.

Definition 24.8 *A basic solution is* ***OPTIMAL*** *if it is both primal and dual feasible.*

It is known from the theory of linear programming that among the optimal solutions there is always at least one basic solution. To prove this statement is beyond the scope of the chapter.

In Problem (24.37)

$$\mathbf{A} = \begin{pmatrix} 3 & -5 & 1 & 0 \\ 3 & 5 & 0 & 1 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 0 \\ 15 \end{pmatrix} \quad \mathbf{c} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \end{pmatrix} .$$

If the basic variables are x_1 and x_2 then

$$\mathbf{B} = \begin{pmatrix} 3 & -5 \\ 3 & 5 \end{pmatrix} \quad \mathbf{B}^{-1} = \frac{1}{30} \begin{pmatrix} 5 & 5 \\ -3 & 3 \end{pmatrix} \quad \mathbf{N} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{c}_B = \begin{pmatrix} 2 \\ 1 \end{pmatrix} .$$

Hence

$$\mathbf{c}_B^T \mathbf{B}^{-1} = (2,1) \frac{1}{30} \begin{pmatrix} 5 & 5 \\ -3 & 3 \end{pmatrix} = \left(\frac{7}{30}, \frac{13}{30} \right)$$

$$\mathbf{B}^{-1}\mathbf{b} = \frac{1}{30} \begin{pmatrix} 5 & 5 \\ -3 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 15 \end{pmatrix} = \begin{pmatrix} 75/30 \\ 45/30 \end{pmatrix} = \begin{pmatrix} 5/2 \\ 3/2 \end{pmatrix}, \quad \mathbf{B}^{-1}\mathbf{N} = \mathbf{B}^{-1} .$$

The last equation is true as $\mathbf{N}$ is the unit matrix. Finally

$$\mathbf{c}_N^T - \mathbf{c}_B^T \mathbf{B}^{-1}\mathbf{N} = (0,\, 0) - \left(\frac{7}{30}, \frac{13}{30} \right) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \left(-\frac{7}{30}, -\frac{13}{30} \right) .$$

One can conclude that this basic solution is both primal and dual feasible.

There are two types of simplex methods. Primal simplex method starts from a primal feasible basic solution. Executing pivoting steps it goes through primal feasible basic solutions and finally even the dual feasibility achieved. The objective function values are monotone increasing in the primal simplex method.

The dual simplex method starts from a dual feasible basic solution it goes through dual feasible basic solutions until even primal feasibility is achieved in the last iteration. The objective function values are monotone decreasing in the dual simplex method. We discuss it in details as it is the main algorithmic tool of the method.

Each simplex method uses its own simplex tableau. Each tableau contains the transformed equivalent forms of the equations and the objective function. In the case of the dual simplex tableau the elements of it are derived from the form of the equations

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N = \mathbf{B}^{-1}\mathbf{b} + \mathbf{B}^{-1}\mathbf{N}(-\mathbf{x}_N) ,$$

where the second equation indicates that the minus sign is associated to non-basic variables. The dual simplex tableau contains the expression of *all* variables by the negative non-basic variables including the objective function variable x_0 and the non-basic variables. For the latter ones the trivial

$$x_j = -(-x_j)$$

equation is included. For example the dual simplex tableau for (24.37) is provided that the basic variables are x_1 and x_2 (see (24.38))

variable	constant	$-x_3$	$-x_4$
x_0	$13/2$	$7/30$	$13/30$
x_1	$5/2$	$1/6$	$1/6$
x_2	$3/2$	$-1/10$	$1/10$
x_3	0	-1	0
x_4	0	0	-1

Generally speaking the potentially non-zero coefficients of the objective function are in the first row, the constant terms are in the first column and all other coefficients are in the inside of the tableau. The order of the rows is never changed. On the other hand a variable which left the basis immediately has a column instead of that variable which entered the basis.

The elements of the dual simplex tableau are denoted by d_{jk} where $k = 0$ refers to the constant term of the equation of variable x_j and otherwise $k \in I_N$ and d_{jk} is the coefficient of the non-basic variable $-x_k$ in the expression of the variable x_j. As x_0 is the objective function variable d_{0k} is the coefficient of $-x_k$ in the equivalent form (24.42) of the objective function. The dual simplex tableau can be seen on Figure 24.5.

Notice that dual feasibility means that there are nonnegative elements in the first row of the tableau with the potential exception of its first element, i.e. with the potential exception of the objective function value.

Without giving the proof of its correctness the pivoting procedure is this. The aim of the pivoting is to eliminate the primal infeasibility, i.e. the negative values of the variables, with the potential exception of the objective function value, i.e. the elimination of the negative terms from the first column. Instead of that basic variable x_p a non-basic one will be expressed from the equation such that the negative constant term becomes zero and the value of the new basic variable, i.e. the value of x_k, becomes positive. It is easy to see that this requirement can be satisfied only if the new expressed variable, say x_k, has a negative coefficient in the equation, i.e. $d_{pk} < 0$. After the change of the basis the row of x_p must become a negative unit vector as x_p became a non-basic variable, thus its expression is

$$x_p = -(-x_p)\,. \tag{24.45}$$

The transformation of the tableau consists of the transformations of the columns such that the form (24.45) of the row of x_p is generated. The position of the (-1) in the row is the crossing of the row of x_p and the column belonging to x_k before pivoting.

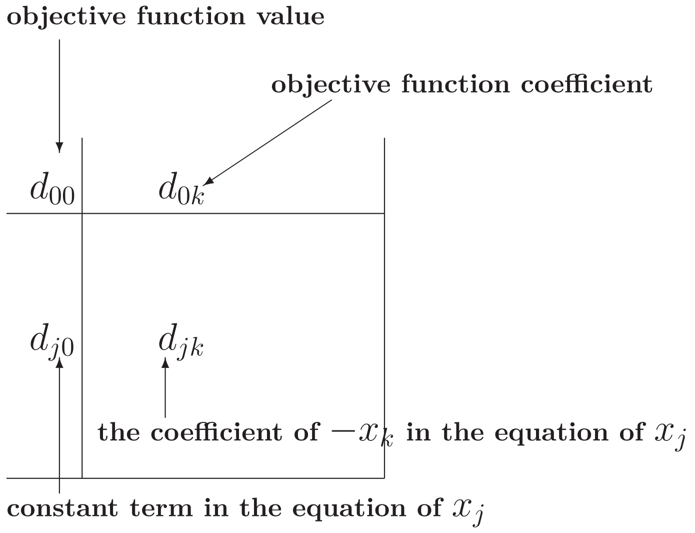

objective function value
objective function coefficient
d_{00}
d_{0k}
d_{j0}
d_{jk}
the coefficient of $-x_k$ in the equation of x_j
constant term in the equation of x_j

Figure 24.5 The elements of the dual simplex tableau.

This column becomes the column of x_p. There is another requirement claiming that the dual feasibility must hold on. Let $\mathbf{d}_j$ be the column of the non-basic variable x_j including $\mathbf{d}_0$ as the column of the constant terms. Then the formulae of the column transformation are the followings where j is either zero or the index of a non-basic variable different from k:

$$\mathbf{d}_j^{new} = \mathbf{d}_j^{old} - \frac{d_{pj}^{old}}{d_{pk}^{old}} \mathbf{d}_k^{old} \tag{24.46}$$

and

$$\mathbf{d}_p^{new} = -\frac{1}{d_{pk}^{old}} \mathbf{d}_k^{old} \, .$$

To maintain dual feasibility means that after the change of the basis the relation $d_{0j}^{new} \geq 0$ must hold for all non-basic indices, i.e. for all $j \in \mathcal{I}_N^{new}$. It follows from (24.46) that k must be chosen such that

$$k = \operatorname{argmax} \left\{ \frac{d_{0j}^{old}}{d_{pj}^{old}} \mid d_{pj}^{old} < 0 \right\} . \tag{24.47}$$

In the course of the branch method in the optimization of the relaxed subproblems dual simplex method can save a lot of computation. On the other hand what is used in the description of the method, is only the effect of one pivoting on the value of the objective function. According to (24.46) the new value is

$$d_{00}^{new} = d_{00}^{old} - \frac{d_{p0}^{old}}{d_{pk}^{old}} d_{0k}^{old} .$$

Notice that d_{p0}^{old}, $d_{pk}^{old} < 0$ and $d_{0k}^{old} \geq 0$. Hence the objective function value decreases by the nonnegative value

$$\frac{d_{p0}^{old}}{d_{pk}^{old}} d_{0k}^{old} . \tag{24.48}$$

The formula (24.48) will be used if a new constraint is introduced at branching and it cuts the previous optimal solution. As the new constraint has nothing to do with the objective function, it will not destroy dual feasibility, but, of course, the optimal solution of the relaxed problem of the branch becomes primal infeasible.

For example the inequality $x_1 \leq 2$ is added to the relaxation (24.37) defining a new branch then it is used in the equation form

$$x_1 + x_5 = 2 , \tag{24.49}$$

where x_5 is nonnegative continuous variable. According to the simplex tableau

$$x_1 = \frac{5}{2} + \frac{1}{6}(-x_3) + \frac{1}{6}(-x_4).$$

Hence

$$x_5 = -\frac{1}{2} - \frac{1}{6}(-x_3) - \frac{1}{6}(-x_4) . \tag{24.50}$$

(24.49) is added to the problem in the form (24.50). Then the dual simplex tableau is

variable	constant	$-x_3$	$-x_4$
x_0	13/2	7/30	13/30
x_1	5/2	1/6	1/6
x_2	3/2	−1/10	1/10
x_3	0	−1	0
x_4	0	0	−1
x_5	−1/2	−1/6	−1/6

Only x_5 has a negative value, thus the first pivoting must be done in its row. Rule (24.47) selects x_3 for entering the basis. Then after the first pivot step the value of the objective function decreases by

$$\frac{-\frac{1}{2}}{-\frac{1}{6}} \times \frac{7}{30} = \frac{7}{10}.$$

If the optimal solution of the relaxed problem is not reached after the first pivoting

variable	constant	$-x_3$	$-x_4$
x_0	$13/2$	$7/30$	$13/30$
x_1	$5/2$	$1/6$	$1/6$
x_2	$3/2$	$-1/10$	$1/10$
x_3	0	-1	0
x_4	0	0	-1
x_6	$-1/2$	$1/6$	$1/6$

then further decrease is possible. But decrease of 0.7 is sure compared to the previous upper bound.

Another important property of the cuts is that if it has no negative coefficient in the form how it is added to the simplex tableau then there is no negative pivot element, i.e. the relaxed problem is infeasible, i.e. the branch is empty. For example the cut $x_1 \geq 3$ leading to an empty branch is added to the problem in the form

$$x_1 - x_6 = 3$$

where x_6 is also a nonnegative variable. Substituting the value of x_1 again the equation is transformed to

$$x_6 = -\frac{1}{2} + \frac{1}{6}(-x_3) + \frac{1}{6}(-x_4)\,.$$

Hence the simplex tableau is obtained. There is a negative value at the crossing point of the first column and the row of x_6. But it is not possible to choose a pivot element in that row, as there is no negative element of the row. It means that feasibility can not be achieved, i.e. that branch is infeasible and thus it is completely fathomed.

24.3.3. Fast bounds on lower and upper branches

The branching is always based on a variable which should be integer but in the current optimal solution of the linear programming relaxation it has fractional value. If it has fractional value then its value is non-zero thus it is basic variable. Assume that its index is p. Remember that $\mathcal{I}$, $\mathcal{I}_B$, and $\mathcal{I}_N$ are the index sets of the integer, basic, and non-basic variables, respectively. Hence $p \in \mathcal{I} \cap \mathcal{I}_B$. According to the last simplex tableau x_p is expressed by the non-basic variables as follows:

$$x_p = d_{p0} + \sum_{j \in \mathcal{I}_N} d_{pj}(-x_j)\,. \tag{24.51}$$

As d_{p0} has fractional value

$$1 > f_p = d_{p0} - \lfloor d_{p0} \rfloor > 0\,.$$

The branch created by the inequality

$$x_p \leq \lfloor d_{p0} \rfloor \tag{24.52}$$

is called *lower branch* and the inequality

$$x_p \ \geq \ \lfloor d_{p0} \rfloor + 1$$

creates the *upper branch*.

Let $\mathcal{J}^+$ and $\mathcal{J}^-$ be the set of indices of the nonbasic variables according to the signs of the coefficients in (24.51), i.e.

$$\mathcal{J}^+(\mathcal{J}^-) \ = \ \{j \mid j \in \mathcal{I}_N;\ d_{pj} > 0\,(d_{pj} < 0)\}\,.$$

First the lower branch is analyzed. It follows from (24.51) that the inequality (24.52) is equivalent to

$$x_p \ - \ \lfloor d_{p0} \rfloor \ = \ f_p \ + \ \sum_{j \in \mathcal{I}_N} d_{pj}(-x_j) \ \leq \ 0.$$

Thus

$$s \ = \ -f_p \ + \ \sum_{j \in \mathcal{I}_N} (-d_{pj})(-x_j) \tag{24.53}$$

is a nonnegative variable and row (24.53) can be added to the dual simplex tableau. It will contain the only negative element in the first column that is the optimization in the lower branch starts by pivoting in this row. (24.53) can be reformulated according to the signs of the coefficients as

$$s \ = \ -f_p \ + \ \sum_{j \in \mathcal{J}^+} (-d_{pj})(-x_j) \ + \ \sum_{j \in \mathcal{J}^-} (-d_{pj})(-x_j)\,. \tag{24.54}$$

The pivot element must be negative, thus it is one of $-d_{pj}$'s with $j \in \mathcal{J}^+$. Hence the first decrease (24.48) of the objective function is

$$P_{lp} \ = \ \min\left\{ \frac{d_{0j}}{d_{pj}} f_p \mid j \in \mathcal{J}^+ \right\}\,. \tag{24.55}$$

In the upper branch the inequality (24.52) is equivalent to

$$x_p \ - \ \lfloor d_{p0} \rfloor \ = \ f_p \ + \ \sum_{j \in \mathcal{I}_N} d_{pj}(-x_j) \ \geq 1\,.$$

Again the nonnegative slack variable s should be introduced. Then the row which can be added to the simplex tableau is

$$s \ = \ (f_p - 1) \ + \ \sum_{j \in \mathcal{J}^+} d_{pj}(-x_j) \ + \ \sum_{j \in \mathcal{J}^-} d_{pj}(-x_j)\,. \tag{24.56}$$

Thus the pivot element must belong to one of the indices $j \in \mathcal{J}^-$ giving the value

$$P_{up} \ = \ \min\left\{ \frac{d_{0j}}{-d_{pj}} (1 - f_p) \mid j \in \mathcal{J}^- \right\}\,. \tag{24.57}$$

Let $\hat{z}$ be the upper bound on the original branch obtained by linear programming. Then the quantities P_{lp} and P_{up} define the upper bounds of the objective functions $\hat{z} - P_{lp}$ and $\hat{z} - P_{up}$ on the lower and upper subbranches, respectively. They are not substituting complete optimization in the subbranches. On the other hand they are easily computable and can give some orientation to the selection of the next branch for further investigation (see below).

The quantities P_{lp} and P_{up} can be improved, i.e. increased. The claim that the variable s defined in (24.54) is nonnegative implies that

$$-f_p \geq \sum_{j \in \mathcal{J}^+} d_{pj}(-x_j)\,. \tag{24.58}$$

In a similar way the nonnegativity of variable s in (24.56) implies that

$$f_p - 1 \geq \sum_{j \in \mathcal{J}^-} (-d_{pj})(-x_j)\,. \tag{24.59}$$

If (24.59) is multiplied by the positive number

$$\frac{f_p}{1-f_p}$$

then it gets the form

$$-f_p \geq \sum_{j \in \mathcal{J}^-} \frac{f_p}{1-f_p}(-d_{pj})(-x_j)\,. \tag{24.60}$$

The inequalities (24.58) and (24.60) can be unified in the form:

$$-f_p \geq \sum_{j \in \mathcal{J}^+} d_{pj}(-x_j) + \sum_{j \in \mathcal{J}^-} \frac{f_p}{1-f_p}(-d_{pj})(-x_j)\,. \tag{24.61}$$

Notice that (24.61) *not* the sum of the two inequalities. The same negative number stands on the left-hand side of both inequalities and is greater or equal than the right-hand side. Then both right-hand sides must have negative value. Hence the left-hand side is greater than their sum.

The same technique is applied to the variable $x_p^{'}$ instead of x_p with

$$x_p^{'} = x_p + \sum_{j \in \mathcal{I} \cap \mathcal{I}_N} \mu_j x_j\,,$$

where μ_j's are integer values to be determined later. $x_p^{'}$ can be expressed by the non-basic variables as

$$x_p^{'} = d_{p0} + \sum_{j \in \mathcal{I} \cap \mathcal{I}_N} (d_{pj} - \mu_j)(-x_j) + \sum_{j \in \mathcal{I}_N \setminus \mathcal{I}} d_{pj}(-x_j)\,.$$

Obviously $x_p^{'}$ is an integer variable as well and its current value if the non-basic variables are fixed to zero is equal to the current value of d_{p0}. Thus it is possible to define the new branches according to the values of $x_p^{'}$. Then the inequality of type (24.61) which is defined by $x_p^{'}$, has the form

$$-f_p \geq \sum_{\substack{j \in \mathcal{I} \cap \mathcal{I}_N \\ d_{pj} - \mu_j \geq 0}} (d_{pj} - \mu_j)(-x_j) + \sum_{\substack{j \in \mathcal{I} \cap \mathcal{I}_N \\ d_{pj} - \mu_j < 0}} \frac{f_p}{1-f_p}(\mu_j - d_{pj})(-x_j)$$

$$+ \sum_{\substack{j \in \mathcal{I}_N \setminus \mathcal{I} \\ d_{pj} > 0}} d_{pj}(-x_j) + \sum_{\substack{j \in \mathcal{I}_N \setminus \mathcal{I} \\ d_{pj} < 0}} \frac{f_p}{1-f_p}(-d_{pj})(-x_j)\,.$$

The appropriate quantities $P_{lp}^{'}$ and $P_{up}^{'}$ are as follows:

$$P_{lp}^{'} = \min\{a, b\}\,,$$

where

$$a = \min\left\{\frac{d_{0j}}{d_{pj} - \mu_j} f_p \mid j \in \mathcal{I} \cap \mathcal{I}_N, d_{pj} - \mu_j > 0\right\}$$

and

$$b = \min\left\{\frac{d_{0j}}{d_{pj}} f_p \mid j \in \mathcal{I}_N \setminus \mathcal{I},\, d_{pj} > 0\right\}$$

further

$$P_{up}^{'} = \min\{c, d\}\,,$$

where

$$c = \min\left\{\frac{d_{0j}(1-f_p)^2}{(\mu_j - d_{pj})f_p} \mid j \in \mathcal{I} \cap \mathcal{I}_N, d_{pj} - \mu_j < 0\right\}$$

and

$$d = \min\left\{-\frac{d_{0j}(1-f_p)^2}{f_p d_{pj}} \mid j \in \mathcal{I}_N \setminus \mathcal{I},\, d_{pj} < 0\right\}\,.$$

The values of the integers must be selected in a way that the absolute values of the coefficients are as small as possible, because the inequality cuts the greatest possible part of the polyhedral set of the continuous relaxation in this way. (See Exercise 24.3-1.) To do so the absolute value of $d_{pj} - \mu_j$ must be small. Depending on its sign it can be either f_j, or $f_j - 1$, where f_j is the fractional part of d_{pj}, i.e. $f_j = d_{pj} - \lfloor d_{pj} \rfloor$.

Assume that $f_j > 0$. If $d_{pj} + \mu_j = f_j$ then the term

$$\frac{d_{0j} f_p}{f_j} \tag{24.62}$$

is present among the terms of the minimum of the lower branch. If $d_{pj} > 0$ then it obviously is at least as great as the term

$$\frac{d_{0j} f_p}{d_{pj}}\,,$$

which appears in P_{lp}, i.e. in the right-hand side of (24.55). If $d_{pj} < 0$ then there is a term

$$\frac{d_{0j}(f_p - 1)}{d_{pj}} \tag{24.63}$$

is in the right-hand side of (24.57) . d_{oj} is a common multiplier in the terms (24.62) and (24.63), therefore it can be disregarded when the terms are compared. Under the assumption that $f_j \le f_p$ it will be shown that

$$\frac{f_p}{f_j} \ge \frac{f_p - 1}{d_{pj}} .$$

As d_{pj} is supposed to be negative the statement is equivalent to

$$d_{pj} f_p \le (f_p - 1) f_j .$$

Hence the inequality

$$(\lfloor d_{pj} \rfloor + f_j)\, f_p \le f_p f_j - f_j$$

must hold. It can be reduced to

$$\lfloor d_{pj} \rfloor\, f_p \le -f_j .$$

It is true as $\lfloor d_{pj} \rfloor \le -1$ and

$$-1 \le \frac{-f_j}{f_p} < 0 .$$

If $d_{pj} + \mu_j = f_j - 1$ then according to (24.57) and (24.61) the term

$$\frac{d_{0j}(1 - f_j)^2}{f_p(1 - f_j)}$$

is present among the terms of the minimum of the upper branch. In a similar way it can be shown that if $f_j > f_p$ then it is always at least as great as the term

$$\frac{d_{0j}(f_j - 1)}{d_{pj}}$$

which is present in the original formula (24.57).

Thus the rule of the choice of the integers μ_j's is

$$\mu_j = \begin{cases} \lfloor d_{pj} \rfloor & \text{if } f_j \le f_p , \\ \lceil d_{pj} \rceil & \text{if } f_j > f_p \end{cases} \tag{24.64}$$

24.3.4. Branching strategies

The B&B frame doesn't have any restriction in the selection of the unfathomed node for the next branching in row 7 of Branch-and-Bound. First two extreme strategies are discussed with pros and cons. The same considerations have to be taken in almost all applications of B&B. The third method is a compromise between the two extreme ones. Finally methods more specific to the integer programming are discussed.

The LIFO Rule LIFO means “Last-In-First-Out”, i.e. one of the branches generated in the last iteration is selected. A general advantage of the rule is that the size of the enumeration tree and the size of the list $\mathcal{L}$ remains as small as possible. In the case of the integer programming problem the creation of the branches is based on the integer values of the variables. Thus the number of the branches is at most g_j+1 if the branching variable is x_j. In the LIFO strategy the number of leaves is strictly less then the number of the created branches on each level with the exemption of the deepest level. Hence at any moment the enumeration tree may not have more than

$$\sum_{j=1}^{n} g_j + 1$$

leaves.

The drawback of the strategy is that the flexibility of the enumeration is lost. The flexibility is one of the the main advantage of B&B in solving pure integer problems.

If the algorithm skips from one branch to another branch far away from the first one then it is necessary to reconstruct the second branch including not only the branching restrictions on the variables but any other information which is necessary to the bounding procedure. In the particular algorithm the procedure determining the bound is linear programming, more precisely a simplex method. If a new restriction as a linear constraint is added to the problem which cuts off the previous optimal solution, then the simplex tableau looses the primal feasibility but the dual feasibility still holds. Thus a dual simplex method can immediately start without carrying out a first phase. (The purpose of the first phase which itself is a complete optimization, is to find a primal or dual feasible basic solution depending for primal or dual simplex method, respectively.) If the B&B method skips to another branch then to get the new bound by the simplex method will require the execution of the first phase, too.

A further consideration concerns to the construction of feasible solutions. Generally speaking if good feasible solutions are known in the early phase of the algorithm then the whole procedure can be accelerated. In the current algorithm branching has a "constructive nature". It means that the value of the branching variable becomes more restricted therefore it either becomes integer in the further optimal solutions in the subbranches of the branch, or it will be restricted further on. Thus it can be expected that sooner or later a complete integer solution is constructed which might be feasible or infeasible. On the other hand if the algorithm skips frequently in the phase when no feasible solution is known then it is very likely that any construction will be finished only later, i.e. the acceleration will not take place, because of the lack of feasible solution.

If a LIFO type step is to be done and the branching variable is x_p then the lower branch should be chosen in step 7 of the algorithm, if

$$z_r - P_{lp} \geq z_r - P_{up}, \quad \text{i.e.} \quad P_{lp} \leq P_{up} \,.$$

The maximal bound The other extreme strategy is that the branch having the maximal bound is selected in each iteration. The idea is simple and clear: it is the most promising branch therefore it worth to explore it.

Unfortunately the idea is not completely true. The bounds of the higher level branches are not accurate enough. This phenomenon has been discussed during the analysis of the numerical example in the subsection 24.1.3 in relation (24.12). Thus a somewhat smaller upper bound in a lower branch can indicate a more promising branch.

The maximal bound strategy can lead to a very wide enumeration tree which may cause memory problems. Moreover the construction of feasible solutions will be slow and therefore the relatively few solutions will be enumerated implicitly, i.e. the number of steps will be high, i.e. the method will be slow.

Fast bounds and estimates If the optimal solution of the relaxed problem is non-integer then it can have several fractional components. All of them must be changed to be integer to obtain the optimal integer programming solution of the branch. The change of the value of each currently fractional variable as a certain cost. The cost of the individual changes are estimated and summed up. The cost means the loss in the value of the objective function. An adjusted value of the bound of the branch is obtained if the sum of the estimated individual costs is subtracted from the current bound. It is important to emphasize that the adjusted value is not an upper or lower bound of the optimal value of integer programming solution of the branch but it is only a realistic estimation.

There are two ways to obtain the estimation. The first one uses the crude values of the fractionality. Let f_j and f_j^0 be the fractional part of variable x_j in the current branch and in the relaxed problem of the original problem, respectively. Further on let z_r, z_0, and $\hat{z}$ be the optimal value of the relaxed problem in the current branch, in the original problem, and the value of the best feasible integer solution found so far. Generally speaking the measure of the fractionality of a real number α is that how far is α to the closest integer, i.e.

$$\min\{\alpha - \lfloor \alpha \rfloor, \lceil \alpha \rceil - \alpha\}\,.$$

Hence the estimate is

$$z_r - (z_0 - \hat{z}) \frac{\sum_{j \in \mathcal{I}} \min\{f_j, 1 - f_j\}}{\sum_{j \in \mathcal{I}} \min\{f_j^0, 1 - f_j^0\}}\,. \tag{24.65}$$

(24.65) takes into account the average inaccuracy of the bounds.

The fast bounds defined in (24.55) and (24.57) can be used also for the same purpose. They concern to the correction of the fractionality of a single variable in the current branch. Hence the estimate

$$z_r - \sum_{j \in \mathcal{I}} \min\{P_{lj}, P_{uj}\}$$

is a natural choice.

A Rule based on depth, bound, and estimates The constraints defining the branches are integer valued lower and upper bounds on the branching variables. Thus one can expect that these new constraints force the majority of the branching variables to be integer. It means that the integrality of the optimal solution of the relaxed problem improves with the depth of the branch. Thus it is possible to connect the last two rules on the following way. The current bound is abandoned and the algorithm selects the best bound is the improvement based on estimates is above a certain threshold.

24.3.5. The selection of the branching variable

In selecting the branching variable again both the fractional part of the non-integer variables and the fast bounds have critical role. A further factor can be the information obtained from the user.

Selection based on the fractional part The most significant change can be expected from that variable which is farthest from integers as the cuts defining the two new branches cut the most. As the measure of fractionality is $\min\{f_j, 1 - f_j\}$ the rule suggest to choose the branching variable x_p as

$$p = \text{argmax}\{\min\{f_j, 1 - f_j\} \mid j \in \mathcal{I}\}$$

Selection based on fast bounds Upper bounds are

$$z_r - P_{lp} \quad \text{and} \quad z_r - P_{up}$$

in the lower and upper branches of branch r if the branching variable is x_p.

Here are five possible selection criteria:

$$\max_{p:} \max\{z_r - P_{lp},\, z_r - P_{up}\} \tag{24.66}$$

$$\max_{p:} \min\{z_r - P_{lp},\, z_r - P_{up}\} \tag{24.67}$$

$$\min_{p:} \max\{z_r - P_{lp},\, z_r - P_{up}\} \tag{24.68}$$

$$\min_{p:} \min\{z_r - P_{lp},\, z_r - P_{up}\} \tag{24.69}$$

$$\max_{p:}\{\mid P_{lp} - P_{up} \mid\}\,. \tag{24.70}$$

Which one can be offered for a B&B algorithm?

Notice that

$$\max\{z_r - P_{lp},\, z_r - P_{up}\}$$

is a correct upper bound of branch r as it has been mentioned earlier. Thus (24.66) selects according to the most inaccurate upper bound. It is obviously not good. (24.68) makes just the opposite it selects the variable giving the most accurate bound. On the other hand

$$\min\{z_r - P_{lp},\, z_r - P_{up}\} \tag{24.71}$$

is the upper bound in the worse one of the two subbranches. The interest of the algorithm is that it will be fathomed without explicit investigation, i.e. the bound of this subbranch will be less than the objective function value of an integer feasible solution. Thus it is good if (24.71) is as small as possible. Hence (24.69) is a good strategy and (24.67) is not. Finally, (24.70) tries to separate the good and low quality feasible solutions. The conclusion is that (24.69) and (24.70) are the two best ones and (24.68) is still applicable, but (24.66) and (24.67) must be avoided.

Priority rule Assume that the numerical problem (24.31)-(24.35) is the model of an industrial problem. Then the final user is the manager and/or expert who must apply the decisions coded into the optimal solution. The expert may know that which factors (decisions) are the most critical ones from the point of view of the managerial problem and the industrial system. The variables belonging to these factors may have a special importance. Therefore it has sense if the user may define a priority order of variables. Then the first non-integer variable of the order can be selected as branching variable.

24.3.6. The numerical example is revisited

The solution of the problem

$$\begin{array}{llrcrcr} \max & x_0 = & 2x_1 & + & x_2 & & \\ & & 3x_1 & - & 5x_2 & \leq & 0 \\ & & 3x_1 & + & 5x_2 & \leq & 15 \\ & & \multicolumn{5}{c}{x_1,\, x_2 \geq 0} \\ & & \multicolumn{5}{c}{x_1,\, x_2 \text{ is integer}\,.} \end{array} \tag{24.36}$$

has been analyzed from geometric point of view in subsection 24.3.1. Now the above-mentioned methods will be applied and the same course of solution will be obtained.

After introducing the slack variables x_3 and x_4 the (primal) simplex method gives the equivalent form (24.38) of the equations and the objective function:

$$\begin{array}{llrcrcrcrcr} \max & x_0 = & 0x_1 & + & 0x_2 & - & \frac{7}{30}x_3 & - & \frac{13}{30}x_4 & + & \frac{13}{2} \\ & & x_1 & + & 0x_2 & + & \frac{1}{6}x_3 & + & \frac{1}{6}x_4 & = & \frac{5}{2} \\ & & 0x_1 & + & x_2 & - & \frac{1}{10}x_3 & + & \frac{1}{10}x_4 & = & \frac{3}{2} \\ & & \multicolumn{9}{l}{x_1,\, x_2\, x_3,\, x_4 \geq 0\,.} \end{array} \tag{24.38}$$

Hence it is clear that the solution $x_1 = \frac{5}{2}$ and $x_2 = \frac{3}{2}$. (24.38) gives the following optimal *dual* simplex tableaux:

$$\begin{array}{cccc} & & -x_3 & -x_4 \\ x_0 & 13/2 & 7/30 & 13/30 \\ x_1 & 5/2 & 1/6 & 1/6 \\ x_2 & 3/2 & -1/10 & 1/10 \\ x_3 & 0 & -1 & 0 \\ x_4 & 0 & 0 & -1 \end{array}\,.$$

The first two branches were defined by the inequalities $x_1 \leq 2$ and $x_1 \geq 3$. The second one is an empty branch. The algebraic evidence of this fact is that there is no negative element in the row of x_1, thus it is not possible to find a pivot element for the dual simplex method after introducing the cut. Now it will be shown in a detailed way. Let s be the appropriate slack variable, i.e. the cut introduced in the form

$$x_1 - s = 3, \quad s \geq 0 .$$

The new variable s must be expressed by the non-basic variables, i.e. by x_3 and x_4:

$$3 = x_1 - s = \frac{5}{2} - \frac{1}{6}x_3 - \frac{1}{6}x_4 - s .$$

Hence

$$s = -\frac{1}{2} + \frac{1}{6}(-x_3) + \frac{1}{6}(-x_4) .$$

When this row is added to the dual simplex tableaux, it is the only row having a negative constant term, but there is no negative coefficient of any non-basic variable proving that the problem is infeasible. Notice that the sign of a coefficient is an immediate consequence of the sign of the coefficient in the row of x_1, i.e. it is not necessary to carry out the calculation of the row of s and it is possible to conclude immediately that the branch is empty.

The fractional part f_1 equals $\frac{1}{2}$. Hence the fast bound (24.55) of the lower branch defined by $x_1 \leq 2$ is

$$\frac{1}{2} \min \left\{ \frac{\frac{7}{30}}{\frac{1}{6}}, \frac{\frac{13}{30}}{\frac{1}{6}} \right\} = \frac{7}{10} .$$

It means that the fast upper bound in the branch is 13/2-7/10=5.8. The bound can be rounded down to 5 as the objective function is integer valued.

Let x_5 be the slack variable of the cut $x_1 \leq 2$, i.e. $x_1 + x_5 = 2$. Hence

$$x_5 = \frac{1}{2} - \left(-\frac{1}{6}\right)(-x_3) - \left(-\frac{1}{6}\right)(-x_4) .$$

If it is added to the simplex tableaux then the pivot element is d_{53}. After the first pivot step the tableaux becomes optimal. It is

$$\begin{array}{crrr} & & -x_5 & -x_4 \\ x_0 & 29/5 & 7/5 & 1/5 \\ x_1 & 2 & 1 & 0 \\ x_2 & 9/5 & -3/5 & 1/5 \\ x_3 & 3 & -6 & 1 \\ x_4 & 0 & 0 & -1 \\ x_5 & 0 & -1 & 0 \end{array} . \qquad (24.72)$$

Notice that the optimal value is 5.8, i.e. exactly the same what was provided by the fast bound. The reason is that the fast bound gives the value of the objective function after the first pivot step. In the current case the first pivot step immediately produced the optimal solution of the relaxed problem.

x_2 is the only variable having non-integer value in simplex tableaux. Thus the branching must be done according to x_2. The two new cuts defining the branches are $x_2 \leq 1$ and $x_2 \geq 2$. There are both positive and negative coefficients in the row of x_2, thus both the lower and upper branches exist. Moreover

$$P_{l2} = \frac{4}{5} \times \frac{1/5}{1/5} = \frac{4}{5}, \quad P_{u2} = \frac{1}{5} \times \frac{7/5}{3/5} = \frac{7}{15}.$$

Thus the continuous upper bound is higher on the upper branch, therefore it is selected first for further branching.

The constraint

$$x_2 - x_6 = 2, \quad x_6 \geq 0$$

are added to the problem. By using the current simplex tableaux the equation

$$x_6 = -\frac{1}{5} - \frac{3}{5}(-x_5) + \frac{1}{5}(-x_4)$$

is obtained. It becomes the last row of the simplex tableaux. In the first pivoting step x_6 enters the basis and x_5 leaves it. The first tableaux is immediately optimal and it is

$$\begin{array}{cccc}
 & & -x_6 & -x_4 \\
x_0 & 16/3 & 7/3 & 2/3 \\
x_1 & 5/3 & 5/3 & 1/3 \\
x_2 & 2 & -1 & 0 \\
x_3 & 5 & -10 & -1 \\
x_4 & 0 & 0 & -1 \\
x_5 & 1/3 & -5/3 & -1/3 \\
x_6 & 0 & -1 & 0
\end{array}$$

Here both x_1 and x_5 are integer variables having non-integer values. Thus branching is possible according to both of them. Notice that the upper branch is empty in the case of x_1, while the lower branch of x_5 is empty as well. x_1 is selected for branching as it is the variable of the original problem. Now

$$P_{l1} = \frac{2}{3} \min \left\{ \frac{7/3}{5/3}, \frac{2/3}{1/3} \right\} = \frac{14}{15}.$$

On the other hand the bound can be improved in accordance with (24.64) as $d_{16} > 1$, i.e. the coefficient of $-x_6$ may be $2/3$ instead of $5/3$. It means that the inequality

$$x_1 + x_6 \leq 1$$

is claimed instead of

$$x_1 \leq 1.$$

It is transferred to the form

$$x_1 + x_6 + x_7 = 1.$$

Hence

$$x_7 = -\frac{2}{3} - \frac{2}{3}(-x_6) - \frac{1}{3}(-x_4)\,.$$

The improved fast bound is obtained from

$$P'_{l1} = \frac{2}{3}\min\left\{\frac{7}{2}, 2\right\} = \frac{4}{3}\,.$$

It means that the objective function can not be greater than 4. After the first pivoting the simplex tableau becomes

		$-x_6$	$-x_7$
x_0	4	1	2
x_1	1	1	1
x_2	2	−1	0
x_3	7	−8	−3
x_4	2	2	−3
x_5	1	−1	−1
x_6	0	−1	0
x_7	0	0	−1

giving the feasible solution $x_1 = 1$ and $x_2 = 2$ with objective function value 4.

There is only one unfathomed branch which is to be generated from tableaux (24.72) by the constraint $x_2 \leq 1$. Let x_8 be the slack variable. Then the equation

$$1 = x_2 + x_8 = \frac{9}{5} - \frac{3}{5}(-x_5) + \frac{1}{5}(-x_4) + x_8$$

gives the cut

$$x_8 = -\frac{4}{5} + \frac{3}{5}(-x_5) - \frac{1}{5}(-x_4)$$

to be added to the tableaux. After two pivoting steps the optimal solution is

		$-x_3$	$-x_6$
x_0	13/3	2/3	13/3
x_1	5/3	1/3	5/3
x_2	1	0	1
x_3	5	−1	0
x_4	5	−1	−10
x_5	1/3	−1/3	−5/3
x_6	0	0	−1

Although the optimal solution is not integer, the branch is fathomed as the upper bound is under 5, i.e. the branch can not contain a feasible solution better than the current best known integer solution. Thus the method is finished.

Exercises

24.3-1 Show that the rule of the choice of the integers μ_j (24.64) is not necessarily optimal from the point of view of the object function. (*Hint.* Assume that variable x_j enters into the basis in the first pivoting. Compare the changes in the objective function value if its coefficient is $-f_j$ and $f_j - 1$, respectively.)

24.4. On the enumeration tree

One critical point of B&B is the storing of the enumeration tree. When a branch is fathomed then even some of its ancestors can become completely fathomed provided that the current branch was the last unfathomed subbranch of the ancestors. The ancestors are stored also otherwise it is not possible to restore the successor. As B&B uses the enumeration tree on a flexible way, it can be necessary to store a large amount of information on branches. It can causes memory problems. On the other hand it would be too expensive from the point of view of calculations to check the ancestors every time if a branch becomes fathomed. This section gives some ideas how to make a trade-off.

The first thing is to decide is that which data are describing a branch. There are two options. The first one is that all necessary informations are stored for each branch. It includes all the branching defining constraints. In that case the same constraint is stored many times, because a branch on a higher level may have many subbranches. As matter of fact the number of branches is very high in the case of large scale problems, thus the memory required by this solution is very high.

The other option is that only those informations are stored which are necessary to the complete reconstruction of the branch. These ones are

- the parent branch, i.e. the branch from which it was generated directly,
- the bound of the objective function on the branch,
- the index of the branching variable,
- the branch defining constraint of the branching variable.

For technical reasons three other attributes are used as well:

- a Boolean variable showing if the branch has been decomposed into subbranches,
- another Boolean variable showing if any unfathomed subbranch of the branch exists,
- and a pointer to the next element in the list of branches.

Thus a branch can be described by a **record** as follows:

```
record Branch
begin
    Parent          : Branch;
    Bound           : integer;
    Variable        : integer;
    Value           : integer;
    Decomposition   : Boolean;
    Descendant      : Boolean;
    suc             : Branch
end;
```

The value of the Parent attribute is **none** if and only if the branch is the initial branch, i.e. the complete problem. It is the root of the B&B tree. The reconstruction of the constraints defining the particular branch is the simplest if it is supposed that the branches are defined by the fixing of a free variable. Assume that Node is a variable of type Branch. At the beginning its value is the branch to be reconstructed. Then the algorithm of the reconstruction is as follows.

Branch-Reconstruction

```
1  while Node ≠ none
2        do $x$[Node.Variable] ← Node.Value;
3           . . .
4           Node ← Node.Parent;
5  return Node
```

The value of a previously fixed variable is set to the appropriate value in row 2. Further operations are possible (row 4). Node becomes its own parent branch in row 5. If it is **none** then the root is passed and all fixings are done.

Sometimes it is necessary to execute some operations on all elements of the list $\mathcal{L}$. The suc attribute of the branches point to the next element of the list. The last element has no next element, therefore the value of suc is **none** in this case. The procedure of changing all elements is somewhat similar to the Branch Reconstruction procedure. The head of the list $\mathcal{L}$ is Tree, i.e. the first element of the list is Tree.suc.

B&B-List

```
1  Node ← Tree.suc
2  while Node≠none
3        . . .
4        Node ← Node.suc
5  return Node
```

The loop runs until there is no next element. The necessary operations are executed in row 4. The variable Node becomes the next element of the list in row 5. To insert a new branch into the list is easy. Assume that it is NewNode of type Branch and it is to be inserted after Node which is in the list. Then the necessary two commands are:

```
NewNode.suc ← Node.suc
      Node.suc ← NewNode
```

If the branches are not stored as objects but they are described in long arrays then the use of attribute suc is superflous and instead of the procedure B&B List a **for** loop can be applied.

The greatest technical problem of B&B from computer science point of view is memory management. Because branches are created in enormous large quantity the fathomed branches must be deleted from the list time to time and the memory occupied by them must be freed. It is a kind of garbage collection. It can be done in three main steps. In the first one value **false** is assigned to the attribute Descendant of all elements of the list. In the second main step an attribute Descendant is changed to **true** if and only if the branch has unfathomed descendant(s). In the third step the unnecessary branches are deleted. It is assumed that there is a procedure Out which gets the branch to be deleted as a parameter and deletes it and frees the part of the memory.

Garbage-Collection

```
Node ← Tree.suc
while Node ≠ none
        Node.Descendant ← False
        Node ← Node.suc
Node ← Tree.suc
while Node ≠ none
        do if not Node.Decomposition  and Node.Bound > $\hat{z}$
         then Pont ← Node.Parent
         while Pont ≠ none do
             Pont.Descendant ← True
             Pont ← Pont.Parent
        Node ← Node.suc
Node ← Tree.suc
 while Node ≠ none do
        Pont ← Node.suc
        if (not Node.Descendant  and Node.Decomposition)  or Node.Bound ≤ $\hat{z}$
        then Out(Node)
        Node ← Pont
return ???
```

24.5. The use of information obtained from other sources

The method can be sped up by using information provided by further algorithmic tools.

24.5.1. Application of heuristic methods

The aim of the application of heuristics methods is to obtain feasible solutions. From theoretical point of view to decide if any feasible solution exists is NP-complete as well. On the other hand heuristics can produce feasible solutions in the case of the majority of the numerical problems. The methods to be applied depend on the nature of the problem in question, i.e. pure binary, bounded integer, mixed integer problems may require different methods. For example for pure integer problems local search and Lagrange multipliers can work well. Lagrange multipliers also provide upper bound (in the case of maximization) of the optimal value.

If a feasible solution is known then it is immediately possible to disregard branches based on their bounds. See row 15 of algorithm Branch and Bound. There the branches having not good enough bounds are automatically eliminated. In the case of pure binary problem an explicit objective function constraint can give a lot of consequences as well.

24.5.2. Preprocessing

Preprocessing means to obtain information on variables and constraints based on algebraic constraints and integrality.

For example if the two constraints of problem (24.36) are summed up then the inequality

$$6x_1 \leq 15$$

is obtained implying that $x_1 \leq 2$.

Let

$$g_i(x) \leq b_i \tag{24.73}$$

be one of the constraints of problem (24.14)-(24.16). Many tests can be based on the following two easy observations:

1. *If the maximal value of the left-hand side of (24.73) of $x \in \mathcal{X}$ is not greater than the right-hand side of (24.73) then the constraint is redundant.*

2. *If the minimal value of the left-hand side of (24.73) if $x \in \mathcal{X}$ is greater than the right-hand side of (24.73) then it is not possible to satisfy the constraint, i.e. the problem (24.14)-(24.16) has no feasible solution.*

If under some further restriction the second observation is true then the restriction in question can be excluded. A typical example is that certain variables are supposed to have maximal/minimal possible value. In this way it is possible to fix a variable or decrease its range.

Lagrange relaxation can be used to fix some variables, too. Assume that the optimal value of Problem (24.22) and (24.16) is $\nu(L(\lambda \mid x_j = \delta))$ under the further condition that x_j must take the value δ. If $\hat{z}$ is the objective function value of a known feasible solution and $\hat{z} > \nu(L(\lambda \mid x_j = \delta))$ then x_j can not take value δ. Further methods are assuming that the LP relaxation of the problem is solved and based on optimal dual prices try to fix the values of variables.

24.6. Branch and Cut

Branch and Cut (B&C) in the simplest case is a B&B method such that the a certain kind of information is collected and used during the whole course of the algorithm. The theoretical background is based on the notion of *integer hull*

Definition 24.9 *Let*

$$\mathcal{P} = \{\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}\}$$

be a polyhedral set where $\mathbf{A}$ *is an* $m \times n$ *matrix,* $\mathbf{x}$ *and* $\mathbf{b}$ *are* n *and* m *dimensional vectors. All elements of* $\mathbf{A}$ *and* $\mathbf{b}$ *are rationals. The convex hull of the integer points of* P *is called the* integer hull *of* $\mathcal{P}$*, i.e. it is the set*

$$\mathrm{conv}(\mathcal{P} \cap Z^n)\,.$$

The integer hull of the polyhedral set of problem (24.36) is the convex hull of the points (0,0), (0,3), (1,2), and (1,1) as it can be seen on Figure 24.2. Thus the description of the integer hull as a polyhedral set is the inequality system:

$$x_1 \geq 0, \quad x_1 + x_2 \leq 3, \quad x_1 \leq 1, \quad x_1 - x_2 \leq 0\,.$$

Under the conditions of the definition the integer hull is a polyhedral set, too. It is a non-trivial statement and in the case of irrational coefficients it can be not true. If the integer hull is known, i.e. a set of linear inequalities defining exactly the integer hull polyhedral set is known, then the integer programming problem can be reduced to a linear programming problem. Thus problem (24.36) is equivalent to the problem

$$\begin{array}{lllllll} \max & x_0 & = & 2x_1 & + & x_2 & & \\ & & & x_1 & & & \geq & 0 \\ & & & x_1 & + & x_2 & \leq & 3 \\ & & & x_1 & & & \leq & 1 \\ & & & x_1 & - & x_2 & \leq & 0\,. \end{array} \tag{24.74}$$

As the linear programming problem easier to solve than the integer programming problem, one may think that it worth to carry out this reduction. It is not completely true. First of all the number of the linear constraint can be extremely high. Thus generating all constraints of the integer hull can be more difficult than the solution of the original problem. Further on the constraints determining the shape of the integer hull on the side opposite to the optimal solution are not contributing to the finding of the optimal solution. For example the optimal solution of (24.74) will not change if the first constraint is deleted and it is allowed both x_1 and x_2 may take negative values.

On the other hand the first general integer programming method is the cutting plane method of Gomory. Its main tool is the cut which is based on the observation that possible to determine linear inequalities such that they cut the non-integer optimal solution of the current LP relaxation, but they do not cut any integer feasible solution. A systematic generation of cuts leads to a finite algorithm which finds an optimal solution and proves its optimality if optimal solution exist, otherwise it proves the non-existence of the optimal solution. From geometrical point of view the result of the introducing of the cuts is that the shape of the polyhedral set of the last LP relaxation is very similar to the integer hull *in the neighborhood of the optimal solution*.

There is the generalization of Gomory's cut called Chvátal (or Chvátal-Gomory) cut. If the two inequalities of (24.36) are summed such that both have weight $\frac{1}{6}$ then the constraint

$$x_1 \le 2.5$$

is obtained. As x_1 must be integer the inequality

$$x_1 \le 2 \tag{24.75}$$

follows immediately. It is not an algebraic consequence of the original constraints. To obtain it the information of the integrality of the variables had to be used. But the method can be continued. If (24.75) has weight $\frac{2}{5}$ and the second constraint of (24.36) has weight $\frac{1}{5}$ then

$$x_1 + x_2 \le 3.8$$

is obtained implying

$$x_1 + x_2 \le 3\,.$$

If the last inequality has weight $\frac{5}{8}$ and the first inequality of (24.36) has weight $\frac{1}{8}$ then the result is

$$x_1 \le \frac{15}{8}$$

implying

$$x_1 \le 1.$$

Finally the integer hull is obtained. In general the idea is as follows. Assume that a polyhedral set is defined by the linear inequality system

$$\mathbf{A}\mathbf{x} \le \mathbf{b}\,. \tag{24.76}$$

Let $\mathbf{y} \ge \underline{0}$ be a vector such that $\mathbf{A}^T\mathbf{y}$ is an integer vector and $\mathbf{y}^T\mathbf{b}$ is a noninteger value. Then

$$\mathbf{y}^T\mathbf{A}\mathbf{x} \;\le\; \lfloor \mathbf{y}^T\mathbf{b} \rfloor$$

is a valid cut, i.e. all integer points of the polyhedral set satisfy it. As a matter of fact it can be proven that a systematic application of the method creates a complete description of the integer hull after finite many steps.

The example shows that Gomory and Chvátal cuts can help to solve a problem. On the other hand they can be incorporated in a B&B frame easily. But in the very general case it is hopeless to generate all effective cuts of this type.

The situation is significantly different in the case of many combinatorial problems. There are many theoretical results known on the type of the facet defining constraints of special polyhedral sets. Here only one example is discussed. It is the

Traveling Salesperson Problem (TSP). A salesman must visit some cities and at the end of the tour he must return to his home city. The problem is to find a tour with minimal possible length. TSP has many applications including cases when the "cities" are products or other objects and the “distance” among them doesn’t satisfy the properties of the geometric distances, i.e. symmetry and triangle inequality may be violated.

The first exact mathematical formulation of the problem is the so-called Dantzig-Fulkerson-Johnson (DFJ) model. DFJ is still the basis of the numerical solutions. Assume that the number of cities is n. Let d_{ij} the distance of the route from city i to city j $(1 \leq i, j \leq n,\ i \neq j)$. DFJ uses the variables x_{ij} such that

$$x_{ij} = \begin{cases} 1 & \text{if the salesman travel from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

The objective function is the minimization on the total travel length:

$$\min \sum_{i=1}^{n} \sum_{i \neq j} d_{ij} x_{ij}\,. \tag{24.77}$$

The set of the constraints consists of three parts. The meaning of the first part is that the salesman must travel from each city to another city exactly once:

$$\sum_{j=1, j\neq i}^{n} x_{ij} = 1 \quad i = 1, \ldots, n\,. \tag{24.78}$$

The second part is very similar. It claims that the salesman must arrive to each city from somewhere else again exactly once:

$$\sum_{i=1, i\neq j}^{n} x_{ij} = 1 \quad j = 1, \ldots, n\,. \tag{24.79}$$

Constraints (24.78) and (24.79) are the constraints of an assignment problem. Taking into account that the variables must be binary Problem (24.77)-(24.79) is really an assignment problem. They don’t exclude solutions consisting of several smaller tours. For example if $n = 6$ and $x_{12} = x_{23} = x_{31} = 1$ and $x_{45} = x_{56} = x_{64} = 1$ then all other variables must be zero. The solution consists of two smaller tours. The first one visits only cities 1, 2, and 3, the second one goes through the cities 4, 5, and 6. The small tours are called *subtours* in the language of the theory of TSP.

Thus further constraints are needed which excludes the subtours. They are called *subtour elimination constraints*. There are two kinds of logic how the subtours can be excluded. The first one claims that in any subset of the cities which has at least two elements but not the complete set of the cities the number of travels must be less than the number of elements of the set. The logic can be formalized as follows:

$$\forall \mathcal{S} \subset \{1, 2, \ldots, n\},\ 1 \leq |\mathcal{S}| \leq n-1 : \sum_{i \in \mathcal{S}} \sum_{j \in \mathcal{S} j \neq i} x_{ij} \leq |\mathcal{S}|\,. \tag{24.80}$$

The other logic claims that the salesman must leave all such sets. Let $\bar{\mathcal{S}} = \{1, 2, \ldots, n\} \setminus \mathcal{S}$. Then the subtour elimination constraints are the inequalities

$$\forall \mathcal{S} \subset \{1, 2, \ldots, n\},\ 1 \leq |\mathcal{S}| \leq n-1: \ \sum_{i \in \mathcal{S}} \sum_{j \in \bar{\mathcal{S}}} x_{ij} \geq 1\,. \tag{24.81}$$

The numbers of the two types of constraints are equal and exponential. Although the constraints (24.78)–(24.80) or (24.78), (24.79), and (24.81) are satisfied by only binary vectors being characteristic vectors of complete tours but the polyhedral set of the LP relaxation is strictly larger than the integer hull.

On the other hand it is clear that it is not possible to claim all of the subtour elimination constraints in the real practice. What can be done? It is possible to claim only the violated once. The difficulty is that the optimal solution of the LP relaxation is a fractional vector in most of the cases and that subtour elimination constraint must be found which is violated by the fractional solution provided that such constraint exists as the subtour elimination constraints are necessary to the description of the integer hull but further constraints are needed, too. Thus it is possible that there is no violated subtour elimination constraint but the optimal solution of the LP relaxation is still fractional.

To find a violated subtour elimination constraint is equivalent to the finding of the absolute minimal cut in the graph which has only the edges having positive weights in the optimal solution of the relaxed problem. If the value of the absolute minimal cut is less than 1 in the directed case or less than 2 in the non-directed case then such a violated constraint exists. The reason can be explained based on the second logic of the constraints. If the condition is satisfied then the current solution doesn't leaves at least one of the two sets of the cut in enough number. There are many effective methods to find the absolute minimal cut.

A general frame of the numerical solution of the TSP is the following. In a B&B frame the calculation of the lower bound is repeated until a new violated subtour elimination constraint is obtained, that is the new inequality is added to the relaxed problem and the LP optimization is carried out again. If all subtour elimination constraints are satisfied and the optimal solution of the relaxed problem is still non-integer then branching is made according to a fractional valued variable.

The frame is rather general. The violated constraint cuts the previous optimal solution and reoptimization is needed. Gomory cuts do the same for the general integer programming problem. In the case of other combinatorial problems special cuts may work if the description of the integer hull is known.

Thus the general idea of B&C is that a cut is generated until it can be found and the improvement in the lower bound is great enough. Otherwise branching is made by a non-integer variable. If the cut generation is made only at the root of the enumeration tree then the name of the method is *Cut and Branch* (C&B). If a cut is generated in a branch then it is locally valid in that branch and in its successors. The cuts generated at the root are valid globally, i.e. in all branches. In some cases, e.e. in binary optimization, it is possible to modify it such that it is valid in the original problem, too.

For practical reasons the type of the generated cut can be restricted. It is the case in TSP as the subtour elimination constraints can be found relatively easily.

24.7. Branch and Price

The ***Branch and Price method*** is the dual of B&C in a certain sense. If a problem has very large number of variables then it is often possible not to work explicitely with all of them but generate only those which may enter the basis of the LP relaxation. This is column generation and is based on the current values of the dual variables called shadow prices. Similarly to B&C the type of the generated columns is restricted. If it is not possible to find a new column then branching is made.

Problems

24-1 Continuous Knapsack Problem
Prove Theorem 24.1. (*Hint.* Let $\mathbf{x}$ be a feasible solution such that there are two indices, say j and k, such that $1 \le j < k \le n$ and $x_j < 1$, and $x_k > 0$. Show that the solution can be improved.)
24-2 TSP's relaxation
Decide if the Assignment Problem can be a relaxation of the Traveling Salesperson Problem in the sense of definition 24.5. Explain your solution regardless that your answer is YES or NO.
24-3 Infeasibility test
Based on the the second observation of Subsection 24.5.2 develop a test for the infeasibility of a linear constraint of binary variables.
24-4 Mandatory fixing
Based on the previous problem develop a test for the mandatory fixing of binary variables satisfying a linear constraint.

Chapter Notes

The principle of B&B first appears in [89]. It solves problems with bounded integer variables. The fast bounds were introduced in [13] and [150]. A good summary of the bounds is [51]. To the best knowledge of the author of this chapter the improvement of the fast bounds appeared first in [152].

B&B can be used as an approximation scheme, too. In that case a branch can be deleted even in the case if its bound is not greater than the objective function value of the current best solution plus an allowed error. [70] showed that there are classes such that the approximate method requires more computation than to solve the problem optimally. B&B is very suitable for parallel processing. This issue is discussed in [22].

Based on the theoretical results of [98] a very effective version of B&C method was developed for pure binary optimization problem by [140] and independently [9]. Especially Egon Balas and his co-authors could achieve a significant progress. Their method of lifting cuts means that a locally generated cut can be made globally valid by solving a larger LP problem and modify the cut according to its optimal solution.

The first integer programming method to solve an IP problem with general, i.e. non-bounded, integer variables is Ralph Gomory's cutting plane method [55]. In a certain sense it is still the only general method. Strong cuts of integer programming problems are discussed in [10]. The surrogate constraint (24.18) has been introduced by [54]. The strength of the inequality depends on the choice of the multipliers λ_i. A rule of thumb is that the optimal dual variables of the continuous problem give a strong inequality provided that the original problem is linear.

The DFJ model of TSP appeared in [36]. It was not only an excellent theoretical result, but is also an enormous computational effort as the capacities and speed of that time computers were far above the recent ones. One important cut of the TSP polyhedral set is the so-called comb inequality. The number of edges of a complete tour is restricted in a special subgraph. The subgraph consists of a subset of cities called *hand* and odd number of further subsets of cities intersecting the hand. They are called *teeth* and their number must be at least three. Numerical problems of TSP are exhaustively discussed in [137].

A good summary of Branch and Price is [12].

25. Comparison Based Ranking

Let $a,\ b\ (b \geq a)$ and $n\ (n \geq 2)$ be nonnegative integers and let $\mathcal{T}(a,b,n)$ be the set of such generalised tournaments, in which every pair of distinct players is connected at most with b, and at least with a arcs. In [77] we gave a necessary and sufficient condition to decide whether a given sequence of nonnegative integers $D = (d_1, d_2, \ldots, d_n)$ can be realized as the outdegree sequence of a $T \in \mathcal{T}(a,b,n)$. Extending the results of [77] we show that for any sequence of nonnegative integers D there exist f and g such that some element $T \in \mathcal{T}(g,f,n)$ has D as its outdegree sequence, and for any (a,b,n)-tournament T' with the same outdegree sequence D hold $a \leq g$ and $b \geq f$. We propose a $\Theta(n)$ algorithm to determine f and g and an $O(d_n n^2)$ algorithm to construct a corresponding tournament T.

25.1. Basic concepts and notations

Let $a,\ b\ (b \geq a)$ and $n\ (n \geq 2)$ be nonnegative integers and let $\mathcal{T}(a,b,n)$ be the set of such generalised tournaments, in which every pair of distinct players is connected at most with b, and at least with a arcs. The elements of $\mathcal{T}(a,b,n)$ are called ***(a,b,n)-tournaments.*** The vector $D = (d_1, d_2, \ldots, d_n)$ of the outdegrees of $T \in \mathcal{T}(a,b,n)$ is called ***the score vector*** of T. If the elements of D are in nondecreasing order, then D is called the ***score sequence*** of T.

An arbitrary vector $D = (d_1, d_2, \ldots, d_n)$ of nonnegative integers is called ***graphical vector,*** iff there exists a loopless multigraph whose degree vector is D, and D is called ***digraphical vector*** (or *score vector*) iff there exists a loopless directed multigraph whose outdegree vector is D.

A nondecreasingly ordered graphical vector is called ***graphical sequence,***, and a nondecreasinly ordered digraphical vector is called ***digraphical sequence*** (or *score sequence*).

The number of arcs of T going from player P_i to player P_j is denoted by m_{ij} $(1 \leq i,j \leq n)$, and the matrix $\mathcal{M} = [1.\ .n, 1.\ .n]$ is called ***point matrix*** or *tournament matrix* of T.

In the last sixty years many efforts were devoted to the study of both types of vectors, resp. sequences. E.g. in the papers [16, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?] the graphical sequences, while in the papers [7, ?, 16, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 90, ?, 105, 106, 107, ?, ?, ?, ?, 135, ?, ?, ?, ?, ?] the score sequences were discussed.

Even in the last two years many authors investigated the conditions, when D

is graphical (e.g. [?, ?]) or digraphical (e.g. [?, ?, 77, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]).

In this paper we deal only with directed graphs and usually follow the terminology used by K. B. Reid [?, 136]. If in the given context a, b and n are fixed or non important, then we speak simply on *tournaments* instead of generalised or (a, b, n)-tournaments.

We consider the loopless directed multigraphs as generalised tournaments, in which the number of arcs from vertex/player P_i to vertex/player P_j is denoted by m_{ij}, where m_{ij} means the number of points won by player P_i in the match with player P_j.

The first question: how one can characterise the set of the score sequences of the (a, b, n)-tournaments. Or, with another words, for which sequences D of nonnegative integers does exist an (a, b, n)-tournament whose outdegree sequence is D. The answer is given in Section 25.2.

If T is an (a, b, n)-tournament with point matrix $\mathcal{M} = [1. .n, 1. .n]$, then let $E(T)$, $F(T)$ and $G(T)$ be defined as follows: $E(T) = \max_{1 \le i,j \le n} m_{ij}$, $F(T) = \max_{1 \le i < j \le n}(m_{ij} + m_{ji})$, and $g(T) = \min_{1 \le i < j \le n}(m_{ij} + m_{ji})$. Let $\Delta(D)$ denote the set of all tournaments having D as outdegree sequence, and let $e(D)$, $f(D)$ and $g(D)$ be defined as follows: $e(D) = \{\min \ E(T) \mid T \in \Delta(D)\}$, $f(D) = \{\min \ F(T) \mid T \in \Delta(D)\}$, and $g(D) = \{\max \ G(T) \mid T \in \Delta(D)\}$. In the sequel we use the short notations E, F, G, e, f, g, and Δ.

Hulett et al. [?, ?], Kapoor et al. [?], and Tripathi et al. [?, ?] investigated the construction problem of a minimal size graph having a prescribed degree set [132, 154]. In a similar way we follow a mini-max approach formulating the following questions: given a sequence D of nonnegative integers,

- How to compute e and how to construct a tournament $T \in \Delta$ characterised by e? In Section 25.3 a formula to compute e, and an algorithm to construct a corresponding tournament are presented.

- How to compute f and g? In Section 25.4 an algorithm to compute f and g is described.

- How to construct a tournament $T \in \Delta$ characterised by f and g? In Section 25.5 an algorithm to construct a corresponding tournament is presented and analysed.

We describe the proposed algorithms in words, by examples and by the pseudocode used in [33].

Researchers of these problems often mention different applications, e.g. in biology [90], chemistry Hakimi [?], and Kim et al. in networks [?].

25.2. Existence of a tournament with arbitrary degree sequence

Since the numbers of points m_{ij} are not limited, it is easy to construct a $(0, d_n, n)$-tournament for any D.

Lemma 25.1 *If $n \geq 2$, then for any vector of nonnegative integers $D = (d_1, d_2, \ldots, d_n)$ there exists a loopless directed multigraph T with outdegree vector D so, that $E \leq d_n$.*

Proof Let $m_{n1} = d_n$ and $m_{i,i+1} = d_i$ for $i = 1, 2, \ldots, n-1$, and let the remaining m_{ij} values be equal to zero. ■

Using weighted graphs it would be easy to extend the definition of the (a, b, n)-tournaments to allow *arbitrary real values* of a, b, and D. The following algorithm Naive-Construct works without changes also for input consisting of real numbers.

We remark that Ore in 1956 [?] gave the necessary and sufficient conditions of the existence of a tournament with prescribed indegree and outdegree vectors. Further Ford and Fulkerson [?, Theorem11.1] published in 1962 necessary and sufficient conditions of the existence of a tournament having prescribed lower and upper bounds for the indegree and outdegree of the vertices. They results also can serve as basis of the existence of a tournament having arbitrary outdegree sequence.

25.2.1. Definition of a naive reconstructing algorithm

Sorting of the elements of D is not necessary.

Input. n: the number of players ($n \geq 2$);
$D = (d_1, d_2, \ldots, d_n)$: arbitrary sequence of nonnegative integer numbers.

Output. $\mathcal{M} = [1. .n, 1. .n]$: the point matrix of the reconstructed tournament.

Working variables. i, j: cycle variables.

```
Naive-Construct($n, D$)
01 for $i = 1$ to $n$
02     for $j = 1$ to $n$
03         $m_{ij} = 0$
04 $m_{n1} = d_n$
05 for $i = 1$ to $n - 1$
06     $m_{i,i+1} = d_i$
07 return $\mathcal{M}$
```

The running time of this algorithm is $\Theta(n^2)$ in worst case (in best case too). Since the point matrix $\mathcal{M}$ has n^2 elements, this algorithm is asymptotically optimal.

25.3. Computation of e

This is also an easy question. From here we suppose that D is a nondecreasing sequence of nonnegative integers, that is $0 \leq d_1 \leq d_2 \leq \ldots \leq d_n$. Let $h = \lceil d_n/(n-1) \rceil$.

Since $\Delta(D)$ is a finite set for any finite score vector D, $e(D) = \min\{E(T) | T \in \Delta(D)\}$ exists.

Lemma 25.2 *If $n \geq 2$, then for any sequence $D = (d_1, d_2, \ldots, d_n)$ there exists a*

$(0, b, n)$-tournament T such that

$$E \le h \qquad \textit{and} \quad b \le 2h, \tag{25.1}$$

and h is the smallest upper bound for e, and $2h$ is the smallest possible upper bound for b.

Proof If all players gather their points in a uniform as possible manner, that is

$$\max_{1 \le j \le n} m_{ij} - \min_{1 \le j \le n,\ i \ne j} m_{ij} \le 1 \quad \text{for } i = 1,\ 2,\ \ldots,\ n, \tag{25.2}$$

then we get $E \le h$, that is the bound is valid. Since player P_n has to gather d_n points, the pigeonhole principle [?, ?] implies $E \ge h$, that is the bound is not improvable. $E \le h$ implies $\max_{1 \le i < j \le n} m_{ij} + m_{ji} \le 2h$. The score sequence $D = (d_1, d_2, \ldots, d_n) = (2n(n-1), 2n(n-1), \ldots, 2n(n-1))$ shows, that the upper bound $b \le 2h$ is not improvable. ■

Corollary 25.3 *If $n \ge 2$, then for any sequence $D = (d_1, d_2, \ldots, d_n)$ holds $e(D) = \lceil d_n/(n-1) \rceil$.*

Proof According to Lemma 25.2 $h = \lceil d_n/(n-1) \rceil$ is the smallest upper bound for e. ■

25.3.1. Definition of a construction algorithm

The following algorithm constructs a $(0, 2h, n)$-tournament T having $E \le h$ for any D.

Input. n: the number of players $(n \ge 2)$;
$D = (d_1, d_2, \ldots, d_n)$: arbitrary sequence of nonnegative integer numbers.
Output. $\mathcal{M} = [1..n, 1..n]$: the point matrix of the tournament.
Working variables. i, j, l: cycle variables;
k: the number of the "larger parts" in the uniform distribution of the points.

```
PIGEONHOLE-CONSTRUCT($n, D$)
01 for $i = 1$ to $n$
02     $m_{ii} = 0$
03     $k = d_i - (n-1)\lfloor d_i/(n-1) \rfloor$
04     for $j = 1$ to $k$
05         $l = i + j \pmod n$
06         $m_{il} = \lceil d_n/(n-1) \rceil$
07     for $j = k+1$ to $n-1$
08         $l = i + j \pmod n$
09         $m_{il} = \lfloor d_n/(n-1) \rfloor$
10 return $\mathcal{M}$
```

The running time of PIGEONHOLE-CONSTRUCT is $\Theta(n^2)$ in worst case (in best case too). Since the point matrix $\mathcal{M}$ has n^2 elements, this algorithm is asymptotically optimal.

25.4. Computation of f and g

Let S_i $(i = 1,\ 2,\ \ldots,\ n)$ be the sum of the first i elements of D, B_i $(i = 1,\ 2,\ \ldots,\ n)$ be the binomial coefficient $n(n-1)/2$. Then the players together can have S_n points only if $fB_n \geq S_n$. Since the score of player P_n is d_n, the pigeonhole principle implies $f \geq \lceil d_n/(n-1) \rceil$.

These observations result the following lower bound for f:

$$f \geq \max\left(\left\lceil \frac{S_n}{B_n} \right\rceil, \left\lceil \frac{d_n}{n-1} \right\rceil\right). \qquad (25.3)$$

If every player gathers his points in a uniform as possible manner then

$$f \leq 2\left\lceil \frac{d_n}{n-1} \right\rceil. \qquad (25.4)$$

These observations imply a useful characterisation of f.

Lemma 25.4 *If $n \geq 2$, then for arbitrary sequence $D = (d_1, d_2, \ldots, d_n)$ there exists a (g, f, n)-tournament having D as its outdegree sequence and the following bounds for f and g:*

$$\max\left(\left\lceil \frac{S}{B_n} \right\rceil, \left\lceil \frac{d_n}{n-1} \right\rceil\right) \leq f \leq 2\left\lceil \frac{d_n}{n-1} \right\rceil, \qquad (25.5)$$

$$0 \leq g \leq f. \qquad (25.6)$$

Proof (25.5) follows from (25.3) and (25.4), (25.6) follows from the definition of f. ■

It is worth to remark, that if $d_n/(n-1)$ is integer and the scores are identical, then the lower and upper bounds in (25.5) coincide and so Lemma 25.4 gives the exact value of F.

In connection with this lemma we consider three examples. If $d_i = d_n = 2c(n-1)$ $(c > 0,\ i = 1,\ 2,\ \ldots,\ n-1)$, then $d_n/(n-1) = 2c$ and $S_n/B_n = c$, that is S_n/B_n is twice larger than $d_n/(n-1)$. In the other extremal case, when $d_i = 0$ $(i = 1,\ 2,\ \ldots, n-1)$ and $d_n = cn(n-1) > 0$, then $d_n/(n-1) = cn$, $S_n/B_n = 2c$, so $d_n/(n-1)$ is $n/2$ times larger, than S_n/B_n.

If $D = (0, 0, 0, 40, 40, 40)$, then Lemma 25.4 gives the bounds $8 \leq f \leq 16$. Elementary calculations show that Figure 25.1 contains the solution with minimal f, where $f = 10$.

In [77] we proved the following assertion.

Theorem 25.5 *For $n \geq 2$ a nondecreasing sequence $D = (d_1, d_2, \ldots, d_n)$ of non-negative integers is the score sequence of some (a, b, n)-tournament if and only if*

$$aB_k \leq \sum_{i=1}^{k} d_i \leq bB_n - L_k - (n-k)d_k \quad (1 \leq k \leq n), \qquad (25.7)$$

Player/Player	P$_1$	P$_2$	P$_3$	P$_4$	P$_5$	P$_5$	Score
P$_1$	—	0	0	0	0	0	0
P$_2$	0	—	0	0	0	0	0
P$_3$	0	0	—	0	0	0	0
P$_4$	10	10	10	—	5	5	40
P$_5$	10	10	10	5	—	5	40
P$_6$	10	10	10	5	5	—	40

Figure 25.1 Point matrix of a $(0, 10, 6)$-tournament with $f = 10$ for $D = (0, 0, 0, 40, 40, 40)$.

where

$$L_0 = 0, \text{ and } L_k = \max\left(L_{k-1},\ bB_k - \sum_{i=1}^{k} d_i\right) \quad (1 \le k \le n). \tag{25.8}$$

The theorem proved by Moon [106], and later by Kemnitz and Dolff [?] for (a, a, n)-tournaments is the special case $a = b$ of Theorem 25.5. Theorem 3.1.4 of [?] is the special case $a = b = 2$. The theorem of Landau [90] is the special case $a = b = 1$ of Theorem 25.5.

25.4.1. Definition of a testing algorithm

The following algorithm Interval-Test decides whether a given D is a score sequence of an (a, b, n)-tournament or not. This algorithm is based on Theorem 25.5 and returns W = True if D is a score sequence, and returns W = False otherwise.

Input. a: minimal number of points divided after each match;
b: maximal number of points divided after each match.
Output. W: logical variable (W = True shows that D is an (a, b, n)-tournament.
Local working variable. i: cycle variable;
$L = (L_0, L_1, \ldots, L_n)$: the sequence of the values of the loss function.
Global working variables. n: the number of players ($n \ge 2$);
$D = (d_1, d_2, \ldots, d_n)$: a nondecreasing sequence of nonnegative integers;
$B = (B_0, B_1, \ldots, B_n)$: the sequence of the binomial coefficients;
$S = (S_0, S_1, \ldots, S_n)$: the sequence of the sums of the i smallest scores.

```
Interval-Test($a, b$)
01 for $i = 1$ to $n$
02     $L_i = \max(L_{i-1},\ bB_n - S_i - (n-i)d_i)$
03     if $S_i < aB_i$
04        $W$ = False
05        return $W$
06        if $S_i > bB_n - L_i - (n-i)d_i$
07           $W \leftarrow$ False
08           return $W$
09 return $W$
```

In worst case INTERVAL-TEST runs in $\Theta(n)$ time even in the general case $0 < a < b$ (n the best case the running time of INTERVAL-TEST is $\Theta(n)$). It is worth to mention, that the often referenced Havel–Hakimi algorithm [**?**, **?**] even in the special case $a = b = 1$ decides in $\Theta(n^2)$ time whether a sequence D is digraphical or not.

25.4.2. Definition of an algorithm computing f and g

The following algorithm is based on the bounds of f and g given by Lemma 25.4 and the logarithmic search algorithm described by D. E. Knuth [**?**, page 410].

Input. No special input (global working variables serve as input).
Output. b: f (the minimal F);
a: g (the maximal G).
Local working variables. i: cycle variable;
l: lower bound of the interval of the possible values of F;
u: upper bound of the interval of the possible values of F.
Global working variables. n: the number of players ($n \geq 2$);
$D = (d_1, d_2, \ldots, d_n)$: a nondecreasing sequence of nonnegative integers;
$B = (B_0, B_1, \ldots, B_n)$: the sequence of the binomial coefficients;
$S = (S_0, S_1, \ldots, S_n)$: the sequence of the sums of the i smallest scores;
W: logical variable (its value is TRUE, when the investigated D is a score sequence).

```
MinF-MaxG
01 $B_0 = S_0 = L_0 = 0$                         ▷ Initialization
02 for $i = 1$ to $n$
03      $B_i = B_{i-1} + i - 1$
04      $S_i = S_{i-1} + d_i$
05 $l = \max(\lceil S_n/B_n \rceil, \lceil d_n/(n-1) \rceil)$
06 $u = 2\lceil d_n/(n-1) \rceil$
07 $W$ = True                                     ▷ Computation of $f$
08 Interval-Test$(0, l)$
09 if $W$ == True
10    $b = l$
11    go to 21
12 $b = \lceil (l+u)/2 \rceil$
13 Interval-Test$(0, f)$
14 if $W$ == True
15    go to 17
16 $l = b$
17 if $u == l + 1$
18    $b = u$
19    go to 37
20 go to 14
21 $l = 0$                                        ▷ Computation of $g$
22 $u = f$
23 Interval-Test$(b, b)$
24 if $W$ == True
25    $a \leftarrow f$
26    go to 37
27 $a = \lceil (l+u)/2 \rceil$
28 Interval-Test$(0, a)$
29 if $W$ == True
30    $l \leftarrow a$
31    go to 33
32 $u = a$
33 if $u == l + 1$
34    $a = l$
35    go to 37
36 go to 27
39 return $a, b$
```

MinF-MaxG determines f and g.

Lemma 25.6 *Algorithm* MinG-MaxG *computes the values f and g for arbitrary sequence $D = (d_1, d_2, \ldots, d_n)$ in $O(n \log(d_n/(n))$ time.*

Proof According to Lemma 25.4 F is an element of the interval $[\lceil d_n/(n-1) \rceil, \lceil 2d_n/(n-1) \rceil]$ and g is an element of the interval $[0, f]$. Using Theorem B of [**?**, page 412] we get that $O(\log(d_n/n))$ calls of Interval-Test is sufficient, so the $O(n)$

run time of Interval-Test implies the required running time of MinF-MaxG. ■

25.4.3. Computing of f and g in linear time

Analysing Theorem 25.5 and the work of algorithm MinF-MaxG one can observe that the maximal value of G and the minimal value of F can be computed independently by Linear-MinF-MaxG.

Input. No special input (global working variables serve as input).
Output. b: f (the minimal F).
a: g (the maximal G).
Local working variables. i: cycle variable.
Global working variables. n: the number of players ($n \geq 2$);
$D = (d_1, d_2, \ldots, d_n)$: a nondecreasing sequence of nonnegative integers;
$B = (B_0, B_1, \ldots, B_n)$: the sequence of the binomial coefficients;
$S = (S_0, S_1, \ldots, S_n)$: the sequence of the sums of the i smallest scores.

```
Linear-MinF-MaxG
01 $B_0 = S_0 = L_0 = 0$                         ▷ Initialization
02 for $i = 1$ to $n$
03     $B_i = B_{i-1} + i - 1$
04     $S_i = S_{i-1} + d_i$
05 $a = 0)$
06 $b = \min 2\lceil d_n/(n-1)\rceil$
07 for $i = 1$ to $n$                             ▷ Computation of $g$
08     $a_i = \left\lceil(2S_i/(n^2 - n)\right\rceil) < a$
09     if $a_i > a$
10     $a = a_i$
11 for $i = 1$ to $n$                             ▷ Computation of $f$
12     $L_i = \max(L_{i-1}, bB_n - S_i - (n-i)d_i$
13     $b_i = (S_i + (n-i)d_i + L_i)/B_i$
14     if $b_i < b$
15     $b = b_i$
16 return $a, b$
```

Lemma 25.7 *Algorithm* Linear-MinG-MaxG *computes the values f and g for arbitrary sequence $D = (d_1, d_2, \ldots, d_n)$ in $\Theta(n)$ time.*

Proof Lines 01–03, 07, and 18 require only constant time, lines 04–06, 09–12, and 13–17 require $\Theta(n)$ time, so the total running time is $\Theta(n)$. ■

25.5. Tournament with f and g

The following reconstruction algorithm Score-Slicing2 is based on balancing between additional points (they are similar to "excess", introduced by Brauer et al. [?])

and missing points introduced in [77]. The greediness of the algorithm Havel–Hakimi [?, ?] also characterises this algorithm.

This algorithm is an extended version of the algorithm Score-Slicing proposed in [77].

25.5.1. Definition of the minimax reconstruction algorithm

The work of the slicing program is managed by the following program Mini-Max.

Input. n: the number of players ($n \geq 2$);
$D = (d_1, d_2, \ldots, d_n)$: a nondecreasing sequence of integers satisfying (25.7).

Output. $\mathcal{M} = [1\,..\,n, 1\,..\,n]$: the point matrix of the reconstructed tournament.

Local working variables. $i,\ j$: cycle variables.

Global working variables. $p = (p_0, p_1, \ldots, p_n)$: provisional score sequence;
$P = (P_0, P_1, \ldots, P_n)$: the partial sums of the provisional scores;
$\mathcal{M}[1\,..\,n, 1\,..\,n]$: matrix of the provisional points.

```
Mini-Max($n, D$)
01 MinF-MaxG($n, D$)                        ▷ Initialization
02 $p_0 = 0$
03 $P_0 = 0$
04 for $i = 1$ to $n$
05     for $j = 1$ to $i - 1$
06         $\mathcal{M}[i, j] = b$
07         for $j = i$ to $n$
08             $\mathcal{M}[i, j] = 0$
09     $p_i = d_i$
10 if $n \geq 3$                             ▷ Score slicing for $n \geq 3$ players
11    for $k = n$ downto 3
12        Score-Slicing2($k$)
13 if $n == 2$                               ▷ Score slicing for 2 players
14    $m_{1,2} = p_1$
15    $m_{2,1} = p_2$
16 return $\mathcal{M}$
```

25.5.2. Definition of the score slicing algorithm

The key part of the reconstruction is the following algorithm Score-Slicing2 [77].

During the reconstruction process we have to take into account the following bounds:

$$a \leq m_{i,j} + m_{j,i} \leq b \quad (1 \leq i < j \leq n); \qquad (25.9)$$

$$\text{modified scores have to satisfy (25.7);} \qquad (25.10)$$

$$m_{i,j} \leq p_i \ (1 \leq i,\ j \leq n, i \neq j); \qquad (25.11)$$

$$\text{the monotonicity } p_1 \leq p_2 \leq \ldots \leq p_k \text{ has to be saved } \ (1 \leq k \leq n) \qquad (25.12)$$

$$m_{ii} = 0 \quad (1 \leq i \leq n). \qquad (25.13)$$

Input. k: the number of the actually investigated players ($k > 2$);
$p_k = (p_0, p_1, p_2, \ldots, p_k)$ ($k = 3,\ 4,\ \cdots,\ n$): prefix of the provisional score sequence p;
$\mathcal{M}[1 \mathinner{.\,.} n, 1 \mathinner{.\,.} n]$: matrix of provisional points;

Output. Local working variables. $A = (A_1, A_2, \ldots, A_n)$ the number of the additional points;
M: missing points: the difference of the number of actual points and the number of maximal possible points of P_k;
d: difference of the maximal decreasable score and the following largest score;
y: number of sliced points per player;
f: frequency of the number of maximal values among the scores $p_1,\ p_2,\ \ldots,\ p_{k-1}$;
$i,\ j$: cycle variables;
m: maximal amount of sliceable points;
$P = (P_0, P_1, \ldots, P_n)$: the sums of the provisional scores;
x: the maximal index i with $i < k$ and $m_{i,k} < b$.

Global working variables: n: the number of players ($n \geq 2$);
$B = (B_0, B_1, B_2, \ldots, B_n)$: the sequence of the binomial coefficients;
a: minimal number of points divided after each match;
b: maximal number of points divided after each match.

```
SCORE-SLICING2($k$)
01 for $i = 1$ to $k - 1$                      ▷ Initialization
02     $P_i = P_{i-1} + p_i$
03     $A_i = P_i - aB_i$
04 $M = (k-1)b - p_k$
05 while $M > 0$ and $A_{k-1} > 0$             ▷ There are missing and additional points
06         $x = k - 1$
07         while $r_{x,k} = b$
08             $x = x - 1$
09         $f = 1$
10         while $p_{x-f+1} = p_{x-f}$
11             $f = f + 1$
12         $d = p_{x-f+1} - p_{x-f}$
13         $m = \min(b, d, \lceil A_x/b \rceil, \lceil M/b \rceil)$
14         for $i = f$ downto 1
15             $y = \min(b - r_{x+1-i,k}, m, M, A_{x+1-i}, p_{x+1-i})$
16             $r_{x+1-i,k} = r_{x+1-i,k} + y$
17             $p_{x+1-i} = p_{x+1-i} - y$
18             $r_{k,x+1-i} = b - r_{x+1-i,k}$
19               $M = M - y$
20             for $j = i$ downto 1
21                 $A_{x+1-i} = A_{x+1-i} - y$
22 while $M > 0$                                ▷ No missing points
23         $i = k - 1$
24         $y = \max(m_{ki} + m_{ik} - a, m_{ki}, M)$
```

```
25          $r_{ki} = r_{ki} - y$
26          $M = M - y$
27          $i = i - 1$
28 **return** $\pi_k, M$
```

Let's consider an example. Figure 25.2 shows the point table of a $(2, 10, 6)$-tournament T.

Player/Player	P$_1$	P$_2$	P$_3$	P$_4$	P$_5$	P$_6$	Score
P$_1$	—	1	5	1	1	1	09
P$_2$	1	—	4	2	0	2	09
P$_3$	3	3	—	5	4	4	19
P$_4$	8	2	5	—	2	3	20
P$_5$	9	9	5	7	—	2	32
P$_6$	8	7	5	6	8	—	34

Figure 25.2 The point table of a $(2, 10, 6)$-tournament T.

The score sequence of T is D = (9,9,19,20,32,34). In [77] the algorithm Score-Slicing2 resulted the point table represented in Figure 25.3.

Player/Player	P$_1$	P$_2$	P$_3$	P$_4$	P$_5$	P$_6$	Score
P$_1$	—	1	1	6	1	0	9
P$_2$	1	—	1	6	1	0	9
P$_3$	1	1	—	6	8	3	19
P$_4$	3	3	3	—	8	3	20
P$_5$	9	9	2	2	—	10	32
P$_6$	10	10	7	7	0	—	34

Figure 25.3 The point table of T reconstructed by Score-Slicing2.

The algorithm Mini-Max starts with the computation of f. MinF-MaxG called in line 01 begins with initialization, including provisional setting of the elements of $\mathcal{M}$ so, that $m_{ij} = b$, if $i > j$, and $m_{ij} = 0$ otherwise. Then MinF-MaxG sets the lower bound $l = \max(9, 7) = 9$ of f in line 07 and tests it in line 10 Interval-Test. The test shows that $l = 9$ is large enough so Mini-Max sets $b = 9$ in line 12 and jumps to line 23 and begins to compute g. Interval-Test called in line 25 shows that $a = 9$ is too large, therefore MinF-MaxG continues with the test of $a = 5$ in line 30. The result is positive, therefore comes the test of $a = 7$, then the test of $a = 8$. Now $u = l + 1$ in line 35, so $a = 8$ is fixed, and the control returns to line 02 of Mini-Max.

Lines 02–09 contain initialization, and Mini-Max begins the reconstruction of a $(8, 9, 6)$-tournament in line 10. The basic idea is that Mini-Max successively determines the won and lost points of P$_6$, P$_5$, P$_4$ and P$_3$ by repeated calls of Score-Slicing2 in line 12, and finally it computes directly the result of the match between

P_2 and P_1.

At first MINI-MAX computes the results of P_6 calling calling SCORE-SLICING2 with parameter $k = 6$. The number of additional points of the first five players is $A_5 = 89 - 8 \cdot 10 = 9$ according to line 03, the number of missing points of P_6 is $M = 5 \cdot 9 - 34 = 11$ according to line 04. Then SCORE-SLICING2 determines the number of maximal numbers among the provisional scores $p_1, p_2, \dots, p_5$ ($f = 1$ according to lines 09–14) and computes the difference between p_5 and p_4 ($d = 12$ according to line 12). In line 13 we get, that $m = 9$ points are sliceable, and P_5 gets these points in the match with P_6 in line 16, so the number of missing points of P_6 decreases to $M = 11 - 9 = 2$ (line 19) and the number of additional point decreases to $A = 9 - 9 = 0$. Therefore the computation continues in lines 22–27 and m_{64} and m_{63} will be decreased by 1 resulting $m_{64} = 8$ and $m_{63} = 8$ as the seventh line and seventh column of Figure 25.4 show. The returned score sequence is $p = (9, 9, 19, 20, 23)$.

Player/Player	P_1	P_2	P_3	P_4	P_5	P_6	Score
P_1	—	4	4	0	0	0	9
P_2	4	—	4	1	0	0	9
P_3	4	4	—	7	4	0	19
P_4	7	7	1	—	5	0	20
P_5	8	8	4	3	—	9	32
P_6	9	9	8	8	0	—	34

Figure 25.4 The point table of T reconstructed by MINI-MAX.

Second time MINI-MAX calls SCORE-SLICING2 with parameter $k = 5$, and get $A_4 = 9$ and $M = 13$. At first A_4 gets 1 point, then A_3 and A_4 get both 4 points, reducing M to 4 and A_4 to 0. The computation continues in line 22 and results the further decrease of m_{54}, m_{53}, m_{52}, and m_{51} by 1, resulting $m_{54} = 3$, $m_{53} = 4$, $m_{52} = 8$, and $m_{51} = 8$ as the sixth row of Figure 25.4 shows.

Third time MINI-MAX calls SCORE-SLICING2 with parameter $k = 4$, and get $A_3 = 11$ and $M = 11$. At first P_3 gets 6 points, then P_3 further 1 point, and P_2 and P_1 also both get 1 point, resulting $m_{34} = 7$, $m_{43} = 2$, $m_{42} = 8$, $m_{24} = 1$, $m_{14} = 1$ and $m_{14} = 8$, further $A_3 = 0$ and $M = 2$. The computation continues in lines 22–27 and results a decrease of m_{43} by 1 point resulting $m_{43} = 1$, $m_{42=8}$, and $m_{41} = 8$, as the fifth row and fifth column of Figure 25.4 show. The returned score sequence is $p = (9, 9, 15)$.

Fourth time MINI-MAX calls SCORE-SLICING2 with parameter $k = 3$, and gets $A_2 = 10$ and $M = 9$. At first P_2 gets 6 points, then ... The returned point vector is $p = (4, 4)$.

Finally MINI-MAX sets $m_{12} = 4$ and $m_{21} = 4$ in lines 14–15 and returns the point matrix represented in Figure 25.4.

The comparison of Figures 25.3 and 25.4 shows a large difference between the simple reconstruction of SCORE-SLICING2 and the minimax reconstruction of MINI-MAX: while in the first case the maximal value of $m_{ij} + m_{ji}$ is 10 and the minimal value is 2, in the second case the maximum equals to 9 and the minimum equals to 8, that is the result is more balanced (the given D does not allow to build a perfectly balanced (k, k, n)-tournament).

25.5.3. Analysis of the minimax reconstruction algorithm

The main result of this paper is the following assertion.

Theorem 25.8 *If $n \geq 2$ is a positive integer and $D = (d_1, d_2, \ldots, d_n)$ is a nondecreasing sequence of nonnegative integers, then there exist positive integers f and g, and a (g, f, n)-tournament T with point matrix $\mathcal{M}$ such, that*

$$f = \min(m_{ij} + m_{ji}) \leq b, \tag{25.14}$$

$$g = \max m_{ij} + m_{ji} \geq a \tag{25.15}$$

for any (a, b, n)-tournament, and algorithm Linear-MinF-MaxG *computes f and g in $\Theta(n)$ time, and algorithm* Mini-Max *generates a suitable T in $O(d_n n^2)$ time.*

Proof The correctness of the algorithms Score-Slicing2, Min$F$-Max$G$ implies the correctness of Mini-Max.

Lines 1–46 of Mini-Max require $O(\log(d_n/n))$ uses of MinG-MaxF, and one search needs $O(n)$ steps for the testing, so the computation of f and g can be executed in $O(n \log(d_n/n))$ times.

The reconstruction part (lines 47–55) uses algorithm Score-Slicing2, which runs in $O(bn^3)$ time [77]. Mini-Max calls Score-Slicing2 $n - 2$ times with $f \leq 2\lceil d_n/n \rceil$, so $n^3 d_n/n = d_n n^2$ finishes the proof. ■

The property of the tournament reconstruction problem that the extremal values of f and g can be determined independently and so there exists a tournament T having both extremal features is called linking property. This concept was introduced by Ford and Fulkerson in 1962 [**?**] and later extended by A. Frank in [**?**].

25.6. Summary

A nondecreasing sequence of nonnegative integers $D = (d_1, d_2, \ldots, d_n)$ is a score sequence of a $(1, 1, 1)$-tournament, iff the sum of the elements of D equals to B_n and the sum of the first i $(i = 1,\ 2,\ \ldots,\ n - 1)$ elements of D is at least B_i [90].

D is a score sequence of a (k, k, n)-tournament, iff the sum of the elements of D equals to kB_n, and the sum of the first i elements of D is at least kB_i [**?**, 105].

D is a score sequence of an (a, b, n)-tournament, iff (25.7) holds [77].

In all 3 cases the decision whether D is digraphical requires only linear time.

In this paper the results of [77] are extended proving that for any D there exists an optimal minimax realization T, that is a tournament having D as its outdegree sequence and maximal G and minimal F in the set of all realization of D.

In a continuation [78] of this paper we construct balanced as possible tournaments in a similar way if not only the outdegree sequence but the indegree sequence is also given.

Chapter Notes

[7] [?] [?] [14] [?] [?] [?] [57]
[?] [?]
[77] [78]
[85] [90] [105] [106] [?]
[?] [?]

Acknowledgement. The author thanks András Frank (Eötvös Loránd University) for valuable advices concerning the application of flow theory and Péter L. Erdős (Alfréd Rényi Institute of Mathematics of HAS) for the consultation.

The research was supported by the project TÁMOP-4.2.B-09/1/KMR–2010-003 of Eötvös Loránd University.

26. Complexity of Words

The complexity of words is a continuously growing field of the combinatorics of words. Hundreds of papers are devoted to different kind of complexities. We try to present in this chapter far from beeing exhaustive the basic notions and results for finite and infinite words.

First of all we summarize the simple (classical) complexity measures, giving formulas and algorithms for several cases. After this, generalized complexities are treated, with different type of algorithms. We finish this chapter by presenting the palindrome complexity.

Finally, references from a rich bibliography are given.

26.1. Simple complexity measures

In this section simple (classical) complexities, as measures of the diversity of the subwords in finite and infinite words, are discussed. First, we present some useful notions related to the finite and infinite words with examples. Word graphs, which play an important role in understanding and obtaining the complexity, are presented in detail with pertinent examples. After this, the subword complexity (as number of subwords), with related notions, is expansively presented.

26.1.1. Finite words

Let A be a finite, nonempty set, called ***alphabet.*** Its elements are called ***letters*** or ***symbols.*** A string $a_1a_2\dots a_n$, formed by (not necessary different) elements of A, is a ***word.*** The length of the word $u = a_1a_2\dots a_n$ is n, and is denoted by $|u|$. The word without any element is the empty word, denoted by ε (sometimes λ). The set of all finite words over A is denoted by A^*. We will use the following notations too:

$$A^+ = A^* \setminus \{\varepsilon\}, \qquad A^n = \Big\{u \in A^* \mid |u| = n\Big\} = \Big\{a_1a_2\dots a_n \mid a_i \in A\Big\},$$

that is A^+ is the set of all finite and nonempty words over A, whilst A^n is the set of all words of length n over A. Obviously $A^0 = \{\varepsilon\}$. The sets A^* and A^+ are infinite denumerable sets.

We define in A^* the binary operation called ***concatenation*** (shortly ***catenation***). If $u = a_1a_2 \dots a_n$ and $v = b_1b_2 \dots b_m$, then

$$w = uv = a_1a_2 \dots a_nb_1b_2 \dots b_m, \qquad |w| = |u| + |v|\,.$$

This binary operation is associative, but not commutative. Its neutral element is ε because $\varepsilon u = u\varepsilon = u$. The set A^* with this neutral element is a monoid. We introduce recursively the ***power of a word:***

- $u^0 = \varepsilon$
- $u^n = u^{n-1}u$, if $n \geq 1$.

A word is ***primitive*** if it is no power of any word, so u is primitive if

$$u = v^n,\ v \neq \varepsilon \quad \Rightarrow \quad n = 1\,.$$

For example, $u = abcab$ is a primitive word, whilst $v = abcabc = (abc)^2$ is not.

The word $u = a_1a_2 \dots a_n$ is ***periodic*** if there is a value p, $1 \leq p < n$ such that

$$a_i = a_{i+p}, \text{ for all } i = 1, 2, \dots, n-p\,,$$

and p is the period of u. The least such p is the ***least period*** of u.

The word $u = abcabca$ is periodic with the least period $p = 3$.

Let us denote by (a, b) the greatest common divisor of the naturals a and b. The following result is obvious.

Theorem 26.1 *If u is periodic, and p and q are periods, then (p, q) is a period too.*

The ***reversal*** (or ***mirror image***) of the word $u = a_1a_2 \dots a_n$ is $u^R = a_na_{n-1} \dots a_1$. Obviously $\left(u^R\right)^R = u$. If $u = u^R$, then u is a ***palindrome.***

The word u is a ***subword*** (or ***factor***) of v if there exist the words p and q such that $v = puq$. If $pq \neq \varepsilon$, then u is a proper subword of v. If $p = \varepsilon$, then u is a ***prefix*** of v, and if $q = \varepsilon$, then u is a ***suffix*** of v. The set of all subwords of length n of u is denoted by $F_n(u)$. $F(u)$ is the set of nonempty subwords of u, so

$$F(u) = \bigcup_{n=1}^{|u|} F_n(u)\,.$$

For example, if $u = abaab$, then

$F_1(u) = \{a, b\},\ F_2(u) = \{ab, ba, aa\},\ F_3(u) = \{aba, baa, aab\},$
$F_4(u) = \{abaa, baab\},\ F_5(u) = \{abaab\}.$

The words $u = a_1a_2 \dots a_m$ and $v = b_1b_2 \dots b_n$ are equal, if

- $m = n$ and
- $a_i = b_i$, for $i = 1, 2, \dots, n\,.$

Theorem 26.2 (Fine–Wilf). *If u and v are words of length n, respective m, and if there are the natural numbers p and q, such that u^p and v^q have a common prefix of length $n + m - (n, m)$, then u and v are powers of the same word.*

The value $n+m-(n,m)$ in the theorem is tight. This can be illustrated by the following example. Here the words u and v have a common prefix of length $n+m-(n,m)-1$, but u and v are not powers of the same word.

$$\begin{array}{lll} u = abaab, & m = |u| = 5, & u^2 = abaababaab\,, \\ v = aba, & n = |v| = 3, & v^3 = abaabaaba\,. \end{array}$$

By the theorem a common prefix of length 7 would ensure that u and v are powers of the same word. We can see that u^2 and v^3 have a common prefix of length 6 ($abaaba$), but u and v are not powers of the same word, so the length of the common prefix given by the theorem is tight.

26.1.2. Infinite words

Beside the finite words we consider ***infinite*** (more precisely infinite at right) words too:

$$u = u_1u_2\ldots u_n\ldots, \quad \text{where } u_1, u_2, \ldots \in A\,.$$

The set of infinite words over the alphabet A is denoted by $A\omega$. If we will study together finite and infinite words the following notation will be useful:

$$A^\infty = A^* \cup A\omega\,.$$

The notions as subwords, prefixes, suffixes can be defined similarly for infinite words too.

The word $v \in A^+$ is a subword of $u \in A\omega$ if there are the words $p \in A^*,\ q \in A\omega$, such that $u = pvq$. If $p \neq \varepsilon$, then p is a prefix of u, whilst q is a suffix of u. Here $F_n(u)$ also represents the set of all subwords of length n of u.

Examples of infinite words over a binary alphabet:

1) The ***power word*** is defined as:

$$p = 010011000111\ldots 0^n1^n\ldots = 010^21^20^31^3\ldots 0^n1^n\ldots\,.$$

It can be seen that

$F_1(p) = \{0,1\},\ F_2(p) = \{01,10,00,11\}$,
$F_3(p) = \{010,100,001,011,110,000,111\},\ldots$

2) The ***Champernowne word*** is obtained by writing in binary representation the natural numbers $0,1,2,3,\ldots$:

$$c = 0\,1\,10\,11\,100\,101\,110\,111\,1000\,1001\,1010\,1011\,1100\,1101\,1110\,1111\,10000\ldots\,.$$

It can be seen that

$F_1(p) = \{0,1\},\ F_2(p) = \{00,01,10,11\}$,
$F_3(p) = \{000,001,010,011,100,101,110,111\},\ldots$

3) The finite ***Fibonacci words*** can be defined recursively as:

$f_0 = 0,\ f_1 = 01$
$f_n = f_{n-1}f_{n-2}$, if $n \geq 2$.

From this definition we obtain:
$f_0 = 0$,
$f_1 = 01$,
$f_2 = 010$,
$f_3 = 01001$,
$f_4 = 01001010$,
$f_5 = 0100101001001$,
$f_6 = 010010100100101001010$.

The infinite Fibonacci word can be defined as the limit of the sequence of finite Fibonacci words:

$$f = \lim_{n\to\infty} f_n \, .$$

The subwords of this word are:
$F_1(f) = \{0, 1\}, \; F_2(f) = \{01, 10, 00\}, \; F_3(f) = \{010, 100, 001, 101\}$,
$F_4(f) = \{0100, 1001, 0010, 0101, 1010\}, \ldots$.

The name of Fibonacci words stems from the Fibonacci numbers, because the length of finite Fibonacci words is related to the Fibonacci numbers: $|f_n| = F_{n+2}$, i.e. the length of the nth finite Fibonacci word f_n is equal to the $(n+2)$th Fibonacci number.

The infinite Fibonacci word has a lot of interesting properties. For example, from the definition, we can see that it cannot contain the subword 11.

The number of 1's in a word u will be denoted by $h(u)$. An infinite word u is ***balanced,*** if for arbitrary subwords x and y of the same length, we have $|h(x) - h(y)| \le 1$, i.e.

$$x, y \in F_n(u) \;\Rightarrow\; |h(x) - h(y)| \le 1 \, .$$

Theorem 26.3 *The infinite Fibonacci word f is balanced.*

Theorem 26.4 *$F_n(f)$ has $n + 1$ elements.*

If word u is concatenated by itself infinitely, then the result is denoted by $u\omega$.

The infinite word u is ***periodic,*** if there is a finite word v, such that $u = v\omega$. This is a generalization of the finite case periodicity. The infinite word u is ***ultimately periodic,*** if there are the words v and w, such that $u = vw\omega$.

The infinite Fibonacci word can be generated by a (homo)morphism too. Let us define this morphism:

$$\chi : A^* \to A^*, \quad \chi(uv) = \chi(u)\chi(v), \quad \forall u, v \in A^* \, .$$

Based on this definition, the function χ can be defined on letters only. A morphism can be extended for infinite words too:

$$\chi : A\omega \to A\omega, \quad \chi(uv) = \chi(u)\chi(v), \quad \forall u \in A^*, v \in A\omega \, .$$

The finite Fibonacci word f_n can be generated by the following morphism:

$$\sigma(0) = 01, \; \sigma(1) = 0 \, .$$

In this case we have the following theorem.

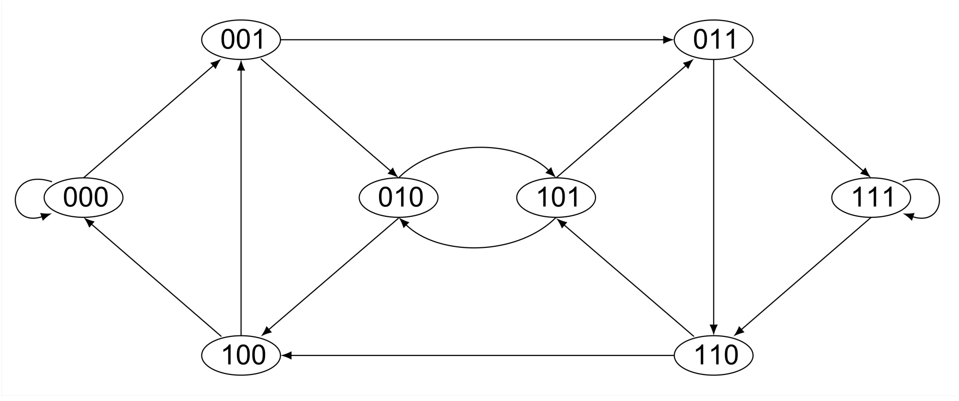


Figure 26.1 The De Bruijn graph $B(2,3)$.

Theorem 26.5 $f_{n+1} = \sigma(f_n)$.

Proof The proof is by induction. Obviously $f_1 = \sigma(f_0)$. Let us presume that $f_k = \sigma(f_{k-1})$ for all $k \leq n$. Because

$$f_{n+1} = f_n f_{n-1},$$

by the induction hypothesis

$$f_{n+1} = \sigma(f_{n-1})\sigma(f_{n-2}) = \sigma(f_{n-1}f_{n-2}) = \sigma(f_n).$$ ■

From this we obtain:

Theorem 26.6 $f_n = \sigma^n(0)$.

The infinite Fibonacci word f is the fixed point of the morphism σ.

$$f = \sigma(f) .$$

26.1.3. Word graphs

Let $V \subseteq A^m$ be a set of words of length m over A, and $E \subseteq AV \cap VA$. We define a digraph, whose vertices are from V, and whose arcs from E. There is an arc from the vertex $a_1a_2\ldots a_m$ to the vertex $b_1b_2\ldots b_m$ if

$$a_2 = b_1,\ \ a_3 = b_2,\ \ldots,\ \ a_m = b_{m-1} \text{ and } a_1a_2\ldots a_mb_m \in E ,$$

that is the last $m-1$ letters in the first word are identical to the first $m-1$ letters in the second word. This arc is labelled by $a_1a_2\ldots a_mb_m$ (or $a_1b_1\ldots b_m$).

De Bruijn graphs If $V = A^m$ and $E = A^{m+1}$, where $A = \{a_1, a_2, \ldots a_n\}$, then the graph is called ***De Bruijn graph,*** denoted by $B(n,m)$.

Figures 26.1 and 26.2 illustrate De Bruijn graphs $B(2,3)$ and $B(3,2)$.

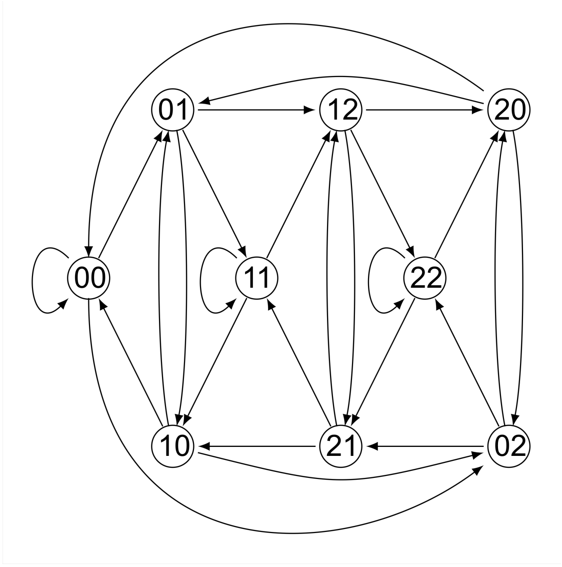


Figure 26.2 The De Bruijn graph $B(3,2)$.

To a walk[1] $x_1x_2\ldots x_m,\quad x_2x_3\ldots x_mx_{m+1},\quad \ldots,\quad z_1z_2\ldots z_m$ in the De Bruijn graph we attach the label $x_1x_2\ldots z_{m-1}z_m$, which is obtained by maximum overlap of the vertices of the walk. In Figure 26.1 in the graph $B(2,3)$ the label attached to the walk $001, 011, 111, 110$ (which is a path) is 001110. The word attached to a Hamiltonian path (which contains all vertices of the graph) in the graph $B(n,m)$ is an (n,m)-type ***De Bruijn word.*** For example, words 0001110100 and 0001011100 are $(2,3)$-type De Bruijn word. An (n,m)-type De Bruijn word contains all words of length m.

A connected digraph[2] is Eulerian[3] if the in-degree of each vertex is equal to its out-degree[4].

Theorem 26.7 *The De Bruijn graph $B(n,m)$ is Eulerian.*

Proof a) The graph is connected because between all pair of vertices $x_1x_2\ldots x_m$ and $z_1z_2\ldots z_m$ there is an oriented path. For vertex $x_1x_2\ldots x_m$ there are n leaving arcs, which enter vertices whose first $m-1$ letters are $x_2x_3\ldots x_m$, and the last letters in this words are all different. Therefore, there is the path $x_1x_2\ldots x_m,\quad x_2x_3\ldots x_mz_1,\quad \ldots,\quad x_mz_1\ldots z_{m-1},\quad z_1z_2\ldots z_m$.

b) There are incoming arcs to vertex $x_1x_2\ldots x_m$ from vertices $yx_1\ldots x_{m-1}$,

[1]In a graph a walk is a sequence of neighbouring edges (or arcs with the same orientation). If the edges or arcs of the walk are all different the walk is called trail, and when all vertices are different, the walk is a path.
[2]A digraph (oriented graph) is connected if between every pair of vertices there is an oriented path at least in a direction.
[3]A digraph is Eulerian if it contains a closed oriented trail with all arcs of the graph.
[4]In-degree (out-degree) of a vertex is the number of arcs which enter (leave) this vertex.

where $y \in A$ (A is the alphabet of the graph, i.e. $V = A^m$). The arcs leaving vertex $x_1x_2\dots x_m$ enter vertices $x_2x_3\dots x_my$, where $y \in A$. Therefore, the graph is Eulerian, because the in-degree and out-degree of each vertex are equal. ■

From this the following theorem is simply obtained.

Theorem 26.8 *An oriented Eulerian trail of the graph $B(n,m)$ (which contains all arcs of graph) is a Hamiltonian path in the graph $B(n,m+1)$, preserving the order.*

For example, in $B(2,2)$ the sequence 000, 001, 010, 101, 011, 111, 110, 100 of arcs is an Eulerian trail. At the same time these words are vertices of a Hamiltonian path in $B(2,3)$.

Algorithm to generate De Bruijn words Generating De Bruijn words is a common task with respectable number of algorithms. We present here the well-known Martin algorithm. Let $A = \{a_1, a_2, \dots, a_n\}$ be an alphabet. Our goal is to generate an (n,m)-type De Bruijn word over the alphabet A.

We begin the algorithm with the word a_1^m, and add at its right end the letter a_k with the greatest possible subscript, such that the suffix of length m of the obtained word does not duplicate a previously occurring subword of length m. Repeat this until such a prolongation is impossible.

When we cannot continue, a De Bruijn word is obtained, with the length $n^m + m - 1$. In the following detailed algorithm, A is the n-letters alphabet, and $B = (b_1, b_2, \dots)$ represents the result, an (n,m)-type De Bruijn word.

```
Martin($A, n, m$)

 1  for $i \leftarrow 1$ to $m$
 2      do $b_i \leftarrow a_1$
 3  $i \leftarrow m$
 4  repeat
 5      $done \leftarrow$ TRUE
 6      $k \leftarrow n$
 7      while $k > 1$
 8            do if $b_{i-m+2}b_{i-m+3}\dots b_ia_k \not\subset b_1b_2\dots b_i$          ▷ Not a subword.
 9                  then $i \leftarrow i + 1$
10                       $b_i \leftarrow a_k$
11                       $done \leftarrow$ FALSE
12                       exit while
13                  else $k \leftarrow k - 1$
14  until $done$
15  return $B$                                        ▷ $B = (b_1b_2, \dots, b_{n^m+m+1})$.
```

Because this algorithm generates all letters of a De Bruijn word of length $(n^m + m-1)$, and n and m are independent, its time complexity is $\Omega(n^m)$. The more precise characterization of the running time depends on the implementation of line 8. The

repeat statement is executed $n^m - 1$ times. The **while** statement is executed at most n times for each step of the **repeat.** The test $b_{i-m+2}b_{i-m+3}\dots b_i a_k \not\subset b_1 b_2 \dots b_i$ can be made in the worst case in mn^m steps. So, the total number of steps is not greater than mn^{2m+1}, resulting a worst case bound $\Theta(n^m+1)$. If we use Knuth-Morris-Pratt string mathching algorithm, then the worst case running time is $\Theta(n^{2m})$.

In chapter **??** a more efficient implementation of the idea of Martin is presented.

Based on this algorithm the following theorem can be stated.

Theorem 26.9 *An (n, m)-type De Bruijn word is the shortest possible among all words containing all words of length m over an alphabet with n letters.*

To generate all (n, m)-type De Bruijn words the following recursive algorithm is given. Here A is also an alphabet with n letters, and B represents an (n, m)-type De Bruijn word. The algorithm is called for each position i with $m+1 \le i \le n^m+m-1$.

All-De-Bruijn(B, i, m)

```
1  for $j \leftarrow 1$ to $n$
2      do $b_i \leftarrow a_j$
3         if $b_{i-m+1}b_{i-m+2}\dots b_i \not\subset b_1 b_2 \dots b_{i-1}$            ▷ Not a subword.
4            then All-De-Bruijn$(b, i+1, m)$
5            else if $length(B) = n^m + m - 1$
6                    then print $B$                                        ▷ A De Bruijn word.
7                         exit for
```

The call of the procedure:

```
for $i = 1$ to $m$
    do $b_i \leftarrow a_1$
All-De-Bruijn $(B, m+1, m)$.
```

This algorithm naturally is exponential.

In following, related to the De Bruijn graphs, the so-called De Bruijn trees will play an important role.

A ***De Bruijn tree*** $T(n, w)$ with the root $w \in A^m$ is a n-ary tree defined recursively as follows:

i. The word w of length m over the alphabet $A = \{a_1, a_2, \dots, a_n\}$ is the root of $T(n, w)$.
ii. If $x_1x_2\dots x_m$ is a leaf in the tree $T(n, w)$, then each word v of the form $x_2x_3\dots x_m a_1$, $x_2x_3\dots x_m a_2$, ..., $x_2x_3\dots x_m a_n$ will be a descendent of $x_1x_2\dots x_m$, if in the path from root to $x_1x_2\dots x_m$ the vertex v does not appears.
iii. The rule *ii* is applied as many as it can.

In Figure 26.3 the De Bruijn tree $T(2, 010)$ is given.

Rauzy graphs If the word u is infinite, and $V = F_n(u)$, $E = F_{n+1}(u)$, then the corresponding word graph is called ***Rauzy graph*** (or ***subword graph***). Figure

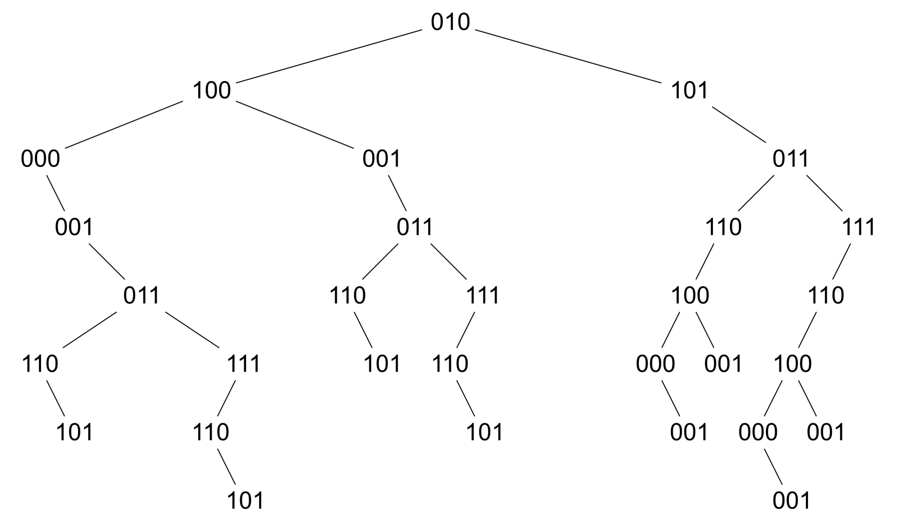


Figure 26.3 The De Bruijn tree $T(2, 010)$.

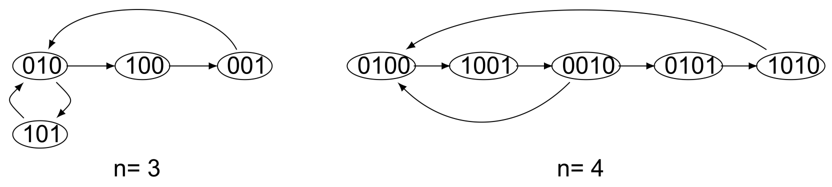


Figure 26.4 Rauzy graphs for the infinite Fibonacci word.

26.4 illustrates the Rauzy graphs of the infinite Fibonacci word for $n = 3$ and $n = 4$. As we have seen, the infinite Fibonacci word is

$$f = 0100101001001010010100100101001001 \ldots ,$$

and $F_1(f) = \{0, 1\}$, $F_2(f) = \{01, 10, 00\}$,
$F_3(f) = \{010, 100, 001, 101\}$, $F_4(f) = \{0100, 1001, 0010, 0101, 1010\}$,
$F_5(f) = \{01001, 10010, 00101, 01010, 10100, 00100\}$.

In the case of the power word $p = 01001100011100001111 \ldots 0^n 1^n \ldots$, where $F_1(p) = \{0, 1\}$, $F_2(p) = \{01, 10, 00, 11\}$,
$F_3(p) = \{010, 100, 000, 001, 011, 111, 110\}$,
$F_4(p) = \{0100, 1001, 0011, 0110, 1100, 1000, 0000, 0001, 0111, 1110, 1111\}$,
the corresponding Rauzy graphs are given in Figure 26.5.

As we can see in Fig, 26.4 and 26.5 there are subwords of length n which can be

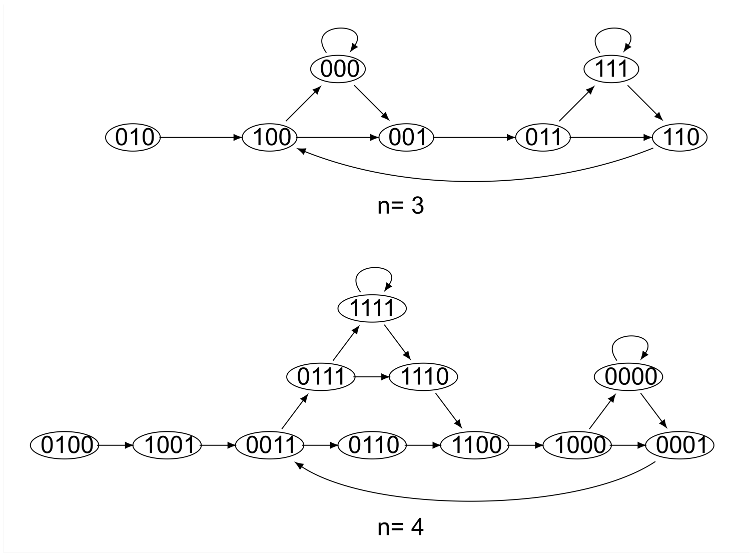


Figure 26.5 Rauzy graphs for the power word.

continued only in a single way (by adding a letter), and there are subwords which can be continued in two different ways (by adding two different letters). These latter subwords are called ***special subwords.*** A subword $v \in F_n(u)$ is a ***right special subword,*** if there are at least two different letters $a \in A$, such that $va \in F_{n+1}(u)$. Similarly, $v \in F_n(u)$ is ***left special subword,*** if there are at least two different letters $a \in A$, such that $av \in F_{n+1}(u)$. A subword is ***bispecial,*** if at the same time is right and left special. For example, the special subwords in Figures 26.4 and 26.5) are:

left special subwords:	010, 0100 (Figure 26.4),
	110, 000, 111, 1110, 0001, 1111, 0011 (Figure 26.5),
right special subwords::	010, 0010 (Figure 26.4),
	011, 000, 111, 0111, 1111, 0011 (Figure 26.5)
bispecial subwords:	010 (Figure 26.4),
	000, 111, 1111, 0011 (Figure 26.5).

26.1.4. Complexity of words

The complexity of words measures the diversity of the subwords of a word. In this regard the word *aaaaa* has smaller complexity then the word *abcab*.

We define the following complexities for a word.

1) The ***subword complexity*** or simply the ***complexity*** of a word assigns to each $n \in \mathbf{N}$ the number of different subwords of length n. For a word u the number

of different subwords of length n is denoted by $f_u(n)$.

$$f_u(n) = \#F_n(u), \quad u \in A^\infty .$$

If the word is finite, then $f_u(n) = 0$, if $n > |u|$.

2) The ***maximal complexity*** is considered only for finite words.

$$C(u) = \max\{f_u(n) \mid n \ge 1, u \in A^*\} .$$

If u is an infinite word, then $C_u^-(n)$ is the ***lower maximal complexity,*** respectively $C_u^+(n)$ the ***upper maximal complexity.***

$$C_u^-(n) = \min_i C(u_i u_{i+1} \dots u_{i+n-1}), \quad C_u^+(n) = \max_i C(u_i u_{i+1} \dots u_{i+n-1}) .$$

3) The ***global maximal complexity*** is defined on the set A^n:

$$G(n) = \max\{C(u) \mid u \in A^n\} .$$

4) The ***total complexity*** for a finite word is the number of all different nonempty subwords[5]

$$K(u) = \sum_{i=1}^{|u|} f_u(i), \quad u \in A^* .$$

For an infinite word $K_u^-(n)$ is the ***lower total complexity,*** and $K_u^+(n)$ is the ***upper total complexity:***

$$K_u^-(n) = \min_i K(u_i u_{i+1} \dots u_{i+n-1}), \quad K_u^+(n) = \max_i K(u_i u_{i+1} \dots u_{i+n-1}) .$$

5) A decomposition $u = u_1 u_2 \dots u_k$ is called a ***factorization*** of u. If each u_i (with the possible exception of u_k) is the shortest prefix of $u_i u_{i+1} \dots u_k$ which does not occur before in u, then this factorization is called the Lempel-Ziv factorization. The number of subwords u_i in such a factorization is the ***Lempel-Ziv factorization complexity*** of u. For example for the word $u = ababaaabb$ the Lempel-Ziv factorization is: $u = a.b.abaa.abb$. So, the Lempel-Ziv factorization complexity of u is $\textsc{lz}(u) = 4$.

6) If in a factorization $u = u_1 u_2 \dots u_k$ each u_i is the longest possible palindrome, then the factorization is called a palindromic factorization, and the number of subwords u_i in this is the ***palindromic factorization complexity.*** For $u = aababbabbabb = aa.babbabbab.b$, so the palindromic factorization complexity of u is $\textsc{pal}(u) = 3$.

7) The ***window complexity*** P_w is defined for infinite words only. For $u = u_0 u_1 u_2 \dots u_n \dots$ the window complexity is

$$P_w(u, n) = \#\{u_{kn} u_{kn+1} \dots u_{(k+1)n-1} \,\big|\, k \ge 0\} .$$

[5]Sometimes the empty subword is considered too. In this case the value of total complexity is increased by 1.

Subword complexity As we have seen

$$f_u(n) = \#F_n(u), \quad \forall u \in A^\infty,\ n \in \mathbf{N}\,.$$

$f_u(n) = 0$, if $n > |u|$.

For example, in the case of $u = abacab$:

$$f_u(1) = 3,\ f_u(2) = 4,\ f_u(3) = 4,\ f_u(4) = 3, f_u(5) = 2,\ f_u(6) = 1\,.$$

In Theorem 26.4 was stated that for the infinite Fibonacci word:

$$f_f(n) = n + 1\,.$$

In the case of the power word $p = 010011\ldots 0^k 1^k \ldots$ the complexity is:

$$f_p(n) = \frac{n(n+1)}{2} + 1\,.$$

This can be proved if we determine the difference $f_p(n+1) - f_p(n)$, which is equal to the number of words of length n which can be continued in two different ways to obtain words of length $n+1$. In two different ways can be extended only the words of the form $0^k 1^{n-k}$ (it can be followed by 1 always, and by 0 when $k \le n-k$) and $1^k 0^{n-k}$ (it can be followed by 0 always, and by 1 when $k < n-k$). Considering separately the cases when n is odd and even, we can see that:

$$f_p(n+1) - f_p(n) = n + 1\,,$$

and from this we get

$$\begin{aligned} f_p(n) &= n + f_p(n-1) = n + (n-1) + f_p(n-2) = \ldots \\ &= n + (n-1) + \ldots + 2 + f_p(1) = \frac{n(n+1)}{2} + 1\,. \end{aligned}$$

In the case of the ***Champernowne word***

$$\begin{aligned} c = u_0 u_1 \ldots u_n \ldots &= 0\,1\,10\,11\,100\,101\,110\,111\,1000 \ldots \\ &= 0110111001011101111000\ldots, \end{aligned}$$

the complexity is $f_c(n) = 2^n$.

Theorem 26.10 *If for the infinite word $u \in A\omega$ there exists an $n \in \mathbf{N}$ such that $f_u(n) \le n$, then u is ultimately periodic.*

Proof $f_u(1) \ge 2$, otherwise the word is trivial (contains just equal letters). Therefore there is a $k \le n$, such that $f_u(k) = f_u(k+1)$. But

$$f_u(k+1) - f_u(k) = \sum_{v \in F_k(u)} \Big(\#\big\{a \in A \mid va \in F_{k+1}(u)\big\} - 1 \Big)\,.$$

It follows that each subword $v \in F_k(u)$ has only one extension to obtain $va \in$

$F_{k+1}(u)$. So, if $v = u_i u_{i+1} \ldots u_{i+k-1} = u_j u_{j+1} \ldots u_{j+k-1}$, then $u_{i+k} = u_{j+k}$. Because $F_k(u)$ is a finite set, and u is infinite, there are i and j $(i < j)$, for which $u_i u_{i+1} \ldots u_{i+k-1} = u_j u_{j+1} \ldots u_{j+k-1}$, but in this case $u_{i+k} = u_{j+k}$ is true too. Then from $u_{i+1} u_{i+2} \ldots u_{i+k} = u_{j+1} u_{j+2} \ldots u_{j+k}$ we obtain the following equality results: $u_{i+k+1} = u_{j+k+1}$, therefore $u_{i+l} = u_{j+l}$ is true for all $l \geq 0$ values. Therefore u is ultimately periodic. ■

A word $u \in A\omega$ is ***Sturmian,*** if $f_u(n) = n + 1$ for all $n \geq 1$.

Sturmian words are the least complexity infinite and non periodic words. The infinite Fibonacci word is Sturmian. Because $f_u(1) = 2$, the Sturmian words are two-letters words.

From the Theorem 26.10 it follows that each infinite and not ultimately periodic word has complexity at least $n + 1$, i.e.

$$u \in A\omega, u \text{ not ultimately periodic } \Rightarrow f_u(n) \geq n + 1\,.$$

The equality holds for Sturmian words.

Infinite words can be characterized using the lower and upper total complexity too.

Theorem 26.11 *If an infinite word u is not ultimately periodic and $n \geq 1$, then*

$$C_u^+(n) \geq \left[\frac{n}{2}\right] + 1, \quad K_u^+(n) \geq \left[\frac{n^2}{4} + n\right].$$

For the Sturmian words equality holds.

Let us denote by $\{x\}$ the fractional part of x, and by $\lfloor x \rfloor$ its integer part. Obviously $x = \lfloor x \rfloor + \{x\}$. The composition of a function R by itself n times will be denoted by R^n. So $R^n = R \circ R \circ \ldots \circ R$ (n times). Sturmian words can be characterized in the following way too:

Theorem 26.12 *A word $u = u_1 u_2 \ldots$ is Sturmian if and only if there exists an irrational number α and a real number z, such that for $R(x) = \{x + \alpha\}$*

$$u_n = \begin{cases} 0, & \text{if } R^n(z) \in (0, 1-\alpha)\,, \\ 1, & \text{if } R^n(z) \in [1-\alpha, 1)\,, \end{cases}$$

or

$$u_n = \begin{cases} 1, & \text{if } R^n(z) \in (0, 1-\alpha)\,, \\ 0, & \text{if } R^n(z) \in [1-\alpha, 1)\,. \end{cases}$$

In the case of the infinite Fibonacci number, these numbers are: $\alpha = z = (\sqrt{5}+1)/2$.

Sturmian words can be generated by the orbit of a billiard ball inside a square too. A billiard ball is launched under an irrational angle from a boundary point of the square. If we consider an endless move of the ball with reflection on boundaries and without friction, an infinite trajectory will result. We put an 0 in the word if the ball reaches a horizontal boundary, and 1 when it reaches a vertical one. In such a way we generate an infinite word. This can be generalized using an $(s + 1)$-letter

u	$f_u(1)$	$f_u(2)$	$f_u(3)$	$f_u(4)$	$f_u(5)$	$f_u(6)$	$f_u(7)$	$f_u(8)$
00100011	2	4	5	5	4	3	2	1
00100100	2	3	3	3	3	3	2	1
00100101	2	3	4	4	4	3	2	1
00100110	2	4	5	5	4	3	2	1
00100111	2	4	5	5	4	3	2	1
00101000	2	3	5	5	4	3	2	1
00101001	2	3	4	5	4	3	2	1
00101011	2	4	4	4	4	3	2	1
01010101	2	2	2	2	2	2	2	1
11111111	1	1	1	1	1	1	1	1

Figure 26.6 Complexity of several binary words.

alphabet and an $(s+1)$-dimensional hypercube. In this case the complexity is

$$f_u(n, s+1) = \sum_{i=0}^{\min(n,s)} \frac{n!s!}{(n-i)!i!(s-i)!} \,.$$

If $s = 1$, then $f_u(n,2) = f_u(n) = n+1$, and if $s = 2$, then $f_u(n,3) = n^2+n+1$.

Maximal complexity For a finite word u

$$C(u) = \max\{f_u(n) \mid n \geq 1\}$$

is the maximal complexity. In Figure 26.6 the values of the complexity function for several words are given for all possible length. From this, we can see for example that $C(00100011) = 5$, $C(00100100) = 3$ etc.

For the complexity of finite words the following interesting result is true.

Theorem 26.13 *If w is a finite word, $f_w(n)$ is its complexity, then there are the natural numbers m_1 and m_2 with $1 \leq m_1 \leq m_2 \leq |w|$ such that*

- $f_w(n+1) > f_w(n)$, *for* $1 \leq n < m_1$,
- $f_w(n+1) = f_w(n)$, *for* $m_1 \leq n < m_2$,
- $f_w(n+1) = f_w(n) - 1$, *for* $m_2 \leq n \leq |w|$.

From the Figure 26.6, for example, if
$w = 00100011$, then $m_1 = 3$, $m_2 = 4$,
$w = 00101001$, then $m_1 = 4$, $m_2 = 4$,
$w = 00101011$, then $m_1 = 2$, $m_2 = 5$.

Global maximal complexity The global maximal complexity is

$$G(n) = \max\{C(u) \mid u \in A^n\}\,,$$

u	$f_u(i)$ $i=1$	$i=2$	$i=3$
000	1	1	1
001	2	2	1
010	2	2	1
011	2	2	1
100	2	2	1
101	2	2	1
110	2	2	1
111	1	1	1

Figure 26.7 Complexity of all 3-length binary words

that is the greatest (maximal) complexity in the set of all words of length n on a given alphabet. The following problems arise:

• what is length of the subwords for which the global maximal complexity is equal to the maximal complexity?

• how many such words exist?

Example 26.1 For the alphabet $A = \{0, 1\}$ the Figure 26.7 and 26.8 contain the complexity of all 3-length and 4-length words.

In this case of the 3-length words (Figure 26.7) the global maximal complexity is 2, and this value is obtained for 1-length and 2-length subwords. There are 6 such words.

For 4-length words (Figure 26.8) the global maximal complexity is 3, and this value is obtained for 2-length words. The number of such words is 8.

To solve the above two problems, the following notations will be used:

$$R(n) = \{i \in \{1, 2, \dots, n\} \mid \exists u \in A^n : f_u(i) = G(n)\},$$

$$M(n) = \#\{u \in A^n : C(u) = G(n)\}.$$

In the table of Figure 26.9 values of $G(n)$, $R(n)$, $M(n)$ are given for length up to 20 over on a binary alphabet.

We shall use the following result to prove some theorems on maximal complexity.

Lemma 26.14 *For each $k \in \mathbf{N}^*$, the shortest word containing all the q^k words of length k over an alphabet with q letters has $q^k + k - 1$ letters (hence in this word each of the q^k words of length k appears only once).*

Theorem 26.15 *If $\#A = q$ and $q^k + k \le n \le q^{k+1} + k$, then $G(n) = n - k$.*

Proof Let us consider at first the case $n = q^{k+1} + k$, $k \ge 1$.

From Lemma 26.14 we obtain the existence of a word w of length $q^{k+1} + k$ which contains all the q^{k+1} words of length $k+1$, hence $f_w(k+1) = q^{k+1}$. It is obvious that $f_w(l) = q^l < f_w(k+1)$ for $l \in \{1, 2, \dots, k\}$ and $f_w(k+1+j) = q^{k+1} - j < f_w(k+1)$ for $j \in \{1, 2, \dots q^{k+1} - 1\}$. Any other word of length $q^{k+1} + k$ will have the maximal

u	$f_u(i)$ $i=1$	$i=2$	$i=3$	$i=4$
0000	1	1	1	1
0001	2	2	2	1
0010	2	3	2	1
0011	2	3	2	1
0100	2	3	2	1
0101	2	2	2	1
0110	2	3	2	1
0111	2	2	2	1
1000	2	2	2	1
1001	2	3	2	1
1010	2	2	2	1
1011	2	3	2	1
1100	2	3	2	1
1101	2	3	2	1
1110	2	2	2	1
1111	1	1	1	1

Figure 26.8 Complexity of all 4-length binary words.

complexity less than or equal to $C(w) = f_w(k+1)$, hence we have $G(n) = q^{k+1} = n - k$.

For $k \geq 1$ we consider now the values of n of the form $n = q^{k+1} + k - r$ with $r \in \{1, 2, \ldots, q^{k+1} - q^k\}$, hence $q^k + k \leq n < q^{k+1} + k$. If from the word w of length $q^{k+1} + k$ considered above we delete the last r letters, we obtain a word w_n of length $n = q^{k+1} + k - r$ with $r \in \{1, 2, \ldots, q^{k+1} - q^k\}$. This word will have $f_{w_n}(k+1) = q^{k+1} - r$ and this value will be its maximal complexity. Indeed, it is obvious that $f_{w_n}(k+1+j) = f_{w_n}(k+1) - j < f_{w_n}(k+1)$ for $j \in \{1, 2, \ldots, n-k-1\}$; for $l \in \{1, 2, \ldots, k\}$ it follows that $f_{w_n}(l) \leq q^l \leq q^k \leq q^{k+1} - r = f_{w_n}(k+1)$, hence $C(w_n) = f_{w_n}(k+1) = q^{k+1} - r$. Because it is not possible for a word of length $n = q^{k+1} + k - r$, with $r \in \{1, 2, \ldots, q^{k+1} - q^k\}$ to have the maximal complexity greater than $q^{k+1} - r$, it follows that $G(n) = q^{k+1} - r = n - k$. ■

Theorem 26.16 *If $\#A = q$ and $q^k + k < n < q^{k+1} + k + 1$ then $R(n) = \{k+1\}$; if $n = q^k + k$ then $R(n) = \{k, k+1\}$.*

Proof In the first part of the proof of Theorem 26.15, we proved for $n = q^{k+1} + k$, $k \geq 1$, the existence of a word w of length n for which $G(n) = f_w(k+1) = n - k$. This means that $k + 1 \in R(n)$. For the word w, as well as for any other word w' of length n, we have $f_{w'}(l) < f_w(k+1)$, $l \neq k+1$, because of the special construction of w, which contains all the words of length $k+1$ in the most compact way. It follows that $R(n) = \{k+1\}$.

As in the second part of the proof of Theorem 26.15, we consider $n = q^{k+1} + k - r$

n	$G(n)$	$R(n)$	$M(n)$
1	1	1	2
2	2	1	2
3	2	1, 2	6
4	3	2	8
5	4	2	4
6	4	2, 3	36
7	5	3	42
8	6	3	48
9	7	3	40
10	8	3	16
11	8	3, 4	558
12	9	4	718
13	10	4	854
14	11	4	920
15	12	4	956
16	13	4	960
17	14	4	912
18	15	4	704
19	16	4	256
20	16	4, 5	79006

Figure 26.9 Values of $G(n)$, $R(n)$, and $M(n)$.

with $r \in \{1, 2, \dots q^{k+1} - q^k\}$ and the word w_n for which $G(n) = f_{w_n}(k+1) = q^{k+1} - r$. We have again $k+1 \in R(n)$. For $l > k+1$, it is obvious that the complexity function of w_n, or of any other word of length n, is strictly less than $f_{w_n}(k+1)$. We examine now the possibility of finding a word w with $f_w(k+1) = n-k$ for which $f_w(l) = n-k$ for $l \le k$. We have $f_w(l) \le q^l \le q^k \le q^{k+1} - r$, hence the equality $f_w(l) = n-k = q^{k+1} - r$ holds only for $l = k$ and $r = q^{k+1} - q^k$, that is for $w = q^k + k$.

We show that for $n = q^k + k$ we have indeed $R(n) = \{k, k+1\}$. If we start with the word of length $q^k + k - 1$ generated by the Martin's algorithm (or with another De Bruijn word) and add to this any letter from A, we obtain obviously a word v of length $n = q^k + k$, which contains all the q^k words of length k and $q^k = n - k$ words of length $k+1$, hence $f_v(k) = f_v(k+1) = G(n)$. ■

Having in mind the Martin algorithm (or other more efficient algorithms), words w with maximal complexity $C(w) = G(n)$ can be easily constructed for each n and for both situations in Theorem 26.16.

Theorem 26.17 *If $\#A = q$ and $q^k + k \le n \le q^{k+1} + k$ then $M(n)$ is equal to the number of different paths of length $n - k - 1$ in the de Bruijn graph $B(q, k+1)$.*

Proof From Theorems 26.15 and 26.16 it follows that the number $M(n)$ of the words of length n with global maximal complexity is given by the number of words $w \in A^n$ with $f_w(k+1) = n-k$. It means that these words contain $n-k$ subwords of length $k+1$, all of them distinct. To enumerate all of them we start successively with each word of $k+1$ letters (hence with each vertex in $B(q, k+1)$) and we add at each step, in turn, one of the symbols from A which does not duplicate a word of length $k+1$ which has already appeared. Of course, not all of the trials will finish in a word of length n, but those which do this, are precisely paths in $B(q, k+1)$ starting with each vertex in turn and having the length $n-k-1$. Hence to each word of length n with $f_w(k+1) = n-k$ we can associate a path and only one of length $n-k-1$ starting from the vertex given by the first $k+1$ letters of the initial word; conversely, any path of length $n-k-1$ will provide a word w of length n which contains $n-k$ distinct subwords of length $k+1$. ■

$M(n)$ can be expressed also as the number of vertices at level $n-k-1$ in the set $\left\{T(q,w) \,\middle|\, w \in A^{k+1}\right\}$ of De Bruijn trees.

Theorem 26.18 *If $n = 2^k + k - 1$, then $M(n) = 2^{2^{k-1}}$.*

Proof In the De Bruijn graph $B(2,k)$ there are $2^{2^{k-1}-k}$ different Hamiltonian cycles. With each vertex of a Hamiltonian cycle a De Bruijn word begins (containing all k-length subwords), which has maximal complexity, so $M(n) = 2^k \cdot 2^{2^{k-1}-k} = 2^{2^{k-1}}$, which proves the theorem. ■

A generalization for an alphabet with $q \geq 2$ letters:

Theorem 26.19 *If $n = q^k + k - 1$, then $M(n) = (q!)^{q^{k-1}}$.*

Total complexity The ***total complexity*** is the number of different nonempty subwords of a given word:

$$K(u) = \sum_{i=1}^{|u|} f_u(i)\,.$$

The total complexity of a trivial word of length n (of the form a^n, $n \geq 1$) is equal to n. The total complexity of a rainbow word (with pairwise different letters) of length n is equal to $\dfrac{n(n+1)}{2}$.

The problem of existence of words with a given total complexity are studied in the following theorems.

Theorem 26.20 *If C is a natural number different from 1, 2 and 4, then there exists a nontrivial word of total complexity equal to C.*

Proof To prove this theorem we give the total complexity of the following k-length words:

$$\begin{aligned} K(a^{k-1}b) &= 2k-1, &\text{for } k \geq 1\,,\\ K(ab^{k-3}aa) &= 4k-8, &\text{for } k \geq 4\,,\\ K(abcd^{k-3}) &= 4k-6, &\text{for } k \geq 3\,. \end{aligned}$$

These can be proved immediately from the definition of the total complexity.

1. If C is odd then we can write $C = 2k-1$ for a given k. It follows that $k = (C+1)/2$, and the word $a^{k-1}b$ has total complexity C.

2. If C is even, then $C = 2\ell$.

2.1. If $\ell = 2h$, then $4k-8 = C$ gives $4k-8 = 4h$, and from this $k = h+2$ results. The word $ab^{k-3}aa$ has total complexity C.

2.2. If $\ell = 2h+1$ then $4k-6 = C$ gives $4k-6 = 4h+2$, and from this $k = h+2$ results. The word $abcd^{k-3}$ has total complexity C. ■

In the proof we have used more than two letters in a word only in the case of the numbers of the form $4h+2$ (case 2.2 above). The new question is, if there exist always nontrivial words formed only of two letters with a given total complexity. The answer is yes anew. We must prove this only for the numbers of the form $4h+2$. If $C = 4h+2$ and $C \geq 34$, we use the followings:

$$\begin{aligned} K(ab^{k-7}abbabb) &= 8k-46, &\text{for } k \geq 10\,,\\ K(ab^{k-7}ababba) &= 8k-42, &\text{for } k \geq 10\,. \end{aligned}$$

If $h = 2s$, then $8k-46 = 4h+2$ gives $k = s+6$, and the word $ab^{k-7}abbabb$ has total complexity $4h+2$.

If $h = 2s+1$, then $8k-42 = 4h+2$ gives $k = s+6$, and the word $ab^{k-7}ababba$ has total complexity $4h+2$. For $C < 34$ only 14, 26 and 30 are feasible. The word ab^4a has total complexity 14, ab^6a has 26, and ab^5aba 30. Easily it can be proved, using a tree, that for 6, 10, 18 and 22 such words does not exist. Then the following theorem is true.

Theorem 26.21 *If C is a natural number different from 1, 2, 4, 6, 10, 18 and 22, then there exists a nontrivial word formed only of two letters, with the given total complexity C.*

The existence of a word with a given length and total complexity is not always assured, as we will prove in what follows.

In relation with the second problem a new one arises: How many words of length n and complexity C there exist? For small n this problem can be studied exhaustively. Let A be of n letters, and let us consider all words of length n over A. By a computer program we have got Figure 26.10, which contains the frequency of words with given length and total complexity.

Let $|A| = n$ and let $\phi_n(C)$ denote the frequency of the words of length n over A having a complexity C. Then we have the following easy to prove results:

$n = 2$

C	2	3
$\phi_n(C)$	2	2

$n = 3$

C	3	4	5	6
$\phi_n(C)$	3	0	18	6

$n = 4$

C	4	5	6	7	8	9	10
$\phi_n(C)$	4	0	0	36	48	144	24

$n = 5$

C	5	6	7	8	9	10	11	12	13	14	15
$\phi_n(C)$	5	0	0	0	60	0	200	400	1140	1200	120

$n = 6$

C	6	7	8	9	10	11	12	13
$\phi_n(C)$	6	0	0	0	0	90	0	0
C	14	15	16	17	18	19	20	21
$\phi_n(C)$	300	990	270	5400	8280	19800	10800	720

Figure 26.10 Frequency of words with given total complexity

$$\phi_n(C) = 0, \qquad \text{if } C < n \text{ or } C > \frac{n(n+1)}{2},$$
$$\phi_n(n) = n,$$
$$\phi_n(2n-1) = 3n(n-1),$$
$$\phi_n\left(\frac{n(n+1)}{2} - 1\right) = \frac{n(n-1)n!}{2},$$
$$\phi_n\left(\frac{n(n+1)}{2}\right) = n!$$

As regards the distribution of the frequency 0, the following are true:

If $C = n+1, n+2, \ldots, 2n-2$, then $\phi_n(C) = 0$.
If $C = 2n, 2n+1, \ldots, 3n-5$, then $\phi_n(C) = 0$.

The question is, if there exists a value from which up to $\frac{n(n+1)}{2}$ no more 0 frequency exist. The answer is positive. Let us denote by b_n the least number between n and $n(n+1)/2$ for which

$$\phi_n(C) \neq 0 \quad \text{for all } C \text{ with} \quad b_n \le C \le \frac{n(n+1)}{2}.$$

The number b_n exists for any n (in the worst case it may be equal to $n(n+1)/2$):

Theorem 26.22 *If $\ell \ge 2$, $0 \le i \le \ell$, $n = \dfrac{\ell(\ell+1)}{2} + 2 + i$, then*

$$b_n = \frac{\ell(\ell^2-1)}{2} + 3\ell + 2 + i(\ell+1).$$

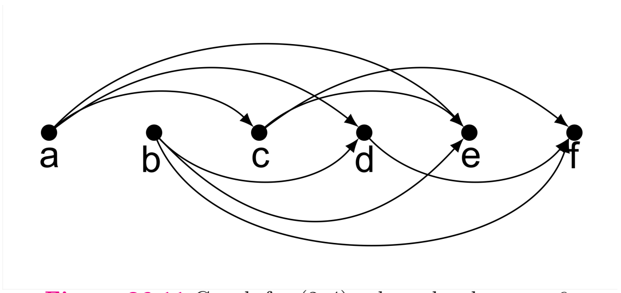


Figure 26.11 Graph for $(2,4)$-subwords when $n = 6$.

26.2. Generalized complexity measures

As we have seen in the previous section, a contiguous part of a word (obtained by erasing a prefix or/and a suffix) is a ***subword*** or ***factor.*** If we eliminate arbitrary letters from a word, what is obtained is a ***scattered subword,*** sometimes called ***subsequence.*** Special scattered subwords, in which the consecutive letters are at distance at least d_1 and at most d_2 in the original word, are called (d_1, d_2)-***subwords***. More formally we give the following definition.

Let n, $d_1 \le d_2$, s be positive integers, and let $u = x_1x_2 \ldots x_n \in A^n$ be a word over the alphabet A. The word $v = x_{i_1}x_{i_2} \ldots x_{i_s}$, where

$i_1 \ge 1$,
$d_1 \le i_{j+1} - i_j \le d_2$, for $j = 1, 2, \ldots, s-1$,
$i_s \le n$,

is a (d_1, d_2)-***subword*** of length s of u.
For example the $(2,4)$-subwords of *aabcade* are: *a*, *ab*, *ac*, *aba*, *aa*, *acd*, *abd*, *aae*, *abae*, *ace*, *abe*, *ad*, *b*, *ba*, *bd*, *bae*, *be*, *c*, *cd*, *ce*, *ae*, *d*, *e*.

The number of different (d_1, d_2)-subwords of a word u is called (d_1, d_2)-***complexity*** and is denoted by $C_u(d_1, d_2)$.
For example, if $u = aabcade$, then $C_u(2,4) = 23$.

26.2.1. Rainbow words

Words with pairwise different letters are called ***rainbow words.*** The (d_1, d_2)-complexity of a rainbow word of length n does not depends on what letters it contains, and is denoted by $C(n; d_1, d_2)$.

To compute the (d_1, d_2)-complexity of a rainbow word of length n, let us consider the word $a_1a_2 \ldots a_n$ (if $i \ne j$, then $a_i \ne a_j$) and the corresponding digraph $G = (V, E)$, with

$V = \{a_1, a_2, \ldots, a_n\}$,
$E = \{(a_i, a_j) \mid d_1 \le j - i \le d_2,\ i = 1, 2, \ldots, n, j = 1, 2, \ldots, n\}$.

For $n = 6, d_1 = 2, d_2 = 4$ see Figure 26.11.
The adjacency matrix $A = \left(a_{ij}\right)_{\substack{i=\overline{1,n}\\ j=\overline{1,n}}}$ of the graph is defined by:

$$a_{ij} = \begin{cases} 1, & \text{if } d_1 \le j - i \le d_2, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for } i = 1, 2, \ldots, n, j = 1, 2, \ldots, n.$$

Because the graph has no directed cycles, the entry in row i and column j in A^k (where $A^k = A^{k-1}A$, with $A^1 = A$) will represent the number of k-length directed paths from a_i to a_j. If A^0 is the identity matrix (with entries equal to 1 only on the first diagonal, and 0 otherwise), let us define the matrix $R = (r_{ij})$:

$$R = A^0 + A + A^2 + \cdots + A^k, \text{ where } A^{k+1} = O \,(\text{the null matrix}).$$

The (d_1, d_2)-complexity of a rainbow word is then

$$C(n; d_1, d_2) = \sum_{i=1}^{n}\sum_{j=1}^{n} r_{ij}.$$

The matrix R can be better computed using a variant of the well-known Warshall algorithm:

```
WARSHALL($A, n$)
1  $W \leftarrow A$
2  for $k \leftarrow 1$ to $n$
3      do for $i \leftarrow 1$ to $n$
4             do for $j \leftarrow 1$ to $n$
5                    do $w_{ij} \leftarrow w_{ij} + w_{ik}w_{kj}$
6  return $W$
```

From W we obtain easily $R = A^0 + W$. The time complexity of this algorithms is $\Theta(n^3)$.

For example let us consider the graph in Figure 26.11. The corresponding adjacency matrix is:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

After applying the Warshall algorithm:

$$W = \begin{pmatrix} 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad R = \begin{pmatrix} 1 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

and then $C(6; 2, 4) = 19$, the sum of entries in R.

The Warshall algorithm combined with the Latin square method can be used to obtain all nontrivial (with length at least 2) (d_1, d_2)-subwords of a given rainbow word $a_1a_2 \ldots a_n$ of length n. Let us consider a matrix $\mathcal{A}$ with the elements A_{ij} which

$n = 6$

$d_1 \backslash d_2$	1	2	3	4	5
1	21	46	58	62	63
2	-	12	17	19	20
3	-	-	9	11	12
4	-	-	-	8	9
5	-	-	-	-	7

$n = 7$

$d_1 \backslash d_2$	1	2	3	4	5	6
1	28	79	110	122	126	127
2	-	16	25	30	32	33
3	-	-	12	15	17	18
4	-	-	-	10	12	13
5	-	-	-	-	9	10
6	-	-	-	-	-	8

Figure 26.12 (d_1, d_2)-complexity for rainbow words of length 6 and 7.

are set of words. Initially this matrix is defined as

$$A_{ij} = \begin{cases} \{a_i a_j\}, & \text{if } d_1 \le j - i \le d_2, \\ \emptyset, & \text{otherwise,} \end{cases} \qquad \text{for } i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, n\,.$$

If $\mathcal{A}$ and $\mathcal{B}$ are sets of words, $\mathcal{AB}$ will be formed by the set of concatenation of each word from $\mathcal{A}$ with each word from $\mathcal{B}$:

$$\mathcal{AB} = \left\{ab \,\middle|\, a \in \mathcal{A}, b \in \mathcal{B}\right\}.$$

If $s = s_1 s_2 \ldots s_p$ is a word, let us denote by $'s$ the word obtained from s by erasing the first character: $'s = s_2 s_3 \ldots s_p$. Let us denote by $'A_{ij}$ the set A_{ij} in which we erase from each element the first character. In this case $'\mathcal{A}$ is a matrix with entries $'A_{ij}$.

Starting with the matrix $\mathcal{A}$ defined as before, the algorithm to obtain all nontrivial (d_1, d_2)-subwords is the following:

```
WARSHALL-LATIN($\mathcal{A}, n$)

1  $\mathcal{W} \leftarrow \mathcal{A}$
2  for $k \leftarrow 1$ to $n$
3      do for $i \leftarrow 1$ to $n$
4             do for $j \leftarrow 1$ to $n$
5                    do if $W_{ik} \ne \emptyset$ and $W_{kj} \ne \emptyset$
6                          then $W_{ij} \leftarrow W_{ij} \cup W_{ik}\,'W_{kj}$
7  return $\mathcal{W}$
```

The set of nontrivial (d_1, d_2)-subwords is $\bigcup_{i,j \in \{1,2,\ldots,n\}} W_{ij}$. The time complexity is also $\Theta(n^3)$.

For $n = 7$, $d_1 = 2$, $d_2 = 4$, the initial matrix is:

$$\mathcal{A} = \begin{pmatrix} \emptyset & \emptyset & \{ac\} & \{ad\} & \{ae\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{bd\} & \{be\} & \{bf\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \{ce\} & \{cf\} & \{cg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{df\} & \{dg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{eg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix},$$

and

$$\mathcal{W} = \begin{pmatrix} \emptyset & \emptyset & \{ac\} & \{ad\} & \{ace, ae\} & \{adf, acf\} & \{aeg, aceg, adg, acg\} \\ \emptyset & \emptyset & \emptyset & \{bd\} & \{be\} & \{bdf, bf\} & \{beg, bdg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \{ce\} & \{cf\} & \{ceg, cg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{df\} & \{dg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{eg\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}.$$

Counting the one-letter subwords too, we obtain $C(7; 2, 4) = 30$.

The case $d_1 = 1$ In this case instead of d_2 we will use d. For a rainbow word, $a_{i,d}$ we will denote the number of $(1, d)$-subwords which finish at the position i. For $i = 1, 2, \dots, n$

$$a_{i,d} = 1 + a_{i-1,d} + a_{i-2,d} + \ldots + a_{i-d,d}\,. \tag{26.1}$$

For simplicity, let us denote $C(n; 1, d)$ by $N(n, d)$. The $(1, d)$-complexity of a rainbow word can be obtained by the formula

$$N(n, d) = \sum_{i=1}^{n} a_{i,d}\,.$$

Because of (26.1) we can write in the case of $d \geq 2$

$$a_{i,d} + \frac{1}{d-1} = \left(a_{i-1,d} + \frac{1}{d-1}\right) + \cdots + \left(a_{i-d,d} + \frac{1}{d-1}\right).$$

Denoting

$$b_{i,d} = a_{i,d} + \frac{1}{d-1}, \quad \text{and} \quad c_{i,d} = (d-1)b_{i,d}\,,$$

we get

$$c_{i,d} = c_{i-1,d} + c_{i-2,d} + \ldots + c_{i-d,d}\,,$$

and the sequence $c_{i,d}$ is one of Fibonacci-type. For any d we have $a_{1,d} = 1$ and from this $c_{1,d} = d$ results. Therefore the numbers $c_{i,d}$ are defined by the following recurrence equations:

$$\begin{aligned} c_{n,d} &= c_{n-1,d} + c_{n-2,d} + \ldots + c_{n-d,d}, && \text{for} \quad n > 0, \\ c_{n,d} &= 1, && \text{for} \quad n \leq 0. \end{aligned}$$

These numbers can be generated by the following generating function:

$$\begin{aligned} F_d(z) &= \sum_{n\geq 0} c_{n,d} z^n = \frac{1+(d-2)z-z^2-\cdots-z^d}{1-2z+z^{d+1}} \\ &= \frac{1+(d-3)z-(d-1)z^2+z^{d+1}}{(1-z)(1-2z+z^{d+1})} \, . \end{aligned}$$

The $(1,d)$-complexity $N(n,d)$ can be expressed with these numbers $c_{n,d}$ by the following formula:

$$N(n,d) = \frac{1}{d-1}\left(\sum_{i=1}^{n} c_{i,d} - n\right), \qquad \text{for } \ d>1 \, ,$$

and

$$N(n,1) = \frac{n(n+1)}{2} \, ,$$

or

$$N(n,d) = N(n-1,d) + \frac{1}{d-1}(c_{n,d}-1), \qquad \text{for} \quad d>1, \ \ n>1 \, .$$

If $d=2$ then

$$F_2(z) = \frac{1-z^2}{1-2z+z^3} = \frac{1+z}{1-z-z^2} = \frac{F(z)}{z} + F(z) \, ,$$

where $F(z)$ is the generating function of the Fibonacci numbers F_n (with $F_0 = 0, \ \ F_1 = 1$). Then, from this formula we have

$$c_{n,2} = F_{n+1} + F_n = F_{n+2} \, ,$$

and

$$N(n,2) = \sum_{i=1}^{n} F_{i+2} - n = F_{n+4} - n - 3 \, .$$

Figure 26.13 contains the values of $N(n,d)$ for $k \leq 10$ and $d \leq 10$.

$$N(n,d) = 2^n - 1, \qquad \text{for any} \quad d \geq n-1.$$

The following theorem gives the value of $N(n,d)$ in the case $n \geq 2d-2$:

Theorem 26.23 *For $n \geq 2d-2$ we have*

$$N(n,n-d) = 2^n - (d-2)\cdot 2^{d-1} - 2 \, .$$

The main step in the proof is based on the formula

$$N(n,n-d-1) = N(n,n-d) - d \cdot 2^{\mathrm{d}-1} \, .$$

The value of $N(n,d)$ can be also obtained by computing the number of sequences

$n \backslash d$	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	3	3	3	3	3	3	3	3	3	3
3	6	7	7	7	7	7	7	7	7	7
4	10	14	15	15	15	15	15	15	15	15
5	15	26	30	31	31	31	31	31	31	31
6	21	46	58	62	63	63	63	63	63	63
7	28	79	110	122	126	127	127	127	127	127
8	36	133	206	238	250	254	255	255	255	255
9	45	221	383	464	494	506	510	511	511	511
10	55	364	709	894	974	1006	1018	1022	1023	1023

Figure 26.13 The $(1, d)$-complexity of words of length n

of length k of 0's and 1's, with no more than $d-1$ adjacent zeros. In such a sequence one 1 represents the presence, one 0 does the absence of a letter of the word in a given $(1, d)$-subword. Let $b_{n,d}$ denote the number of n-length sequences of zeros and ones, in which the first and last position is 1, and the number of adjacent zeros is at most $d-1$. Then it can be proved easily that

$$\begin{aligned}
&b_{n,d} = b_{n-1,d} + b_{n-2,d} + \ldots + b_{n-d,d}\,, \text{ for } \quad k > 1\,,\\
&b_{1,d} = 1\,,\\
&b_{n,d} = 0, \text{ for all } n \le 0\,,
\end{aligned}$$

because any such sequence of length $n-i$ $(i = 1, 2, ..., d)$ can be continued in order to obtain a similar sequence of length n in only one way (by adding a sequence of the form $0^{i-1}1$ on the right). For $b_{n,d}$ the following formula also can be derived:

$$b_{n,d} = 2b_{n-1,d} - b_{n-1-d,d}\,.$$

If we add one 1 or 0 at an internal position (e.g at the $(n-2)^{th}$ position) of each $b_{n-1,d}$ sequences, then we obtain $2b_{n-1,d}$ sequences of length n, but from these, $b_{n-1-d,d}$ sequences will have d adjacent zeros.

The generating function corresponding to $b_{n,d}$ is

$$B_d(z) = \sum_{n\ge 0} b_{n,d} z^n = \frac{z}{1 - z \cdots - z^d} = \frac{z(1-z)}{1 - 2z + z^{d+1}}\,.$$

By adding zeros on the left and/or on the right to these sequences, we can obtain the number $N(k, d)$, as the number of all these sequences. Thus

$$N(k, d) = b_{k,d} + 2b_{k-1,d} + 3b_{k-2,d} + \cdots + kb_{1,d}\,.$$

(i zeros can be added in $i+1$ ways to these sequences: 0 on the left and i on the right, 1 on the left and $i-1$ on the right, and so on).

From the above formula, the generating function corresponding to the complexities $N(k, d)$ can be obtained as a product of the two generating functions $B_d(z)$ and

$A(z) = \sum_{n\geq 0} nz^n = 1/(1-z)^2$, thus:

$$N_d(z) = \sum_{n\geq 0} N(n,d)z^n = \frac{z}{(1-z)(1-2z+z^{d+1})}\,.$$

The case $d_2 = n-1$ In the sequel instead of d_1 we will use d. In this case the distance between two letters picked up to be neighbours in a subword is at least d.

Let us denote by $b_{n,d}(i)$ the number of $(d,n-1)$-subwords which begin at the position i in a rainbow word of length n. Using our previous example $(abcdef)$, we can see that $b_{6,2}(1) = 8$, $b_{6,2}(2) = 5$, $b_{6,2}(3) = 3$, $b_{6,2}(4) = 2$, $b_{6,2}(5) = 1$, and $b_{6,2}(6) = 1$.

The following formula immediately results:

$$b_{n,d}(i) = 1 + b_{n,d}(i{+}d) + b_{n,d}(i{+}d{+}1) + \cdots + b_{n,d}(n)\,, \tag{26.2}$$

for $n > d$, and $1 \leq i \leq n-d\,,$

$$b_{n,d}(1) = 1 \text{ for } \ n \leq d\,.$$

For simplicity, $C(n;d,n)$ will be denoted by $K(n,d)$.

The $(d,n-1)$-complexity of rainbow words can be computed by the formula:

$$K(n,d) = \sum_{i=1}^{n} b_{n,d}(i)\,. \tag{26.3}$$

This can be expressed also as

$$K(n,d) = \sum_{k=1}^{n} b_{k,d}(1)\,, \tag{26.4}$$

because of the formula

$$K(n+1,d) = K(n,d) + b_{n+1,d}(1)\,.$$

In the case $d = 1$ the complexity $K(n,1)$ can be computed easily: $K(n,1) = 2^n - 1$.

From (26.2) we get the following algorithm for the computation of $b_{n,d}(i)$. The numbers $b_{n,d}(k)\,(k = 1,2,\ldots)$ for a given n and d are obtained in the array $b = (b_1, b_2, \ldots)$. Initially all these elements are equal to -1. The call for the given n and d and the desired i is:

```
Input $(n,d,i)$
for $k \leftarrow 1$ to $n$
    do $b_k \leftarrow -1$
B$(n,d,i)$                                        ▷ Array $b$ is a global one.
Output $b_1,\ b_2,\ \ldots,\ b_n$
```

$n \backslash d$	1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1
2	3	2	2	2	2	2	2	2	2	2	2
3	7	4	3	3	3	3	3	3	3	3	3
4	15	7	5	4	4	4	4	4	4	4	4
5	31	12	8	6	5	5	5	5	5	5	5
6	63	20	12	9	7	6	6	6	6	6	6
7	127	33	18	13	10	8	7	7	7	7	7
8	255	54	27	18	14	11	9	8	8	8	8
9	511	88	40	25	19	15	12	10	9	9	9
10	1023	143	59	35	25	20	16	13	11	10	10
11	2047	232	87	49	33	26	21	17	14	12	11
12	4095	376	128	68	44	33	27	22	18	15	13

Figure 26.14 Values of $K(n, d)$.

The recursive algorithm is the following:

```
B(n, d, i)
1  $p \leftarrow 1$
2  for $k \leftarrow i + d$ to $n$
3      do if $b_k = -1$
4             then B$(n, d, k)$
5          $p \leftarrow p + b_k$
6  $b_i \leftarrow p$
7  return
```

This algorithm is a linear one.

If the call is $B(8, 2, 1)$, the elements will be obtained in the following order: $b_7 = 1$, $b_8 = 1$, $b_5 = 3$, $b_6 = 2$, $b_3 = 8$, $b_4 = 5$, and $b_1 = 21$.

Lemma 26.24 *$b_{n,2}(1) = F_n$, where F_n is the nth Fibonacci number.*

Proof Let us consider a rainbow word $a_1 a_2 \ldots a_n$ and let us count all its $(2, n-1)$-subwords which begin with a_2. If we change a_2 for a_1 in each $(2, n-1)$-subword which begin with a_2, we obtain $(2, n-1)$-subwords too. If we add a_1 in front of each $(2, n-1)$-subword which begin with a_3, we obtain $(2, n-1)$-subwords too. Thus

$$b_{n,2}(1) = b_{n-1,2}(1) + b_{n-2,2}(1) .$$

So $b_{n,2}(1)$ is a Fibonacci number, and because $b_{1,2}(1) = 1$, we obtain that $b_{n,2}(1) = F_n$. ■

Theorem 26.25 *$K(n, 2) = F_{n+2} - 1$, where F_n is the nth Fibonacci number.*

Proof From equation (26.4) and Lemma 26.24:

$$\begin{aligned} K(n,2) &= b_{1,2}(1)+b_{2,2}(1)+b_{3,2}(1)+b_{4,2}(1)+\cdots+b_{n,2}(1) \\ &= F_1+F_2+\cdots+F_n \\ &= F_{n+2}-1\,. \end{aligned}$$ ■

If we use the notation $M_{n,d}=b_{n,d}(1)$, because of the formula

$$b_{n,d}(1)=b_{n-1,d}(1)+b_{n-d,d}(1)\,,$$

a generalized middle sequence will be obtained:

$$\begin{aligned} M_{n,d} &= M_{n-1,d}+M_{n-d,d}\,, \quad \text{for } n\ge d\ge 2\,, \\ M_{0,d} &= 0,\ M_{1,n}=1,\ \ldots,\ M_{d-1,d}=1\,. \end{aligned} \tag{26.5}$$

Let us call this sequence ***d-middle sequence.*** Because of the equality $M_{n,2}=F_n$, the d-middle sequence can be considered as a generalization of the Fibonacci sequence.

Then next linear algorithm computes $M_{n,d}$, by using an array $M_0,M_1,\ldots,M_{d-1}$ to store the necessary previous elements:

Middle(n,d)

```
1  $M_0 \leftarrow 0$
2  **for** $i \leftarrow 1$ **to** $d-1$
3      **do** $M_i \leftarrow 1$
4  **for** $i \leftarrow d$ **to** $n$
5      **do** $M_{i \bmod d} \leftarrow M_{(i-1) \bmod d} + M_{(i-d) \bmod d}$
6         print $M_{i \bmod d}$
7  **return**
```

Using the generating function $M_d(z)=\sum_{n\ge 0} M_{n,d}z^n$, the following closed formula can be obtained:

$$M_d(z)=\frac{z}{1-z-z^d}\,. \tag{26.6}$$

This can be used to compute the sum $s_{n,d}=\sum_{n=1}^{n} M_{i,d}$, which is the coefficient of z^{n+d} in the expansion of the function

$$\frac{z^d}{1-z-z^d}\cdot\frac{1}{1-z}=\frac{z^d}{1-z-z^d}+\frac{z}{1-z-z^d}-\frac{z}{1-z}\,.$$

So $s_{n.d}=M_{n+(d-1),d}+M_{n,d}-1=M_{n+d,d}-1$. Therefore

$$\sum_{i=1}^{n} M_{i,d}=M_{n+d,d}-1\,. \tag{26.7}$$

Theorem 26.26 *$K(n,d) = M_{n+d,d} - 1$, where $n > d$ and $M_{n,d}$ is the nth elements of d-middle sequence.*

Proof The proof is similar to that in Theorem 26.25 taking into account the equation (26.7). ■

Theorem 26.27 $K(n,d) = \sum_{k\geq 0} \binom{n-(d-1)k}{k+1}$, *for $n \geq 2, d \geq 1$.*

Proof Let us consider the generating function $G(z) = \dfrac{1}{1-z} = 1 + z + z^2 + \cdots$. Then, taking into account the equation (26.6) we obtain $M_d(z) = zG(z+z^d) = z + z(z+z^d) + z(z+z^d)^2 + \cdots + z(z+z^d)^i + \cdots$. The general term in this expansion is equal to

$$z^{i+1} \sum_{k=1}^{i} \binom{i}{k} z^{(d-1)k},$$

and the coefficient of z^{n+1} is equal to

$$\sum_{k\geq 0} \binom{n-(d-1)k}{k}.$$

The coefficient of z^{n+d} is

$$M_{n+d,d} = \sum_{k\geq 0} \binom{n+d-1-(d-1)k}{k}. \tag{26.8}$$

By Theorem 26.26 $K(n,d) = M_{n+d,d} - 1$, and an easy computation yields

$$K(n,d) = \sum_{k\geq 0} \binom{n-(d-1)k}{k+1}.$$

■

26.2.2. General words

The algorithm WARSHALL-LATIN can be used for nonrainbow words too, with the remark that repeating subwords must be eliminated. For the word $aabbbaaa$ and $d_1 = 2$, $d_2 = 4$ the result is: $ab, abb, aba, abba, abaa, aa, aaa, bb, ba, bba, baa$, and with a and b we have $C_{aabbbaaa}(2,4) = 13$.

26.3. Palindrome complexity

The ***palindrome complexity function*** pal_w of a finite or infinite word w attaches to each $n \in \mathbf{N}$ the number of palindrome subwords of length n in w, denoted by

$\mathrm{pal}_w(n)$.

The ***total palindrome complexity*** of a finite word $w \in A^*$ is equal to the number of all nonempty palindrome subwords of w, i.e.:

$$P(w) = \sum_{n=1}^{|w|} \mathrm{pal}_w(n)\,.$$

This is similar to the total complexity of words.

26.3.1. Palindromes in finite words

Theorem 26.28 *The total palindrome complexity $P(w)$ of any finite word w satisfies $P(w) \le |w|$.*

Proof We proceed by induction on the length n of the word w. For $n = 1$ we have $P(w) = 1$.

We consider $n \ge 2$ and suppose that the assertion holds for all words of length $n-1$. Let $w = a_1a_2\ldots a_n$ be a word of length n and $u = a_1a_2\ldots a_{n-1}$ its prefix of length $n-1$. By the induction hypothesis it is true that $P(u) \le n-1$.

If $a_n \ne a_j$ for each $j \in \{1, 2, \ldots n-1\}$, the only palindrome in w which is not in u is a_n, hence $P(w) = P(u) + 1 \le n$.

If there is an index j, $1 \le j \le n-1$ such that $a_n = a_j$, then $P(w) > P(u)$ if and only if w has suffixes which are palindromes. Let us suppose that there are at least two such suffixes $a_ia_{i+1}\ldots a_n$ and $a_{i+k}a_{i+k+1}\ldots a_n$, $1 \le k \le n-i$, which are palindromes. It follows that

$$\begin{aligned}
&a_i = a_n = a_{i+k}\\
&a_{i+1} = a_{n-1} = a_{i+k+1}\\
&\cdots\\
&a_{n-k} = a_{i+k} = a_n,
\end{aligned}$$

hence $a_{i+k}\ldots a_n = a_i\ldots a_{n-k}$. The last palindrome appears in u (because of $k \ge 1$) and has been already counted in $P(u)$. It follows that $P(w) \le P(u) + 1 \le n$. ■

This result shows that the total number of palindromes in a word cannot be larger than the length of that word. We examine now if there are words which are 'poor' in palindromes. In the next lemma we construct finite words w_n of arbitrary length $n \ge 9$, which contain precisely 8 palindromes.

Let us denote by $w^{\frac{p}{q}}$ the fractional power of the word w of length q, which is the prefix of length p of w^p.

Lemma 26.29 *If $w_n = (001011)^{\frac{n}{6}}$, $n \ge 9$, then $P(w_n) = 8$.*

Proof In w_n there are the following palindromes: 0, 1, 00, 11, 010, 101, 0110, 1001. Because 010 and 101 are situated in w_n between 0 on the left and 1 on the right, these cannot be continued to obtain any palindromes. The same is true for 1001 and 0110, which are situated between 1 on the left and 0 on the right, excepting the cases when 1001 is a suffix. So, there are no other palindromes in w_n. ∎

Remark 26.30 *If u is a circular permutation of 001011 and $n \geq 9$ then $P(u^{\frac{n}{6}}) = 8$ too. Because we can interchange 0 with 1, for any n there will be at least 12 words of length n with total complexity equal to 8.*

We shall give now, beside the upper delimitation from Theorem 26.28, lower bounds for the number of palindromes contained in finite binary words. (In the trivial case of a 1-letter alphabet it is obvious that, for any word w, $P(w) = |w|$.)

Theorem 26.31 *If w is a finite word of length n on a 2-letter alphabet, then*

$$\begin{array}{ll} P(w) = n, & \text{for } 1 \leq n \leq 7\,, \\ 7 \leq P(w) \leq 8, & \text{for } n = 8\,, \\ 8 \leq P(w) \leq n, & \text{for } n \geq 9\,. \end{array}$$

Proof Up to 8 the computation can be made by a computer program. For $n \geq 9$, Lemma 26.29 gives words v_n for which $P(v_n) = 8$. The maximum value is obtained for words of the form a^n, $a \in A$, $n \in \mathbf{N}$. ∎

Remark 26.32 *For all the short binary words (up to $|w| = 7$), the palindrome complexity takes the maximum possible value given in Theorem 26.28; from the words with $|w| = 8$, only four (out of 2^8) have $P(w) = 7$, namely 00110100, 00101100 and their complemented words.*

In the following lemmas we construct binary words which have a given total palindrome complexity greater than or equal to 8.

Lemma 26.33 *If $u_{k,\ell} = 0^k 10110^\ell 1$ for $k \geq 2$ and $1 \leq \ell \leq k-1$, then $P(u_{k,\ell}) = k+6$.*

Proof In the prefix of length k of $u_{k,\ell}$ there are always k palindromes $(1, \ldots, 1^k)$. The other palindromes different from these are 1, 11, 010, 101, 0110 and $10^\ell 1$ (for $\ell \geq 2$), respectively 101101 (for $\ell = 1$). In each case $P(u_{k,\ell}) = k+6$. ∎

Lemma 26.34 *If $v_{k,\ell} = (0^k 1011)^{\frac{k+\ell+5}{k+4}}$ for $k \geq 2$ and $k \leq \ell \leq n-k-5$, then $P(v_{k,\ell}) = k+6$.*

Proof Since $\ell \geq k$, the prefix of $u_{k,j}$ is at least $0^k 10110^k 1$, which includes the palindromes $0, \ldots, 0^k$, 1, 11, 010, 101, 0110 and $10^k 1$, hence $P(v_{k,\ell}) \geq k+6$. The palindromes 010 and 101 are situated between 0 and 1, while 0110 and $10^k 1$ are between 1 and 0 (excepting the cases when they are suffixes), no matter how large is ℓ. It follows that $v_{k,\ell}$ contains no other palindromes, hence $P(v_{k,\ell}) = k+6$. ∎

Remark 26.35 *If $k = 2$, then the word $v_{2,\ell}$ is equal to $w_{\ell+7}$, with w_n defined in Lemma 26.29.*

We can determine now precisely the image of the restriction of the palindrome complexity function to A^n, $n \geq 1$.

Theorem 26.36 *Let A be a binary alphabet. Then*

$$P(A^n) = \begin{cases} \{n\}, & \textit{for } 1 \leq n \leq 7\,, \\ \{7,8\}, & \textit{for } n = 8\,, \\ \{8,\ldots,n\}, & \textit{for } n \geq 9\,. \end{cases}$$

Proof Having in mind the result in Theorem 26.31, we have to prove only that for each n and i so that $8 \leq i \leq n$, there exists always a binary word $w_{n,i}$ of length n for which the total palindrome complexity is $P(w_{n,i}) = i$. Let n and i be given so that $8 \leq i \leq n$. We denote $k = i - 6 \geq 2$ and $\ell = n - k - 5$.

If $\ell \leq k - 1$, we take $w_{n,i} = u_{k,\ell}$ (from Lemma 26.33); if $\ell \geq k$, $w_{n,i} = v_{k,\ell}$ (from Lemma 26.34). It follows that $|w_{n,i}| = n$ and $P(w_{n,i}) = k + 6 = i$. ■

Example 26.2 Let us consider $n = 25$ and $i = 15$. Then $k = 15-6 = 9$, $\ell = 26-15 = 11$.

Because $\ell > k - 1$, we use $v_{9,11} = (0^9 1011)^{\frac{25}{13}} = 0^9 10110^9 101$, whose total palindrome complexity is 15.

We give similar results for the case of alphabets with $q \geq 3$ letters.

Theorem 26.37 *If w is a finite word of length n over a q-letter ($q \geq 3$) alphabet, then*

$$\begin{array}{ll} P(w) = n, & \textit{for } n \in \{1,2\}\,, \\ 3 \leq P(w) \leq n, & \textit{for } n \geq 3\,. \end{array}$$

Proof For $n \in \{1,2\}$ it can be checked directly. Let us consider now $n \geq 3$ and a word of length at least 3. If this is a trivial word (containing only one letter n times), its total palindrome complexity is $n \geq 3$. If in the word there appear exactly two letters a_1 and a_2, it will have as palindromes those two letters and at least one of a_1^2, a_2^2, $a_1a_2a_1$ or $a_2a_1a_2$, hence again $P(w) \geq 3$. If the word contains a third letter, then obviously $P(w) \geq 3$. So, the total complexity cannot be less then 3. ■

Theorem 26.38 *Let A be a q-letter ($q \geq 3$) alphabet. Then for*

$$P(A^n) = \begin{cases} \{n\}, & \textit{for } 1 \leq n \leq 2\,, \\ \{3,\ldots,n\}, & \textit{for } n \geq 3\,. \end{cases}$$

Proof It remains to prove that for each n and i so that $3 \leq i \leq n$, there exists always a word $w_{n,i}$ of length n, for which the total palindrome complexity is $P(w_{n,i}) = i$. Such a word is $w_{n,i} = a_1^{i-3}(a_1a_2a_3)^{\frac{n-i+3}{3}}$, which has $i - 2$ palindromes in its prefix of length $i - 2$, and other two palindromes a_2 and a_3 in what follows. ■

26.3.2. Palindromes in infinite words

Sturmian words The number of palindromes in the infinite Sturmian words is given by the following theorem.

Theorem 26.39 *If u is an infinite Sturmian word, then*

$$\mathrm{pal}_u(n) = \begin{cases} 1, & \textit{if } n \textit{ is even}, \\ 2, & \textit{if } n \textit{ is odd}. \end{cases}$$

Power word Let us recall the power word as being
$p = 01001100011100001111\ldots 0^n1^n \ldots$.

Theorem 26.40 *The palindrome complexity of the power word p is*

$$\mathrm{pal}_p(n) = 2\left\lfloor \frac{n}{3} \right\rfloor + 1 + \varepsilon\,,$$

where

$$\varepsilon = \begin{cases} 0, & \textit{if } n \textit{ divisible by } 3, \\ 1, & \textit{otherwise}. \end{cases}$$

Proof There exist the following cases:

Case $n = 3k$. Palindrome subwords are:

$0^i1^{3k-2i}0^i$ for $i = 0, 1, \ldots k$,
$1^i0^{3k-2i}1^i$ for $i = 0, 1, \ldots k-1$, so $\mathrm{pal}_p(3k) = 2k+1$.

Case $n = 3k+1$. Palindrome subwords are:

$0^i1^{3k+1-2i}0^i$ for $i = 0, 1, \ldots k$,
$1^i0^{3k+1-2i}1^i$ for $i = 0, 1, \ldots k$, so $\mathrm{pal}_p(3k+1) = 2k+2$.

Case $n = 3k+2$. Palindrome subwords are:

$0^i1^{3k+2-2i}0^i$ for $i = 0, 1, \ldots k$,
$1^i0^{3k+2-2i}1^i$ for $i = 0, 1, \ldots k$, so $\mathrm{pal}_p(3k+2) = 2k+2$. ■

The palindrome subwords of the power word have the following properties:
• Every palindrome subword which contains both 0's and 1's occurs only once in the power word.
• If we use the notations $U_{iji} = 0^i1^j0^i$ and $V_{iji} = 1^i0^j1^i$ then there are the unique decompositions:

$$p = U_{111}U_{121}U_{232}U_{242}U_{353}U_{363}\ldots U_{k,2k-1,k}U_{k,2k,k}\cdots,$$

$$p = 0V_{121}V_{232}V_{141}V_{353}V_{262}V_{474}V_{383}\ldots V_{k+1,2k+1,k+1}V_{k,2k+2,k}\ldots.$$

Champernowne word The Champernowne word is defined as the concatenation of consecutive binary written natural numbers:
$c = 0\,1\,10\,11\,100\,101\,110\,111\,1000\,1001\,1010\,1011\,1100\ldots$.

Theorem 26.41 *The palindrome complexity of the Champernowne word is*

$$\mathrm{pal}_c(n) = 2^{\lfloor \frac{n}{2} \rfloor + \varepsilon},$$

where

$$\varepsilon = \begin{cases} 0, & \textit{if } n \textit{ is even}, \\ 1, & \textit{if } n \textit{ is odd}. \end{cases}$$

Proof Any palindrome w of length n can be continued as $0w0$ and $1w1$ to obtain palindromes of length $n+2$. This theorem results from the following: $\mathrm{pal}_c(1) = 2$, $\mathrm{pal}_c(2) = 2$ and for $n \geq 1$ we have

$\mathrm{pal}_c(2n+1) = 2\mathrm{pal}_c(2n-1)$,

$\mathrm{pal}_c(2n+2) = 2\mathrm{pal}_c(2n)$. ■

The following algorithm generates all palindromes up to a given length of a Sturmian word beginning with the letter a, and generated by the morphism σ.

The idea is the following. If p is the least value for which $\sigma^p(a)$ and $\sigma^p(b)$ are both of odd length (such a p always exists), we consider conjugates[6] of these words, which are palindromes (such conjugates always exists), and we define the following morphism:

$\pi(a) = \mathrm{conj}\big(\sigma^p(a)\big)$,

$\pi(b) = \mathrm{conj}\big(\sigma^p(b)\big)$,

where $\mathrm{conj}(u)$ produces a conjugate of u, which is a palindrome.

The sequences $\big(\pi^n(a)\big)_{n\geq 0}$ and $\big(\pi^n(b)\big)_{n\geq 0}$ generate all odd length palindromes, and the sequence $\big(\pi^n(aa)\big)_{n\geq 0}$ all even length palindromes.

If α is a word, then $'\alpha'$ represents the word which is obtained from α by erasing its first and last letter. More generally, $^{m\prime}\alpha^{\prime m}$ is obtained from α by erasing its first m and last m letters.

[6]If $w = uv$ then vu is a conjugate of w.

```
Sturmian-Palindromes($n$)
if $n$ is even
  then $n \leftarrow n - 1$
let $p$ be the least value for which $\sigma^p(a)$ and $\sigma^p(b)$ are both of odd length
let define the morphism: $\pi(a) = \mathrm{conj}\big(\sigma^p(a)\big)$ and $\pi(b) = \mathrm{conj}\big(\sigma^p(b)\big)$
$\alpha \leftarrow a$
while $|\alpha| < n$
      do $\alpha \leftarrow \pi(\alpha)$
$m \leftarrow (|\alpha| - n)/2$
$\alpha \leftarrow {}^{m\prime}\alpha^{\prime m}$
$\beta \leftarrow b$
while $|\beta| < n$
      do $\beta \leftarrow \pi(\beta)$
$m \leftarrow (|\beta| - n)/2$
$\beta \leftarrow {}^{m\prime}\beta^{\prime m}$
repeat print $\alpha, \beta$                              ▷ Printing odd length palindromes.
         $\alpha \leftarrow {}'\alpha'$
         $\beta \leftarrow {}'\beta'$
until $\alpha = \varepsilon$ and $\beta = \varepsilon$
$\gamma \leftarrow aa$
while $|\gamma| < n + 1$
      do $\gamma \leftarrow \pi(\gamma)$
$m \leftarrow (|\gamma| - n - 1)/2$
$\gamma \leftarrow {}^{m\prime}\gamma^{\prime m}$
repeat print $\gamma$                                  ▷ Printing even length palindromes.
         $\gamma \leftarrow {}'\gamma'$
until $\gamma = \varepsilon$
```

Because any substitution requires no more than cn steps, where c is a constant, the algorithm is a linear one.

In the case of the Fibonacci word the morphism σ is defined by

$\sigma(a) = ab$, $\sigma(b) = a$,

and because

$\sigma(a) = ab$, $\sigma^2(a) = aba$, $\sigma^3(a) = abaab$, $|\sigma^3(a)| = |abaab| = 5$,

$\sigma(b) = a$, $\sigma^{(}b) = ab$, $\sigma^3(b) = aba$, $|\sigma^3(b)| = |aba| = 3$,

both being odd numbers, p will be equal to 3.

The word $abaab$ is not a palindrome, and for the morphism π we will use the adequate conjugate $ababa$, which is a palindrome.

In this case the morphism π is defined by

$\pi(a) = ababa$,

$\pi(b) = aba$.

For example, if $n = 14$, the following are obtained:

$\pi^2(a) = ababa\,aba\,ababa\,aba\,ababa$, and then $\alpha = aabaababaabaa$,

$\pi^2(b) = ababa\,aba\,ababa$, and $\beta = ababaabaababa$,

$\pi^3(aa) = ababaabaababaabaababaababaabaababaabaababa$, and

$\gamma = baababaababaab$.

The odd palindromes obtained are:		The even palindromes obtained are:
$aabaababaabaa$,	$ababaabaababa$,	$baababaababaab$,
$abaababaaba$,	$babaabaabab$,	$aababaababaa$,
$baababaab$,	$abaabaaba$,	$ababaababa$,
$aababaa$,	$baabaab$,	$babaabab$,
$ababa$,	$aabaa$,	$abaaba$,
bab,	aba,	$baab$,
a,	b,	aa.

Problems

26-1 Generating function 1
Let $b_{n,d}$ denote the number of sequences of length n of zeros and ones, in which the first and last position is 1, and the number of adjacent zeros is at most $d-1$. Prove that the generating function corresponding to $b_{n,d}$ is

$$B_d(z) = \sum_{n \geq 0} b_{n,d} z^n = \frac{z(1-z)}{1-2z+z^{d+1}} \, .$$

Hint. See Subsection 26.2.1.)
26-2 Generating function 2
Prove that the generating function of $N(n,d)$, the number of all $(1,d)$-subwords of a rainbow word of length n, is

$$N_d(z) = \sum_{n \geq 0} N(n,d) z^n = \frac{z}{(1-z)(1-2z+z^{d+1})}$$

(*Hint.* (See Subsection 26.2.1.)
26-3 Window complexity
Compute the window complexity of the infinite Fibonacci word.
26-4 Circuits in De Bruijn graphs
Prove that in the De Bruijn graph $B(q,m)$ there exist circuits (directed cycles) of any length from 1 to q^m.

Chapter Notes

The basic notions and results on combinatorics of words are given in Lothaire's [93, 94, 95] and Fogg's books [49]. Neither Lothaire nor Fogg is a single author, they are pseudonyms of groups of authors. A chapter on combinatorics of words written by Choffrut and Karhumäki [29] appeared in a handbook on formal languages.

The different complexities are defined as follows: total complexity in Iványi [74], maximal and total maximal complexity in Anisiu, Blázsik, Kása [4], $(1,d)$-complexity in Iványi [74] (called d-complexity) and used also in Kása [86]), $(d, n-1)$-complexity (called super-d-complexity) in Kása [88], scattered complexity in Kása [87], factorization complexity in Ilie [72] and window complexity in Cassaigne, Kaboré, Tapsoba [24].

The power word, lower/upper maximal/total complexities are defined in Ferenczi, Kása [46]. In this paper a characterization of Sturmian words by upper maximal and upper total complexities (Theorem 26.11) is also given. The maximal complexity of finite words is studied in Anisiu, Blázsik, Kása [4]. The total complexity of finite words is described in Kása [86], where the results of the Theorem 26.22 is conjectured too, and proved later by Levé and Séébold [91].

Different generalized complexity measures of words are defined and studied by Iványi [74] and Kása [86, 88, 87].

The results on palindrome complexity are described in M.-C. Anisiu, V. Anisiu, Kása [3] for finite words, and in Droubay, Pirillo [39] for infinite words. The algorithm for palindrome generation in Sturmian words is from this paper too.

Applications of complexities in social sciences are given in Elzinga [42, 41], and in biology in Troyanskaya et al. [151].

It is worth to consult other papers too, such as [6, 23, 37, 45, 139] (on complexity problems) and [1, 8, 17, 18, 20, 35, 71] (on palindromes).

27. Conflict Situations

In all areas of everyday life there are situations when conflicting aspects have to be taken into account simultaneously. A problem becomes even more difficult when several decision makers' or interest groups' mutual agreement is needed to find a solution.

Conflict situations are divided in three categories in terms of mathematics:

1. One decision maker has to make a decision by taking several conflicting aspects into account
2. Several decision makers have to find a common solution, when every decision maker takes only one criterion into account
3. Several decision makers look for a common solution, but every decision maker takes several criteria into account

In the first case the problem is a multi-objective optimization problem, where the objective functions make up the various aspects. The second case is a typical problem of the game theory, when the decision makers are the players and the criteria mean the payoff functions. The third case appears as Pareto games in the literature, when the different players only strive to find Pareto optimal solutions instead of optimal ones.

In this chapter we will discuss the basics of this very important and complex topic.

27.1. The basics of multi-objective programming

Suppose, that one decision maker wants to find the best decision alternative on the basis of several, usually conflicting criteria. The criteria usually represent decision objectives. At first these are usually defined verbally, such as clean air, cheap maintenance, etc. Before the mathematical model is given, firstly, these objectives have to be described by quantifiable indices. It often occurs that one criterion is described by more than one indices, such as the quality of the air, since the simultaneous presence of many types of pollution have an effect on it. In mathematics it is usually assumed that the bigger value of the certain indices (they will be called ***objective functions***) means favorable value, hence we want to maximize all of the objective functions simultaneously. If we want to minimize one of the objective functions, we can safely multiply its value by (-1), and maximize the resulting new objective function. If in the case of one of the objective functions, the goal is to attain some kind of optimal value, we can maximize the deviation from it by multiplying it by (-1).

If X denotes the set of possible decision alternatives, and $f_i : X \to \mathbb{R}$ denotes the ith objective function $(i = 1, 2, \dots, I)$, the problem can be described mathematically as follows:

$$f_i(x) \to max \qquad (i = 1, 2, \dots, I)\,, \tag{27.1}$$

supposing that $x \in X$.

In the case of a single objective function we look for an ***optimal solution.*** Optimal solutions satisfy the following requirements:

(i) An optimal solution is always better than any non-optimal solution.

(ii) There is no such possible solution that provides better objective functions than an optimal solution.

(iii) If more than one optimal solution exist simultaneously, they are equivalent in the meaning that they have the same objective functions.

These properties come from the simple fact that the ***consequential space,***

$$H = \{u | u = f(x) \ \text{ for some } \ x \in X\} \tag{27.2}$$

is a subset of the real number line, which is totally ordered. In the case of multiple objective functions, the

$$H = \{\mathbf{u} = (u_1, \dots, u_I) | u_i = f_i(x), i = 1, 2, \dots, I \ \text{ for some } \ x \in X\} \tag{27.3}$$

consequential space is a subset of the I-dimensional Euclidean space, which is only partially ordered. Another complication results from the fact that a decision alternative that could maximize all of the objective functions simultaneously doesn't usually exist.

Let's denote

$$f_i^\star = \max\{f_i(x) | x \in X\} \tag{27.4}$$

the maximum of the ith objective function, then the

$$\mathbf{f}^\star = (f_1^\star, \dots, f_I^\star)$$

point is called ***ideal point.*** If $\mathbf{f}^\star \in H$, then there exits an $x^\star$ decision for which $f_i(x^\star) = f_i^\star, i = 1, 2, \dots, I$. In such special cases $x^\star$ satisfies the previously defined (i)-(iii) conditions. However, if $\mathbf{f}^\star \notin H$, the situation is much more complicated. In that case we look for Pareto optimal solutions instead of optimal ones.

Definition 27.1 *An alternative $x \in X$ is said to be* ***Pareto optimal,*** *if there is no $\overline{x} \in X$ such that $f_i(\overline{x}) \geq f_i(x)$ for all $i = 1, 2, \ldots, I$, with at least one strict inequality.*

It is not necessary that a multi-purpose optimization problem has Pareto optimal solution, as the case of the

$$H = \{(f_1, f_2) | f_1 + f_2 < 1\}$$

set shows it. Since H is open set, $(f_1 + \epsilon_1, f_2 + \epsilon_2) \in H$ for arbitrary $(f_1, f_2) \in H$ and for a small enough positive ϵ_1 and ϵ_2.

Theorem 27.2 *If X bounded, closed in a finite dimensional Euclidean space and all of the objective functions are continuous, there is Pareto optimal solution.*

The following two examples present a discrete and a continuous problem.

Example 27.1 Assume that during the planning of a sewage plant one out of two options must be chosen. The expenditure of the first option is two billion Ft, and its daily capacity is 1500 m^3. The second option is more expensive, three billion Ft with 2000 m^3 daily capacity. In this case $X = \{1, 2\}$, $f_1 = -$expenditure, $f_2 =$ capacity. The following table summarizes the data:

Options	f_1	f_2
1	-2	1500
2	-3	2000

Figure 27.1 Planning of a sewage plant.

Both options are Pareto optimal, since $-2 > -3$ and $2000 > 1500$.The H consequential space consists of two points: $(-2, 1500)$ and $(-3, 2000)$.

Example 27.2 The optimal combination of three technology variants is used in a sewage station. The first variant removes 3,2,1 mg/m^3 from one kind of pollution, and 1,3,2 mg/m^3 quantity from the another kind of pollution. Let x_1, x_2 and $1-x_1-x_2$ denote the percentage composition of the three technology variants.

The restrictive conditions:

$$\begin{aligned} x_1, x_2 &\geq 0 \\ x_1 + x_2 &\leq 1\,, \end{aligned}$$

the quantity of the removed pollution:

$$\begin{aligned} 3x_1 + 2x_2 + (1 - x_1 - x_2) &= 2x_1 + x_2 + 1 \\ x_1 + 3x_2 + 2(1 - x_1 - x_2) &= -x_1 + x_2 + 2\,. \end{aligned}$$

Since the third term is constant, we get the following two objective-function optimum problem:

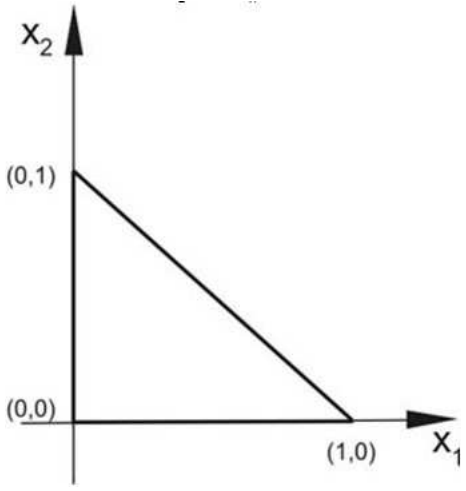


Figure 27.2 The image of set X.

$$2x_1 + x_2, -x_1 + x_2 \longrightarrow max$$

provided that

$$\begin{aligned} x_1, x_2 &\geq 0 \\ x_1 + x_2 &\leq 1\,. \end{aligned}$$

A H consequential space can be determined as follows. From the

$$\begin{aligned} f_1 &= 2x_1 + x_2 \\ f_2 &= -x_1 + x_2 \end{aligned}$$

equations

$$x_1 = \frac{f_1 - f_2}{3} \text{ and } x_2 = \frac{f_1 - 2f_2}{3}\,,$$

and from the restrictive conditions the following conditions arises for the f_1 and f_2 objective functions:

$$\begin{aligned} x_1 \geq 0 &\iff f_1 - f_2 \geq 0 \\ x_2 \geq 0 &\iff f_1 + 2f_2 \geq 0 \\ x_1 + x_2 \leq 1 &\iff 2f_1 + f_2 \leq 3\,. \end{aligned}$$

Figures **??** and **??** display the X and H sets.

On the basis of the image of the H set, it is clear that the points of the straight section joining $(1,1)$ to $(2,-1)$ are Pareto optimal. Point $(2,-1)$ isn't better than any possible point of H, because in the first objective function it results the worst possible planes. The points of the section are not equivalent to each other, either, going down from the point $(1,1)$ towards point $(2,1)$, the first objective function is increasing, but the second one is continually decreasing. Thus the (ii) and (iii) properties of the optimal solution doesn't

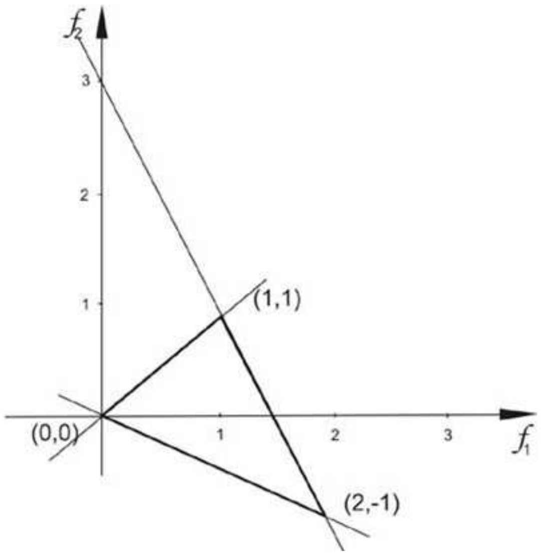


Figure 27.3 The image of set H.

remain valid in the case of multi-objection.

As we saw in the previous example, the different Pareto optimal solutions result in different objective function values, so it is primary importance to decide which one should be selected in a particular case. This question is answered by the methodology of the multi-objective programming. Most methods' basis is to substitute some real-valued „ “value-function” for the objective functions, that is the preference generated by the objective functions is replaced by a single real-valued function. In this chapter the most frequently used methods of multi-objective programming are discussed.

27.1.1. Applications of utility functions

A natural method is the following. We assign one utility function to every objective function. Let $u_i(f_i(x))$ denote the utility function of the ith objective function. The construction of the u_i function can be done by the usual manner of the theory of utility functions, for example the decision maker can subjectively define the u_i values for the given f_i values, then a continuous function can be interpolated on the resulting point. In the case of additive independent utility function additive, whereas in the case of independent of usefulness utility function additive or multiplicative aggregate utility function can be obtained. That is, the form of the aggregate utility function is either

$$u(\mathbf{f}) = \sum_{i=1}^{I} k_i u_i(f_i) \tag{27.5}$$

or

$$ku(\mathbf{f}) + 1 = \prod_{i=1}^{I} kk_i u_i(f_i) + 1 \,. \tag{27.6}$$

In such cases the multi-objective optimization problem can be rewrite to one objective-function form:

$$u(\mathbf{f}) \longrightarrow max \tag{27.7}$$

provided that $x \in X$, and thus $u(\mathbf{f})$ means the "value-function".

Example 27.3 Consider again the decision making problem of the previous example. The range of the first objective function is $[0,2]$, while the range of the second one is $[-1,1]$. Assuming linear utility functions

$$u_1(f_1) = \frac{1}{2}(f_1) \text{ and } u_2(f_2) = \frac{1}{2}(f_2) + 1 \,.$$

In addition, suppose that the decision maker gave the

$$u(0,-1) = 0, u(2,1) = 1, \text{ and the } u(0,1) = \frac{1}{4}$$

values. Assuming linear utility functions

$$u(f_1, f_2) = k_1 u_1(f_1) + k_2 u_2(f_2) \,,$$

and in accordance with the given values

$$\begin{aligned} 0 &= k_1 0 + k_2 0 \\ 1 &= k_1 1 + k_2 1 \\ \frac{1}{4} &= k_1 0 + k_2 1 \,. \end{aligned}$$

By the third equation $k_2 = \frac{1}{4}$, and by the second one we obtain $k_1 = \frac{3}{4}$, so that

$$u(f_1, f_2) = \frac{3}{4}u_1(f_1) + \frac{1}{4}u_2(f_2) = \frac{3}{4}\frac{1}{2}(2x_1 + x_2) + \frac{1}{4}\frac{1}{2}(-x_1 + x_2 + 1) = \frac{5}{8}x_1 + \frac{4}{8}x_2 + \frac{1}{8} \,.$$

Thus we solve the following one objective-function problem:

$$\frac{5}{8}x_1 + \frac{4}{8}x_2 \longrightarrow max$$

provided that

$$\begin{aligned} x_1, x_2 &\geq 0 \\ x_1 + x_2 &\leq 1 \,. \end{aligned}$$

Apparently, the optimal solution is: $x_1 = 1$, $x_2 = 0$, that is the first technology must be chosen.

Assume that the number of objective functions is n and the decision maker gives N vectors: $(f_1^{(l)}, \ldots, f_n^{(l)})$ and the related $u^{(l)}$ aggregated utility function values.

Then the $k_1, \ldots, k_n$ coefficients can be given by the solution of the

$$k_1 u_1(f_1^{(l)}) + \cdots + k_n u_n(f_n^{(l)}) = u^{(l)} \qquad (l = 1, 2, \ldots, N)$$

equation system. We always suppose that $N \geq n$, so that we have at least as many equations as the number of unknown quantities. If the equation system is contradictory, we determine the best fitting solution by the method of least squares. Suppose that

$$\mathbf{U} = \begin{pmatrix} u_{11} & \cdots & u_{1n} \\ u_{21} & \cdots & u_{2n} \\ \vdots & & \vdots \\ u_{N1} & \cdots & u_{Nn} \end{pmatrix} \text{ és } \mathbf{u} = \begin{pmatrix} u^{(1)} \\ u^{(2)} \\ \vdots \\ u^{(N)} \end{pmatrix} .$$

The formal algorithm is as follows:

```
Utility-Function-Method(u)
1 for $i \leftarrow 1$ to $N$
2     do for $j \leftarrow 1$ to $n$
3          do $u_{ij} \leftarrow u_j(f_j^{(i)})$
4 $\mathbf{k} \leftarrow (\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T\mathbf{u}$ the vector of solutions
5 return $\mathbf{k}$
```

27.1.2. Weighting method

Using this method the value-function is chosen as the linear combination of the original object functions, that is we solve the

$$\sum_{i=1}^{I} \alpha_i f_i(x) \longrightarrow max \qquad (x \in X) \tag{27.8}$$

problem. If we measure the certain objective functions in different dimensions, the aggregate utility function can't be interpreted, since we add up terms in different units. In this case we generally normalize the objective functions. Let m_i and M_i the minimum and maximum of the f_i objective function on the set X. Then the normalized ith objective function is given by the

$$\overline{f_i}(x) = \frac{f_i(x) - m_i}{M_i - m_i}$$

formula, and in the (27.8) problem f_i is replaced by $\overline{f_i}$:

$$\sum_{i=1}^{I} \alpha_i \overline{f_i}(x) \longrightarrow max\,. \qquad (x \in X) \tag{27.9}$$

It can be shown, that if all of the α_i weights are positive, the optimal solutions of (27.9) are Pareto optimal with regard to the original problem.

Example 27.4 Consider again the case of Example 27.2. From Figure 27.3, we can see that $m_1 = 0$, $M_1 = 2$, $m_2 = -1$, and $M_2 = 1$. Thus the normalized objective functions are:

$$\overline{f_1}(x_1, x_2) = \frac{2x_1 + x_2 - 0}{2 - 0} = x_1 + \frac{1}{2}x_2$$

and

$$\overline{f_2}(x_1, x_2) = \frac{-x_1 + x_2 + 1}{1 + 1} = -\frac{1}{2}x_1 + \frac{1}{2}x_2 + \frac{1}{2} .$$

Assume that the objective functions are equally important, so we choose equivalent weights: $\alpha_1 = \alpha_2 = \frac{1}{2}$, in this way the aggregate objective function is:

$$\frac{1}{2}(x_1 + \frac{1}{2}x_2) + \frac{1}{2}(-\frac{1}{2}x_1 + \frac{1}{2}x_2 + \frac{1}{2}) = \frac{1}{4}x_1 + \frac{1}{2}x_2 + \frac{1}{4} .$$

It is easy to see that the optimal solution on set X:

$$x_1 = 0, x_2 = 1 ,$$

that is, only the second technology variant can be chosen.

Suppose that $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_I)$. The formal algorithm is as follows:

```
WEIGHTING-METHOD($\boldsymbol{\alpha}$)
1  for $i \leftarrow 1$ to $I$
2      do $m_i \leftarrow (f_i(x) \longrightarrow min)$
3         $M_i \leftarrow (f_i(x) \longrightarrow max)$
4  $k \leftarrow (\sum_{i=1}^{I} \alpha_i \overline{f_i} \longrightarrow max)$
5  return $k$
```

27.1.3. Distance-dependent methods

If we normalize the objective functions, the certain normalized objective functions most favorable value is 1 and the most unfavourable is 0. So that $\mathbf{1} = (1, 1, \ldots, 1)$ is the ideal point and $\mathbf{0} = (0, 0, \ldots, 0)$ is the worst yield vector.

In the case of distance-dependent methods we either want to get nearest to the vector $\mathbf{1}$ or get farthest from the point $\mathbf{0}$, so that we solve either the

$$\varrho(\mathbf{f}(x), \mathbf{1}) \longrightarrow min \qquad (x \in X) \tag{27.10}$$

or the

$$\varrho(\mathbf{f}(x), \mathbf{0}) \longrightarrow max \qquad (x \in X) \tag{27.11}$$

problem, where ϱ denotes some distance function in $\mathbb{R}^I$.

In practical applications the following distance functions are used most frequently:

$$\varrho_1(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{I} \alpha_i |a_i - b_i| \tag{27.12}$$

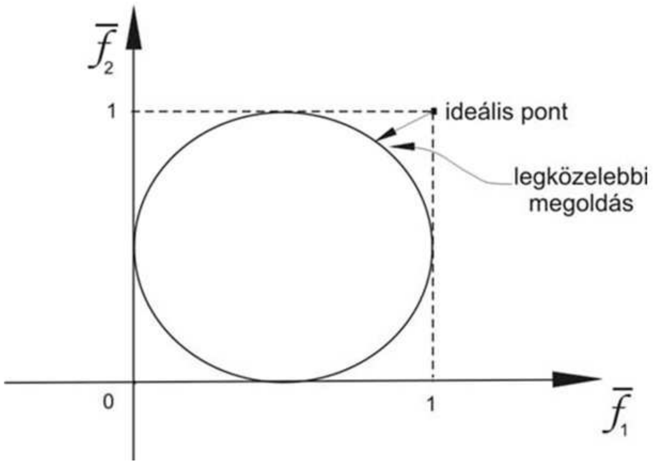


Figure 27.4 Minimizing distance.

$$\varrho_2(\mathbf{a},\mathbf{b}) = \left(\sum_{i=1}^{I} \alpha_i |a_i - b_i|^2\right)^{\frac{1}{2}} \tag{27.13}$$

$$\varrho_\infty(\mathbf{a},\mathbf{b}) = \max_i\{\alpha_i |a_i - b_i|\} \tag{27.14}$$

$$\varrho_g(\mathbf{a},\mathbf{b}) = \prod_{i=1}^{I} |a_i - b_i|^{\alpha_i} \, . \tag{27.15}$$

The $\varrho_1, \varrho_1, \varrho_\infty$ distance functions the commonly known Minkowski distance for $p = 1, 2, \infty$. The ϱ_g geometric distance doesn't satisfy the usual requirements of distance functions however, it is frequently used in practice. As we will see it later, Nash's classical conflict resolution algorithm uses the geometric distance as well. It is easy to prove that the methods using the ϱ_1 distance are equivalent of the weighting method. Notice firstly that

$$\varrho_1(\mathbf{f}(x),\mathbf{1}) = \sum_{i=1}^{I} \alpha_i |f_i(x) - 1| = \sum_{i=1}^{I} \alpha_i |1 - f_i(x)| = \sum_{i=1}^{I} \alpha_i - \sum_{i=1}^{I} \alpha_i f_i(x) \, , \tag{27.16}$$

where the first term is constant, while the second term is the objective function of the weighting method. Similarly,

$$\varrho_1(\mathbf{f}(x),\mathbf{0}) = \sum_{i=1}^{I} \alpha_i |f_i(x) - 0| = \sum_{i=1}^{I} \alpha_i (f_i(x) - 0) = \sum_{i=1}^{I} \alpha_i f_i(x) \tag{27.17}$$

which is the objective function of the weighting method.

The method is illustrated in Figures 27.4. and 27.5.

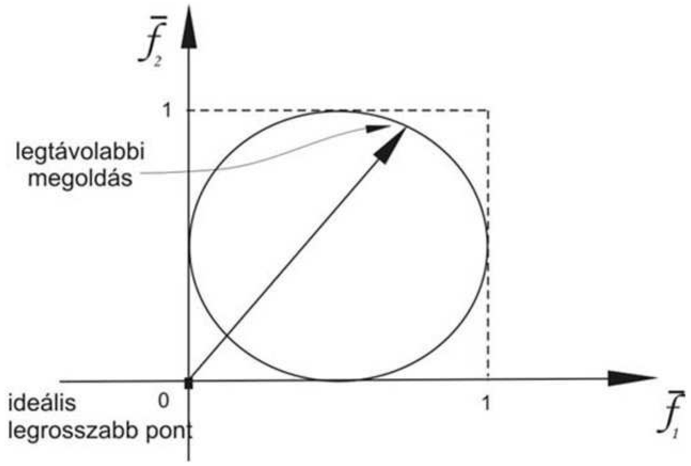


Figure 27.5 Maximizing distance.

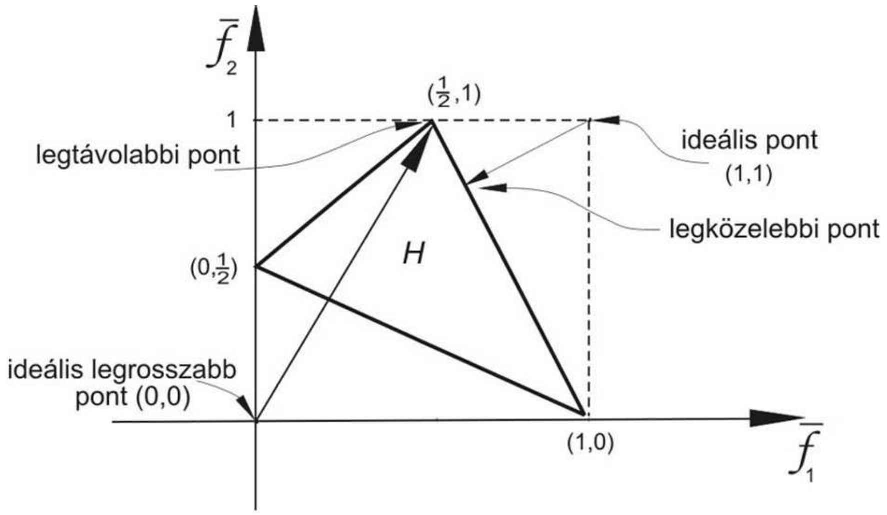


Figure 27.6 The image of the normalized set H.

Example 27.5 Consider again the problem of the previous example. The normalized consequences are shown by Figure 27.6. The two coordinates are:

$$\overline{f_1} = \frac{f_1}{2} \quad \text{and} \quad \overline{f_2} = \frac{f_2+1}{2}\,.$$

Choosing the $\alpha_1 = \alpha_2 = \frac{1}{2}$ and the ϱ_2 distances, the nearest point of $\overline{H}$ to the ideal point is

$$\overline{f_1} = \frac{3}{5}, \overline{f_2} = \frac{4}{5}\,.$$

Hence

$$f_1 = 2\overline{f_1} = 2x_1 + x_2 = \frac{6}{5} \text{ and } f_2 = 2\overline{f_1} - 1 = -x_1 + x_2 = \frac{3}{5},$$

that is the optimal decision is:

$$x_1 = \frac{1}{5}, x_2 = \frac{4}{5}, 1 - x_1 - x_2 = 0.$$

Therefore only the first two technology must be chosen in 20% and 80% proportion.

Let's choose again equivalent weights ($\alpha_1 = \alpha_2 = \frac{1}{2}$) and the ϱ_2 distance, but look for the farthest point of $\overline{H}$ from the ideal worst point. We can see from Figure 27.5, that the solution is

$$\overline{f_1} = \frac{f_1}{2}, \overline{f_2} = 1,$$

so

$$f_1 = 2\overline{f_1} = 1, f_2 = 2\overline{f_2} - 1 = 1.$$

Thus the optimal decision is: $x_1 = 0$ and $x_2 = 1$

The formal algorithm is as follows:

Distance-Dependent-Method($\boldsymbol{\varrho}, \mathbf{f}$)

```
1  for $i \leftarrow 1$ to $I$
2      do $m_i \leftarrow (f_i(x) \longrightarrow min)$
3         $M_i \leftarrow (f_i(x) \longrightarrow max)$
4         $\overline{f_i}(x) \leftarrow (f_i(x) - m_i)/(M_i - m_i)$
5  $k \leftarrow (\varrho(\overline{\mathbf{f}}(x), \mathbf{1}) \longrightarrow min)$ or $k \leftarrow (\varrho(\overline{\mathbf{f}}(x), \mathbf{0}) \longrightarrow max)$
6  return $k$
```

27.1.4. Direction-dependent methods

Assume that we have a $\mathbf{f}_*$ point in set H, on which we'd like to improve. $\mathbf{f}_*$ denotes the present position, on which the decision maker wants to improve, or at design level we can choose the worst point for the starting one. Furthermore, we assume that the decision maker gives an improvement direction vector, which is denoted by $\mathbf{v}$. After that, the task is to reach the farthest possible point in set H starting from $\mathbf{f}_*$ along the $\mathbf{v}$ direction vector. Thus, mathematically we solve the

$$t \longrightarrow max \qquad (\mathbf{f}_* + t\mathbf{v} \in H) \tag{27.18}$$

optimum task, and the related decision is given by the solution of the

$$\mathbf{f}(x) = \mathbf{f}_* + t\mathbf{v} \tag{27.19}$$

equation under the optimal t value. The method is illustrated in Figure 27.7.

Example 27.6 Let's consider again the problem of Example 27.2, and assume that $\mathbf{f}_* = (0, -1)$, which contains the worst possible objective function values in its components. If we want to improve the objective functions equally, we have to choose $\mathbf{v} = (1, 1)$. The

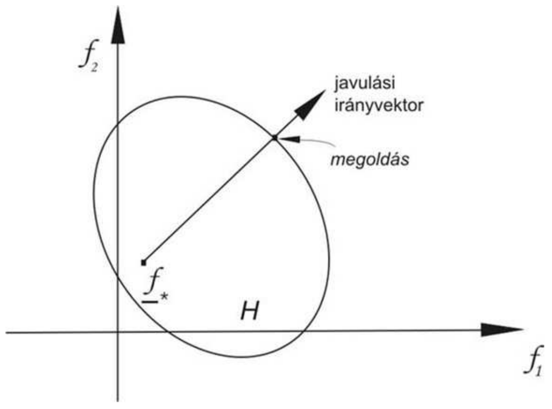


Figure 27.7 Direction-dependent methods.

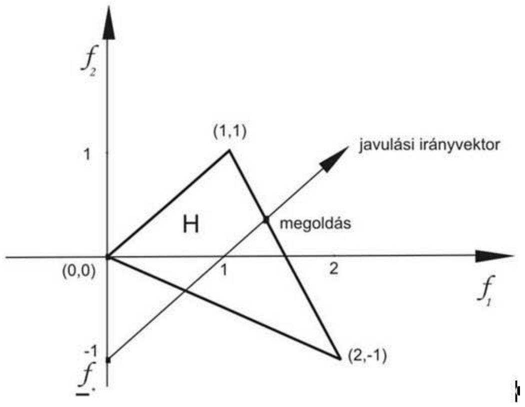


Figure 27.8 The graphical solution of Example 27.6

graphical solution is illustrated in Figure 27.8, that

$$f_1 = \frac{4}{3} \text{ and } f_2 = \frac{1}{3}\,,$$

so the appropriate values of the decision variables are the following:

$$x_1 = \frac{1}{3} \text{ és } x_2 = \frac{2}{3}\,.$$

A very rarely used variant of the method is when we diminishes the object function values systematically starting from an unreachable ideal point until a possible solution is given. If $\mathbf{f}^*$ denotes this ideal point, the (27.18) optimum task is modified as follows:

$$t \longrightarrow min \qquad (\mathbf{f}^* - t\mathbf{v} \in H) \tag{27.20}$$

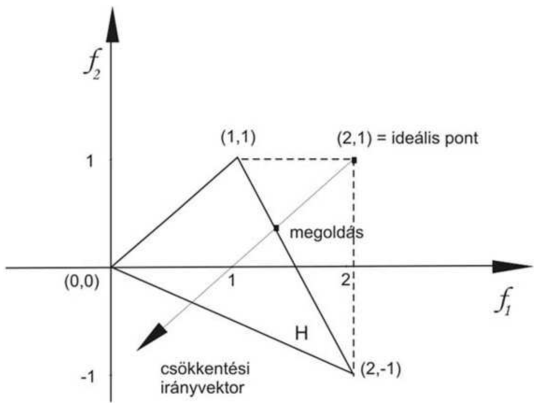


Figure 27.9 The graphical solution of Example 27.7

and the appropriate decision is given by the solution of the

$$\mathbf{f} = \mathbf{f}^* - t\mathbf{v} \tag{27.21}$$

equation.

Example 27.7 To return to the previous example, consider again that $\mathbf{f}^* = (2, 1)$ and $\mathbf{v} = (1, 1)$, that is we want to diminish the object functions equally. Figure 27.9 shows the graphical solution of the problem, in which we can see that the given solution is the same as the solution of the previous example.

Applying the method is to solve the (27.18) or the (27.20) optimum tasks, and the optimal decision is given by the solution of the (27.19) or the (27.21) equations.

Exercises

27.1-1 Determine the consequence space H for the following exercise:

$$x_1 + x_2 \longrightarrow max \qquad x_1 - x_2 \longrightarrow max$$

provided that

$$\begin{aligned} x_1, x_2 &\geq 0 \\ 3x_1 + x_2 &\leq 3 \\ x_1 + 3x_2 &\leq 3\,. \end{aligned}$$

27.1-2 Consider the utility functions of the decision maker: $u_1(f_1) = f_1$ és $u_2(f_2) = \frac{1}{2}f_2$. Furthermore, assume that the decision maker gave the $u(0, 0) = 0, u(1, 0) = u(0, 1) = \frac{1}{2}$ values. Determine the form of the aggregate utility function.
27.1-3 Solve Exercise 27.1-1 using the weighting-method without normalizing the objective functions. Choose the $\alpha_1 = \alpha_2 = \frac{1}{2}$ weights.

27.1-4 Repeat the previous exercise, but do normalize the objective functions.
27.1-5 Solve Exercise 27.1-1 with normalized objective functions, $\alpha_1 = \alpha_2 = \frac{1}{2}$ weights and minimizing the

(i) ϱ_1 distance

(ii) ϱ_2 distance

(iii) ϱ_∞distance.

27.1-6 Repeat the previous exercise, but maximize the distance from the $\mathbf{0}$ vector instead of minimizing it.
27.1-7 Solve Exercise 27.1-1 using the direction-dependent method, choosing $\mathbf{f}_* = (0, -1)$ and $\mathbf{v} = (1, 1)$.
27.1-8 Repeat the previous exercise, but this time choose $\mathbf{f}_* = (\frac{3}{2}, 1)$ and $\mathbf{v} = (1, 1)$.

27.2. Method of equilibrium

In this chapter we assume that I decision makers interested in the selection of a mutual decision alternative. Let $f_i : X \mapsto \mathbb{R}$ denote the objective function of the ith decision maker, which is also called payoff function in the game theory literature. Depending on the decision makers relationship to each other we can speak about cooperative and non-cooperative games. In the first case the decision makers care about only their own benefits, while in the second case they strive for an agreement when every one of them are better off than in the non-cooperative case. In this chapter we will discuss the non-cooperative case, while the cooperative case will be topic of the next chapter.

Let's denote $H_i(x)$ for $i = 1, 2, \ldots, I$ and $x \in X$, the set of the decision alternatives into which the ith decision maker can move over without the others' support. Evidently $H_i(x) \subseteq X$.

Definition 27.3 *An $x^* \in X$ alternative is equilibrium if for all i and $x \in H_i(x^*)$,*

$$f_i(x) \le f_i(x^*)\,. \tag{27.22}$$

This definition can also be formulated that x^* is stable in the sense that none of the decision makers can change the decision alternative from x^* alone to change any objective function value for the better. In the case of non-cooperative games, the equilibrium are the solutions of the game.

For any $x \in X$ and i decision maker, the set

$$L_i(x) = \{z | z \in H_i(x) \ \text{ and for all } \ y \in H_i(x),\ \ f_i(z) \ge f_i(y)\} \tag{27.23}$$

is called the set of the best answers of the ith decision maker to alternative x. It is clear that the elements of $L_i(x)$ are those alternatives which the ith decision maker can move over from x, and which ensure the best objective functions out of all the

		$i = 2$	
		1	2
$i = 1$	1	$(1, 2)$	$(2, 1)$
	2	$(2, 4)$	$(0, 5)$

Figure 27.10 Game with no equilibrium.

possible alternatives. According to inequality (27.22) it is also clear that x^* is an equilibrium if and only if for all $i = 1, 2, \ldots, I$, $x^* \in L_i(x^*)$, that is x^* is mutual fixed point of the L_i point-to-set maps. Thus, the existence of equilibrium can be traced to the existence of mutual fixed point of point-to-set maps, so the problem can be solved by the usual methods.

It is a very common case when the collective decision is made up by the personal decisions of the certain decision makers. Let X_i denote the set of the ith decision maker's alternatives, let $x_i \in X_i$ be the concrete alternatives, and let $f_i(x_1, \ldots, x_I)$ be the objective function of the ith decision maker. That is the collective decision is $x = (x_1, \ldots, x_I) \in X_1 \times X_2 \times \cdots \times X_I = X$. In this case

$$H_i(x_1, \ldots, x_I) = \{(x_1, \ldots, x_{i-1}, z_i, x_{i+1}, \ldots, x_I) | z_i \in X_i\}$$

and the (27.22) definition of equilibrium is modified as follows:

$$f_i(x_1^*, \ldots, x_{i-1}^*, x_i, x_{i+1}^*, \ldots, x_I^*) \le f_i(x_i^*, \ldots, x_I^*) \,. \tag{27.24}$$

In the game theory literature the equilibrium is also called ***Nash-equilibrium.***

The existence of an equilibrium is not guaranteed in general. To illustrate this let's consider the $I = 2$ case, when both decision makers can choose between to alternatives: $X_1 = \{1, 2\}$ and $X_2 = \{1, 2\}$. The objective function values are shown in Figure 27.10, where the the first number in the parentheses shows the first, the second number shows the second decision maker's objective function value. If equilibrium exists, it might not be unique, what can be proved by the case of constant objective functions, when every decision alternative is an equilibrium.

If the $X_1, \ldots, X_I$ sets are finite, the equilibrium can be found easily by the method of reckoning, when we check for all of the $\mathbf{x} = (x_1, \ldots, x_I)$ decision vectors whether the component x_i can be changed for the better of the f_i objective function. If the answer is yes, $\mathbf{x}$ is not equilibrium. If none of the components can be changed in such manner, $\mathbf{x}$ is equilibrium. For the formal algorithm, let's assume that $X_1 = \{1, 2, \ldots, n_i\}$.

Equilibrium-Search

```
 1  for $i_1 \leftarrow 1$ to $n_1$
 2      do for $i_2 \leftarrow 1$ to $n_2$
 3              $\ddots$
 4              do for $i_I \leftarrow 1$ to $n_I$
 5                  do $key \leftarrow 0$
 6                  for $k \leftarrow 1$ to $n$
 7                      do for $j \leftarrow 1$ to $n_k$
 8                          do if $f_k(i_1,\ldots,i_{k-1},j,i_{k+1},\ldots,i_I) > f(i_1,\ldots,i_I)$
 9                              then $key \leftarrow 1$ and go to 10
10                  if $key = 0$
11                      then $(i_1,\ldots,i_I)$ is equilibrium
```

The existence of equilibrium is guaranteed by the following theorem.

Theorem 27.4 *Assume that for all $i = 1, 2, \ldots, I$*

(i) X_i is convex, bounded and closed in a final dimensional Euclidean space;

(ii) f_i is continuous on the set X;

(iii) for any fixed $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_I$, f_i is concave in x_i.

Then there is at least one equilibrium.

Determination of the equilibrium is usually based on the observation that for all i, x_i^* is the solution of the

$$f_i(x_1^*, \ldots, x_{i-1}^*, x_i, x_{i+1}^*, \ldots, x_I^*) \longrightarrow max \qquad (x_i \in X_i) \tag{27.25}$$

optimum task. Writing the necessary conditions of the optimal solution (for example the Kuhn-Tucker conditions) we can get an equality-inequality system which solutions include the equilibrium. To illustrate this method let's assume that

$$X_i = \{x_i | g_i(x_i) \geq 0\}$$

where x_i is a finite dimensional vector and g_i is a vector-valued function. In this way (27.25) can be rewritten as follows:

$$f_i(x_1^*, \ldots, x_{i-1}^*, x_i, x_{i+1}^*, \ldots, x_I^*) \longrightarrow max \qquad (g_i(x_i) \geq 0)\,. \tag{27.26}$$

In this case the Kuhn-Tucker necessary conditions are:

$$\begin{array}{rcl} u_i & \geq & 0 \\ g_i(x_i) & \geq & 0 \\ \nabla_i f_i(x) + u_i^T \nabla_i g_i(x_i) & = & 0^T \\ u_i^T g_i(x_i) & = & 0\,, \end{array} \tag{27.27}$$

where ∇_i denotes the gradient at x_i, and u_i is a vector which has the same length as g_i. If we formulate the (27.27) conditions for $i = 1, 2, \ldots, I$, we get an equality-inequality system which can be solved by computer methods. It is easy to see that (27.27) can also be rewritten to an nonlinear optimization task:

$$\begin{array}{rcl} \sum_{i=1}^{I} u_i^T g_i(x_i) & \longrightarrow & min \\ u_i & \geq & 0 \\ g_i(x_i) & \geq & 0 \\ \nabla_i f_i(x) + u_i^T \nabla_i g_i(x_i) & = & 0^T \,. \end{array} \tag{27.28}$$

If the optimal objective function value is positive, the (27.27) system doesn't have a solution, and if the optimal objective function value is zero, any optimal solution is also a solution of the (27.27) system, so the equilibrium are among the optimal solutions. We know about the sufficiency of the Kuhn-Tucker conditions that if f_i is concave in x_i with all i, the Kuhn-Tucker conditions are also sufficient, thus every solution of (27.27) gives an equilibrium.

The formal algorithm is as follows:

Kuhn–Tucker-Equilibrium

```
1  for $i \leftarrow 1$ to $I$
2       do $\mathbf{g_i} \leftarrow \nabla_i f_i$
3          $\mathbf{J_i} \leftarrow \nabla_i g_i(x_i)$
4  $(x_1, \ldots, x_I) \leftarrow$ the solution of the (27.28) optimum task
5  if $\sum_{i=1}^{I} u_i^T g_i(x_i) > 0$
6     then   return "there is no equilibrium"
7     else   return $(x_1, \ldots, x_I)$
```

Example 27.8 Assume that I production plant produce some water purification device sold into households. Let x_i denote the quantity produced by the ith production plant, let $c_i(x_i)$ be the cost function of it, and let $p(\sum_{j=1}^{I} x_j)$ be the sale price, which depends on the total quantity to be put on the market. Furthermore, be L_i is the capacity of the ith production plant. Thus, the possible X_i decision set is the $[0, L_i]$ closed interval, which can be defined by the

$$\begin{array}{rcl} x_i & \geq & 0 \\ L_i - x_i & \geq & 0 \end{array} \tag{27.29}$$

conditions, so

$$g_i(x_i) = \begin{pmatrix} x_i \\ L_i - x_i \end{pmatrix} .$$

The objective function of the ith production plant is the profit of that:

$$f_i(x_1, \ldots, x_n) = x_i p(x_1 + \cdots + x_n) - c_i(x_i) \,. \tag{27.30}$$

Since $g_i(x_i)$ is two-dimensional, u_i is a two-element vector as well, and the (27.28) optimum task has the following form:

$$\begin{array}{rcl} \sum_{i=1}^{I} (u_i^{(1)} x_i + u_i^{(2)} (L_i - x_i)) & \longrightarrow & min \\ u_i^{(1)}, u_i^{(2)} & \geq & 0 \\ x_i & \geq & 0 \\ L_i - x_i & \geq & 0 \\ p(\sum_{j=1}^{I} x_j) + x_i p'(\sum_{j=1}^{I} x_j) - c_i'(x_i) + (u_i^{(1)}, u_i^{(2)}) \begin{pmatrix} 1 \\ -1 \end{pmatrix} & = & 0 \,. \end{array} \tag{27.31}$$

Let's introduce the $\alpha_i = u_i^{(1)} - u_i^{(2)}$ new variables, and for the sake of notational convenience be $\beta_i = u_i^{(2)}$, then taking account of the last condition, we get the following problem:

$$\begin{array}{rcl} \sum_{i=1}^{I}(-x_i(p(\sum_{j=1}^{I} x_j) + x_i p'(\sum_{j=1}^{I} x_j) - c_i'(x_i)) + \beta_i L_i) & \longrightarrow & min \\ \beta_i & \geq & 0 \\ x_i & \geq & 0 \\ x_i & \leq & L_i\,. \end{array} \tag{27.32}$$

Let's notice that in case of optimum $\beta_i = 0$, so the last term of the objective function can be neglected.

Consider the special case of $I = 3$, $c_i(x_i) = ix_i^3 + x_i$, $L_i = 1$, $p(s) = 2 - 2s - s^3$. The (27.32) problem is now simplified as follows:

$$\begin{array}{rcl} \sum_{i=1}^{3} x_i(2 - 2s - s^2 - 2x_i - 2x_i s - 3ix_i^2 - 1) & \longrightarrow & max \\ x_i & \geq & 0 \\ x_i & \leq & 1 \\ x_1 + x_2 + x_3 & = & s\,. \end{array} \tag{27.33}$$

Using a simple computer program we can get the optimal solution:

$$x_1^* = 0.1077, x_2^* = 0.0986, x_3^* = 0.0919\,,$$

which is the equilibrium as well.

Exercises

27.2-1 Let $I = 2$, $X_1 = X_2 = [0,1]$, $f_1(x_1, x_2) = x_1 + x_2 - x_1^2$, $f_2(x_1, x_2) = x_1 + x_2 - x_2^2$. Formulate the (27.27) conditions and solve them as well.

27.2-2 Formulate and solve the optimum problem (27.28) for the previous exercise.

27.2-3 Let again $I = 2$. $X_1 = X_2 = [-1,1]$, $f_1(x_1, x_2) = -(x_1 + x_2)^2 + x_1 + 2x_2$, $f_2(x_1, x_2) = -(x_1 + x_2)^2 + 2x_1 + x_2$. Repeat Exercise 27.2-1.

27.2-4 Repeat Exercise 27.2-2 for the problem given in the previous exercise.

27.3. Methods of cooperative games

Similarly to the previous chapter let X_i denote again the decision set of the ith decision maker and let $x_i \in X_i$ be the concrete decision alternatives. Furthermore, let $f_i(x_1, \ldots, x_I)$ denote the objective function of the ith decision maker. Let S be some subset of the decision makers, which is usually called ***coalition*** in the game theory. For arbitrary $S \subseteq \{1, 2, \ldots, I\}$, let's introduce the

$$v(S) = \max_{x_i \in X_i} \min_{x_j \in X_j} \sum_{k \in S} f_k(x_1, \ldots, x_I) \qquad (i \in S, j \notin S) \tag{27.34}$$

function, which is also called the characteristic function defined on all of the subsets of the set $\{1, 2, \ldots, I\}$, if we add the $v(\emptyset) = 0$ and

$$v(\{1, 2, \ldots, I\}) = \max_{x_i \in X_i} \sum_{k=1}^{I} f_k(x_1, \ldots, x_I) \qquad (1 \leq i \leq I)$$

special cases to definition (27.34).

Consider again that all of the X_i sets are finite for $X_i = \{1, 2, \ldots, n_i\}$, $i = 1, 2, \ldots, I$. Be S a coalition. The value of $v(S)$ is given by the following algorithm, where $|S|$ denotes the number of elements of S, $k_1, k_2, \ldots, k_{|S|}$ denotes the elements and $l_1, l_2, \ldots, l_{I-|S|}$ the elements which are not in S.

```
Characteristic-Function($S$)
 1  $v(S) \leftarrow -M$, where $M$ a very big positive number
 2  for $i_1 \leftarrow 1$ to $n_{k_1}$
 3      $\ddots$
 4      do for $i_{|S|} \leftarrow 1$ to $n_{k_{|S|}}$
 5             do for $j_1 \leftarrow 1$ to $n_{l_1}$
 6                    $\ddots$
 7                    do for $j_{I-|S|} \leftarrow 1$ to $n_{l_{I-|S|}}$
 8                           do $Z \leftarrow M$, where $M$ a very big positive number
 9                              $V \leftarrow \sum_{t=1}^{|S|} f_{i_t}(i_1, \ldots, i_{|S|}, j_1, \ldots, j_{I-|S|})$
10                              if $V < Z$
11                                 then $Z \leftarrow V$
12             if     $Z > v(S)$
13                    then $v(S) \leftarrow Z$
14  return $v(S)$
```

Example 27.9 Let's return to the problem discussed in the previous example, and assume that $I = 3$, $L_i = 3$, $p(\sum_{i=1}^{I} x_i) = 10 - \sum_{i=1}^{I} x_i$ és $c_i(x_i) = x_i + 1$ for $i = 1, 2, 3$. Since the cost functions are identical, the objective functions are identical as well:

$$f_i(x_1, x_2, x_3) = x_i(10 - x_1 - x_2 - x_3) - (x_i + 1)\,.$$

In the following we determine the characteristic function. At first be $S = \{i\}$, then

$$v(S) = \max_{x_i} \min_{x_j} \{x_i(10 - x_1 - x_2 - x_3) - (x_i + 1)\} \qquad (j \neq i)\,.$$

Since the function strictly decreases in the $x_j (i \neq j)$ variables, the minimal value of it is given at $x_j = 3$, so

$$v(S) = \max_{i} x_i(4 - x_i) - (x_i + 1) = \max_{0 \le x_i \le 3} (-x_i^2 + 3x_i - 1) = \frac{5}{4}\,,$$

what is easy to see by plain differentiation. Similarly for $S = \{i, j\}$

$$v(S) = \max_{i,j} \min_{k \neq i,j} \{(x_i + x_j)(10 - x_1 - x_2 - x_3) - (x_i + 1) - (x_j + 1)\}\,.$$

Similarly to the previous case the minimal value is given at $x_k = 3$, so

$$v(S) = \max_{0 \le x_i, x_j \le 3} \{(x_i + x_j)(7 - x_i - x_j) - (x_i + x_j + 2)\} = \max_{0 \le x \le 6} \{x(7 - x) - (x + 2)\} =$$

$$= \max_{0 \le x \le 6} \{-x^2 + 6x - 2\} = 7$$

where we introduced the new $x = x_i + x_j$ variable. In the case of $S = \{1, 2, 3\}$

$$v(S) = \max_{0 \le x_1, x_2, x_3 \le 3} \{(x_1 + x_2 + x_3)(10 - x_1 - x_2 - x_3) - (x_1 + 1) - (x_2 + 1) - (x_3 + 1)\} =$$

$$= \max_{0 \le x \le 9} \{x(10 - x) - (x + 3)\} = \max_{0 \le x \le 9} \{-x^2 + 9x - 3)\} = 17.25\,,$$

where this time we introduced the $x = x_1 + x_2 + x_3$ variable.

Definition (27.34) can be interpreted in a way that the $v(S)$ characteristic function value gives the ***guaranteed*** aggregate objective function value of the S coalition regardless of the behavior of the others. The central issue of the theory and practice of the cooperative games is how should the certain decision makers share in the maximal aggregate profit $v(\{1, 2, \dots, I\})$ attainable together. An $(\phi_1, \phi_2, \dots, \phi_I)$ division is usually called ***imputation***, if

$$\phi_i \ge v(\{i\}) \tag{27.35}$$

for $i = 1, 2, \dots, I$ and

$$\sum_{i=1}^{I} \phi_i = v(\{1, 2, \dots, I\})\,. \tag{27.36}$$

The inequality system (27.35)–(27.36) is usually satisfied by infinite number of imputations, so we have to specify additional conditions in order to select one special element of the imputation set. We can run into a similar situation while discussing the multi-objective programming, when we looks for a special Pareto optimal solution using the concrete methods.

Example 27.10 In the previous case a (ϕ_1, ϕ_2, ϕ_3) vector is imputation if

$$\begin{aligned} \phi_1, \phi_2, \phi_3 &\ge 1.25 \\ \phi_1 + \phi_2, \phi_1 + \phi_3, \phi_2 + \phi_3 &\ge 7 \\ \phi_1 + \phi_2 + \phi_3 &= 17.2\,. \end{aligned}$$

The most popular solving approach is the ***Shapley value,*** which can be defined as follows:

$$\phi_i = \sum_{S \subseteq \{1,2,\dots,I\}} \frac{(s-1)!(I-s)!}{I!} (v(S) - v(S - \{i\}))\,, \tag{27.37}$$

where s denotes the number of elements of the S coalition.

Let's assume again that the decision makers are fully cooperating, that is they formulate the coalition {1,2,. . . ,I}, and the certain decision makers join to the coalition in random order. The difference $v(S) - v(S - \{i\})$ indicates the contribution to the S coalition by the ith decision maker, while expression (27.37) indicates the average contribution of the same decision maker. It can be shown that $(\phi_1, \phi_2, \dots, \phi_I)$ is an imputation.

The Shapley value can be computed by following algorithm:

Shapley-Value

```
1  for $\forall S \subseteq \{1, \dots, I\}$
2      do $v(S) \leftarrow$ Characteristic-Function($S$)
3  for $i \leftarrow 1$ to $I$
4      do use (27.37) for calculating $\phi_i$
```

Example 27.11 In the previous example we calculated the value of the characteristic function. Because of the symmetry, $\phi_1 = \phi_2 = \phi_3$ must be true for the case of Shapley value. Since $\phi_1 + \phi_2 + \phi_3 = v(\{1,2,3\}) = 17.25, \phi 1 = \phi_2 = \phi_3 = 5.75$. We get the same value by formula (27.37) too. Let's consider the ϕ_1 value. If $i \notin S$, $v(S) = v(S - \{i\})$, so the appropriate terms of the sum are zero-valued. The non-zero terms are given for coalitions $S = \{1\}, S = \{1,2\}, S = \{1,3\}$ and $S = \{1,2,3\}$, so

$$\phi_1 = \frac{0!2!}{3!}(\frac{5}{4} - 0) + \frac{1!1!}{3!}(7 - \frac{5}{4}) + \frac{1!1!}{3!}(7 - \frac{5}{4}) + \frac{2!0!}{3!}(\frac{69}{4} - 7) =$$

$$\frac{1}{6}(\frac{10}{4} + \frac{23}{4} + \frac{23}{3} + \frac{82}{4}) = \frac{138}{24} = 5.75\,.$$

An alternative solution approach requires the stability of the solution. It is said that the vector $\boldsymbol{\phi} = (\phi_1, \dots, \phi_I)$ majorizes the vector $\boldsymbol{\psi} = (\psi_1, \dots, \psi_I)$ in coalition S, if

$$\sum_{i \in S} \phi_i > \sum_{i \in S} \psi_i\,,$$

that is the S coalition has an in interest to switch from payoff vector $\boldsymbol{\phi}$ to payoff vector $\boldsymbol{\psi}$, or $\boldsymbol{\psi}$ is instabil for coalition S. The ***Neumann–Morgenstern solution*** is a V set of imputations for which

(i) There is no $\boldsymbol{\phi}, \boldsymbol{\psi} \in V$, that $\boldsymbol{\phi}$ majorizes $\boldsymbol{\psi}$ in some coalition (inner stability)

(ii) If $\boldsymbol{\psi} \notin V$, there is $\boldsymbol{\phi} \in V$, that $\boldsymbol{\phi}$ majorizes $\boldsymbol{\psi}$-t in at least one coalition (outer stability).

The main difficulty of this conception is that there is no general existence theorem for the existence of a non-empty V set, and there is no general method for constructing the set V.

Exercises

27.3-1 Let $I = 3$, $X_1 = X_2 = X_3 = [0,1]$, $f_i(x_1, x_2, x_3) = x_1 + x_2 + x_3 - x_i^2$ $(i = 1,2,3)$. Determine the $v(S)$ characteristic function.
27.3-2 Formulate the (27.35), (27.36) condition system for the game of the previous exercise.
27.3-3 Determine the ψ_i Shapley values for the game of Exercise 27.3-1.

27.4. Collective decision-making

In the previous chapter we assumed that the objective functions are given by numerical values. These numerical values also mean preferences, since the ith decision maker prefers alternative x to z, if $f_i(x) > f_i(z)$. In this chapter we will discuss such methods which don't require the knowledge of the objective functions, but the preferences of the certain decision makers.

Let I denote again the number of decision makers, and X the set of decision alternatives. If the ith decision maker prefers alternative x to y, this is denoted by $x \succ_i y$, if prefers alternative x to y or thinks to be equal, it is denoted by $x \succeq_i y$. Assume that

(i) For all $x, y \in X$, $x \succeq_i y$ or $y \succeq_i x$ (or both)

(ii) For $x \succeq_i y$ and $y \succeq_i z$, $x \succeq_i z$.

Condition (i) requires that the $\succeq_i$ partial order be a total order, while condition (ii) requires to be transitive.

Definition 27.5 *A group decision-making function combines arbitrary individual* $(\succeq_1, \succeq_2, \dots, \succeq_I)$ *partial orders into one partial order, which is also called the collective preference structure of the group.*

We illustrate the definition of group decision-making function by some simple example.

Example 27.12 Be $x, y \in X$ arbitrary, and for all i

$$\alpha_i = \begin{cases} 1, & \text{ha } x \succ_i y, \\ 0, & \text{ha } x \sim_i y, \\ -1, & \text{ha } x \prec_i y. \end{cases}$$

Let $\beta_i, \beta_2, \dots, \beta_I$ given positive constant, and

$$\alpha = \sum_{i=1}^{I} \beta_i \alpha_i \,.$$

The group decision-making function means:

$$\begin{aligned} x \succ y &\iff \alpha > 0 \\ x \sim y &\iff \alpha = 0 \\ x \prec y &\iff \alpha < 0 \,. \end{aligned}$$

The ***majority rule*** is a special case of it when $\beta_1 = \beta_2 = \cdots = \beta_I = 1$.

Example 27.13 An i_0 decision maker is called ***dictator***, if his or her opinion prevails in group decision-making:

$$\begin{aligned} x \succ y &\iff x \succ_{i_0} y \\ x \sim y &\iff x \sim_{i_0} y \\ x \prec y &\iff x \prec_{i_0} y \,. \end{aligned}$$

This kind of group decision-making is also called dictatorship.

Example 27.14 In the case of ***Borda measure*** we assume that α is a finite set and the preferences of the decision makers is expressed by a $c_i(x)$ measure for all $x \in X$. For example $c_i(x) = 1$, if x is the best, $c_i(x) = 2$, if x is the second best alternative for the ith decision maker, and so on, $c_i(x) = I$, if x is the worst alternative. Then

$$\begin{aligned} x \succ y \quad &\Longleftrightarrow \quad \sum_{i=1}^{I} c_i(x) > \sum_{i=1}^{I} c_i(y) \\ x \sim y \quad &\Longleftrightarrow \quad \sum_{i=1}^{I} c_i(x) = \sum_{i=1}^{I} c_i(y) \\ x \prec y \quad &\Longleftrightarrow \quad \sum_{i=1}^{I} c_i(x) < \sum_{i=1}^{I} c_i(y)\,. \end{aligned}$$

A group decision-making function is called ***Pareto*** or ***Pareto function***, if for all $x, y \in X$ and $x \succ_i y\,(i = 1, 2, \ldots, I)$, $x \succ y$ necessarily. That is, if all the decision makers prefer x to y, it must be the same way in the collective preference of the group. A group decision-making function is said to satisfy the condition of ***pairwise independence***, if any two $(\succeq_1, \ldots, \succeq_I)$ and $(\succeq'_1, \ldots, \succeq'_I)$ preference structure satisfy the followings. Let $x, y \in X$ such that for arbitrary i, $x \succeq_i y$ if and only if $x \succeq'_i y$, and $y \succeq_i x$ if and only if $y \succeq'_i x$. Then $x \succeq y$ if and only if $x \succeq' y$, and $y \succeq x$ if and only if $y \succeq' x$ in the collective preference of the group.

Example 27.15 It is easy to see that the Borda measure is Pareto, but it doesn't satisfy the condition of pairwise independence. The first statement is evident, while the second one can be illustrated by a simple example. Be $I = 2$, $\alpha = \{x, y, z\}$. Let's assume that

$$\begin{aligned} &x \succ_1 z \succ_1 y \\ &y \succ_2 x \succ_2 z \end{aligned}$$

and

$$\begin{aligned} &x \succ'_1 y \succ'_1 z \\ &y \succ'_2 z \succ'_2 x \quad . \end{aligned}$$

Then $c(x) = 1 + 2 = 3, c(y) = 3 + 1 = 4$, thus $y \succ x$. However $c'(x) = 1 + 3 = 4, c'(y) = 2 + 1 = 3$, so $x \succ y$. As we can see the certain decision makers preference order between x and y is the same in both case, but the collective preference of the group is different.

Let $\mathbb{R}_I$ denote the set of the I-element full and transitive partial orders on an at least three-element X set, and be $\preceq$ the collective preference of the group which is Pareto and satisfies the condition of pairwise independence. Then $\preceq$ is necessarily dictatorial. This result originated with Arrow shows that there is no such group decision-making function which could satisfy these two basic and natural requirements.

Decision makers	Alternatives				Weights
	1	2	...	N	
1	a_{11}	a_{12}	...	a_{1N}	α_1
2	a_{21}	a_{22}	...	a_{2N}	α_2
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
I	a_{I1}	a_{I2}	...	a_{IN}	α_I

Figure 27.11 Group decision-making table.

Example 27.16 The method of ***paired comparison*** is as follows. Be $x, y \in X$ arbitrary, and let's denote $P(x, y)$ the number of decision makers, to which $x \succ_i y$. After that, the collective preference of the group is the following:

$$\begin{aligned} x \succ y &\iff P(x,y) > P(y,x) \\ x \sim y &\iff P(x,y) = P(y,x) \\ x \prec y &\iff P(x,y) < P(y,x)\,, \end{aligned}$$

that is $x \succ y$ if and only if more than one decision makers prefer the x alternative to y. Let's assume again that X consists of three elements, $X = \{x, y, z\}$ and the individual preferences for $I = 3$

$$\begin{aligned} &x \succ_1 y \succ_1 z \\ &z \succ_2 x \succ_2 y \\ &y \succ_3 z \succ_3 x \quad . \end{aligned}$$

Thus, in the collective preference $x \succ y$, because $P(x, y) = 2$ and $P(y, x) = 1$. Similarly $y \succ z$, because $P(y, z) = 2$ and $P(z, y) = 1$, and $z \succ x$, since $P(z, x) = 2$ and $P(x, z) = 1$. Therefore $x \succ y \succ z \succ x$ which is inconsistent with the requirements of transitivity.

The methods discussed so far didn't take account of the important circumstance that the decision makers aren't necessarily in the same position, that is they can have different importance. This importance can be characterized by weights. In this generalized case we have to modify the group decision-making methods as required. Let's assume that X is finite set, denote N the number of alternatives. We denote the preferences of the decision makers by the numbers ranging from 1 to N, where 1 is assigned to the most favorable, while N is assigned to most unfavorable alternative. It's imaginable that the two alternatives are equally important, then we use fractions. For example, if we can't distinguish between the priority of the 2nd and 3rd alternatives, then we assign 2.5 to each of them. Usually the average value of the indistinguishable alternatives is assigned to each of them. In this way, the problem of the group decision can be given by a table which rows correspond to the decision makers and columns correspond to the decision alternatives. Every row of the table is a permutation of the $1, 2, \ldots, N$ numbers, at most some element of it is replaced by some average value if they are equally-preferred. Figure 27.11 shows the given table in which the last column contains the weights of the decision makers.

In this general case the ***majority rule*** can be defined as follows. For all of the j

alternatives determine first the aggregate weight of the decision makers to which the alternative j is the best possibility, then select that alternative for the best collective one for which this sum is the biggest. If our goal is not only to select the best, but to rank all of the alternatives, then we have to choose descending order in this sum to rank the alternatives, where the biggest sum selects the best, and the smallest sum selects the worst alternative. Mathematically, be

$$f(a_{ij}) = \begin{cases} 1, & \text{ha } \ a_{ij} = 1, \\ 0 & \text{otherwise} \end{cases} \tag{27.38}$$

and

$$A_j = \sum_{i=1}^{I} f(a_{ij})\alpha_i \tag{27.39}$$

for $j = 1, 2, \ldots, I$. The j_0th alternative is considered the best by the group, if

$$A_{j_0} = \max_j \{A_j\} . \tag{27.40}$$

The formal algorithm is as follows:

```
MAJORITY-RULE($A$)
1  $A_1 \leftarrow 0, A_2 \leftarrow 0, \ldots, A_N \leftarrow 0, max \leftarrow 0$
2  for $i \leftarrow 1$ to $N$
3      do for $j \leftarrow 1$ to $I$
4              do if  $a_{ji} = 1$
5                     then $A_i \leftarrow A_i + \alpha_j$
6      if  $A_i > max$
7          then  $max \leftarrow A_i$
8                $ind \leftarrow i$
9  return $ind$
```

Applying the ***Borda measure***, let

$$B_j = \sum_{i=1}^{I} a_{ij}\alpha_i , \tag{27.41}$$

and alternative j_0 is the result of the group decision if

$$B_{j_0} = \min_j \{B_j\} . \tag{27.42}$$

The Borda measure can be described by the following algorithm:

Borda-Measure-Method(A, α)

```
1  $B_1 \leftarrow 0, B_2 \leftarrow 0, \ldots, B_N \leftarrow 0, max \leftarrow 0$
2  for $j \leftarrow 1$ to $N$
3      do for $i \leftarrow 1$ to $I$
4             do $B_j \leftarrow B_j + a_{ij}\alpha_i$
5      if  $B_j > max$
6          then  $max \leftarrow B_j$
7                $ind \leftarrow j$
8  return $ind$
```

Applying the method of ***paired comparison***, let with any $j, j' \in X$

$$P(j, j') = \sum_{\{i|a_{ij}<a_{ij'}\}} \alpha_i \tag{27.43}$$

which gives the weight of the decision makers who prefer the alternative j to j'. In the collective decision

$$j \succ j' \iff P(j, j') > P(j', j) \,.$$

In many cases the collective partial order given this way doesn't result in a clearly best alternative. In such cases further analysis (for example using some other method) need on the

$$S^* = \{j|j \in X \text{ and theres is no such } j' \in X, \text{ for which } j' \succ j\}$$

non-dominated alternative set.

By this algorithm we construct a matrix consists of the $\{0, 1\}$ elements, where $a_{jl} = 1$ if and only if the j alternative is better in all then alternative l. In the case of draw $a_{jl} = \frac{1}{2}$.

Paired-Comparison(A)

```
1  for $j \leftarrow 1$ to $N - 1$
2      do for $l \leftarrow j$ to $N$
3             do $z \leftarrow 0$
4                for $i \leftarrow 1$ to $I$
5                    do if  $a_{ij} > a_{il}$
6                           then  $z \leftarrow z + 1$
7                if  $z > \frac{N}{2}$
8                    then  $a_{jl} \leftarrow 1$
9                if  $z = \frac{N}{2}$
10                   then  $a_{jl} \leftarrow \frac{1}{2}$
11               if  $z < \frac{N}{2}$
12                   then  $a_{jl} \leftarrow 0$
13               $a_{lj} \leftarrow a_{jl}$
14 return $A$
```

Example 27.17 Four proposal were received by the Environmental Authority for the

Committee Members	Alternatives 1	2	3	4	Weights
1	1	3	2	4	0.3
2	2	1	4	3	0.2
3	1	3	2	4	0.2
4	2	3	1	4	0.1
5	3	1	4	2	0.1
6	1	4	2	3	0.1

Figure 27.12 The database of Example 27.17

cleaning of a chemically contaminated site. A committee consists of 6 people has to choose the best proposal and thereafter the authority can conclude the contract for realizing the proposal. Figure 27.12 shows the relative weight of the committee members and the personal preferences.

Majority rule

Using the ***majority rule***

$$\begin{aligned} A_1 &= 0.3 + 0.2 + 0.1 = 0.6 \\ A_2 &= 0.2 + 0.1 = 0.3 \\ A_3 &= 0.1 \\ A_4 &= 0\,, \end{aligned}$$

so the first alternative is the best.

Using the ***Borda measure***

$$\begin{aligned} B_1 &= 0.3 + 0.4 + 0.2 + 0.2 + 0.3 + 0.1 = 1.5 \\ B_2 &= 0.9 + 0.2 + 0.6 + 0.3 + 0.1 + 0.4 = 2.5 \\ B_3 &= 0.6 + 0.8 + 0.4 + 0.1 + 0.4 + 0.2 = 2.5 \\ B_4 &= 1.2 + 0.6 + 0.8 + 0.4 + 0.2 + 0.3 = 3.5\,. \end{aligned}$$

In this case the first alternative is the best as well, but this method shows equally good the second and third alternatives. Notice, that in the case of the previous method the second alternative was better than the third one.

In the case of the method of ***paired comparison***

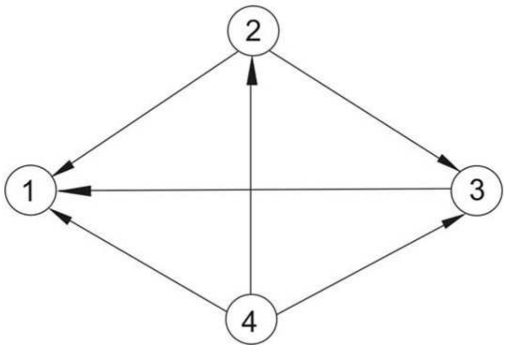


Figure 27.13 The preference graph of Example 27.17

$$\begin{aligned}
P(1,2) &= 0.3+0.2+0.1+0.1=0.7\\
P(2,1) &= 0.2+0.1=0.3\\
P(1,3) &= 0.3+0.2+0.2+0.1+0.1=0.9\\
P(3,1) &= 0.1\\
P(1,4) &= 0.3+0.2+0.2+0.1+0.1=0.9\\
P(4,1) &= 0.1\\
P(2,3) &= 0.2+0.1+0.1=0.4\\
P(3,2) &= 0.3+0.2+0.1=0.6\\
P(2,4) &= 0.3+0.2+0.2+0.1+0.1=0.9\\
P(4,2) &= 0.1\\
P(3,4) &= 0.3+0.2+0.1+0.1=0.7\\
P(4,3) &= 0.2+0.1=0.3\,.
\end{aligned}$$

Thus $1 \succ 2, 1 \succ 3, 1 \succ 4, 3 \succ 2, 2 \succ 4$ and $3 \succ 4$. These references are showed by Figure 27.13. The first alternative is better than any others, so this is the obvious choice.

In the above example all three methods gave the same result. However, in several practical cases one can get different results and the decision makers have to choose on the basis of other criteria.

Exercises

27.4-1 Let's consider the following group decision-making table:
Apply the majority rule.

27.4-2 Apply the Borda measure to the previous exercise.

27.4-3 Apply the method of paired comparison to Exercise 27.4-1.

27.4-4 Let's consider now the following group decision-making table:
Repeat Exercise 27.4-1 for this exercise.

27.4-5 Apply the Borda measure to the previous exercise.

27.4-6 Apply the method of paired comparison to Exercise 27.4-4.

Decision makers	Alternatives					Weights
	1	2	3	4	5	
1	1	3	2	5	4	3
2	1	4	5	2	3	2
3	5	4	1	3	2	2
4	4	3	2	1	5	1

Figure 27.14 Group decision-making table

Decision makers	Alternatives			Weights
	1	2	3	
1	1	2	3	1
2	3	2	1	1
3	2	1	3	1
4	1	3	2	1

Figure 27.15 Group decision-making table

27.5. Applications of Pareto games

Let I denote again the number of decision makers but suppose now that the decision makers have more than one objective functions separately. There are several possibility to handle such problems:

(A) In the application of multi-objective programming, let α_i denote the weight of the ith decision maker, and let $\beta_{i1}, \beta_{i2}, \ldots, \beta_{ic(i)}$ be the weights of this decision maker's objective functions. Here $c(i)$ denote the number of the ith decision maker's objective functions. Thus we can get an optimization problem with the $\sum_{i=1}^{I} c(i)$ objective function, where all of the decision makers' all the objective functions mean the objective function of the problem, and the weights of the certain objective functions are the $\alpha_i \beta_{ij}$ sequences. We can use any of the methods from Chapter 27.1. to solve this problem.

(B) We can get another family of methods in the following way. Determine an utility function for every decision maker (as described in Chapter 27.1.1.), which compresses the decision maker's preferences into one function. In the application of this method every decision maker has only one (new) objective function, so any methods and solution concepts can be used from the previous chapters.

(C) A third method can be given, if we determine only the partial order of the certain decision makers defined on an alternative set by some method instead of the construction of utility functions. After that we can use any method of Chapter 27.4. directly.

Decision maker	Objective function	Alternatives				Objective function	Decisi
		1	2	3	4	weight	w
1	1	90	75	80	85	0.5	
	2	0.9	0.8	0.7	0.8	0.5	
2	1	85	80	70	90	0.6	
	2	0.8	0.9	0.8	0.85	0.4	
3	1	80	90	75	70	0.7	
	2	0.85	0.8	0.9	0.8	0.3	

Figure 27.16 The database of Example 27.18

Example 27.18 Modify the previous chapter as follows. Let's suppose again that we choose from four alternatives, but assume now that the committee consists of three people and every member of it has two objective functions. The first objective function is the technical standards of the proposed solution on a subjective scale, while the second one are the odds of the exact implementation. The latter one is judged subjectively by the decision makers individually by the preceding works of the supplier. The data is shown in Figure 27.16., where we assume that the first objective function is judged on a subjective scale from 0 to 100, so the normalized objective function values are given dividing by 100. Using the weighting method we get the following aggregate utility function values for the separate decision makers:

1. Decision maker

$$\begin{array}{lrcl}
\text{First alternative:} & 0.9(0.5)+0.9(0.5) & = & 0.9 \\
\text{Second alternative:} & 0.75(0.5)+0.8(0.5) & = & 0.775 \\
\text{Third alternative:} & 0.8(0.5)+0.7(0.5) & = & 0.75 \\
\text{Fourth alternative:} & 0.85(0.5)+0.8(0.5) & = & 0.825
\end{array}$$

2. Decision maker

$$\begin{array}{lrcl}
\text{First alternative:} & 0.85(0.6)+0.8(0.4) & = & 0.83 \\
\text{Second alternative:} & 0.8(0.6)+0.9(0.4) & = & 0.84 \\
\text{Third alternative:} & 0.7(0.6)+0.8(0.4) & = & 0.74 \\
\text{Fourth alternative:} & 0.9(0.6)+0.85(0.4) & = & 0.88
\end{array}$$

3. Decision maker

$$\begin{array}{lrcl}
\text{First alternative:} & 0.8(0.7)+0.85(0.3) & = & 0.815 \\
\text{Second alternative:} & 0.9(0.7)+0.8(0.3) & = & 0.87 \\
\text{Third alternative:} & 0.75(0.7)+0.9(0.3) & = & 0.795 \\
\text{Fourth alternative:} & 0.7(0.7)+0.8(0.3) & = & 0.73
\end{array}$$

The preferences thus are the following:

$$1 \succ_1 4 \succ_1 2 \succ_1 3, 4 \succ_2 2 \succ_2 1 \succ_2 3, \quad \text{and} \quad 2 \succ_3 1 \succ_3 3 \succ_3 4\,.$$

For example, in the application of Borda measure

$$\begin{aligned} B_1 &= 1(0.4) + 3(0.3) + 2(0.3) = 1.9 \\ B_2 &= 3(0.4) + 2(0.3) + 1(0.3) = 2.1 \\ B_3 &= 4(0.4) + 4(0.3) + 3(0.3) = 3.7 \\ B_4 &= 2(0.4) + 1(0.3) + 4(0.3) = 2.3 \end{aligned}$$

are given, so the group-order of the four alternatives

$$1 \succ 2 \succ 4 \succ 3\,.$$

Exercises

27.5-1 Let's consider the following table:

Decision maker	Objective function	Alternatives			Objective function	Decision mak
		1	2	3	weight	weight
1	1	0.6	0.8	0.7	0.6	0.5
	2	0.9	0.7	0.6	0.4	
2	1	0.5	0.3	0.4	0.5	0.25
	2	0.6	0.8	0.7	0.5	
3	1	0.4	0.5	0.6	0.4	0.25
	2	0.7	0.6	0.6	0.4	
	3	0.5	0.8	0.6	0.2	

Figure 27.17

Let's consider that the objective functions are already normalized. Use method (A) to solve the exercise.

27.5-2 Use method (B) for the previous exercise, where the certain decision makers' utility functions are given by the weighting method, and the group decision making is given by the Borda measure.

27.5-3 Solve Exercise 27.5-2 using the method of paired comparison instead of Borda measure.

27.6. Axiomatic methods

For the sake of simplicity, let's consider that $I = 2$, that is we'd like to solve the conflict between two decision makers. Assume that the consequential space H is convex, bounded and closed in $\mathbb{R}^2$, and there is given a $\mathbf{f}_* = (f_{1*}, f_{2*})$ point which gives the objective function values of the decision makers in cases where they are unable to agree. We assume that there is such $\mathbf{f} \in H$ that $\mathbf{f} > \mathbf{f}_*$. The conflict is characterized by the $(H, \mathbf{f}_*)$ pair. The solution obviously has to depend on both H

and $\mathbf{f}_*$, so it is some function of them: $\phi(H, \mathbf{f}_*)$.

For the case of the different solution concepts we demand that the solution function satisfies some requirements which treated as axioms. These axioms require the correctness of the solution, the certain axioms characterize this correctness in different ways.

In the case of the ***classical Nash*** solution we assume the following:

(i) $\phi(H, \mathbf{f}_*) \in H$ (possibility)

(ii) $\phi(H, \mathbf{f}_*) \geq \mathbf{f}_*$ (rationality)

(iii) $\phi(H, \mathbf{f}_*)$ is Pareto solution in H (Pareto optimality)

(iv) If $H_1 \subseteq H$ and $\phi(H, \mathbf{f}_*) \in H_1$, necessarily $\phi(H_1, \mathbf{f}_*) = \phi(H, \mathbf{f}_*)$ (independence of irrelevant alternatives)

(v) Be $T : \mathbb{R}^2 \mapsto \mathbb{R}^2$ such linear transformation that $T(f_1, f_2) = (\alpha_1 f_1 + \beta_1, \alpha_2 f_2 + \beta_2)$ is positive for α_1 and α_2. Then $\phi(T(H), T(\mathbf{f}_*)) = T(\phi(H, \mathbf{f}_*))$ (invariant to affine transformations)

(vi) If H and $\mathbf{f}_*$ are symmetrical, that is $f_{1*} = f_{2*}$ and $(f_1, f_2) \in H \iff (f_2, f_1) \in H$, then the components of $\phi(H, \mathbf{f}_*)$ be equals (symmetry).

Condition (i) demands the possibility of the solution. Condition (ii) requires that none of the rational decision makers agree on a solution which is worse than the one could be achieved without consensus. On the basis of condition (iii) there is no better solution than the friendly solution. According to requirement (iv), if after the consensus some alternatives lost their possibility, but the solution is still possible, the solution remains the same for the reduced consequential space. If the dimension of any of the objective functions changes, the solution can't change. This is required by (v), and the last condition means that if two decision makers are in the same absolutely same situation defining the conflict, we have to treat them in the same way in the case of solution. The following essential result originates from Nash:

Theorem 27.6 *The (i)-(vi) conditions are satisfied by exactly one solution function, and $\phi(H, \mathbf{f}_*)$ can be given by as the*

$$\begin{array}{rcll} (f_1 - f_{1*})(f_2 - f_{2*}) & \longrightarrow & max & ((f_1, f_2) \in H) \\ f_1 & \geq & f_{1*} & \\ f_2 & \geq & f_{2*} & \end{array} \tag{27.44}$$

optimum problem unique solution.

Example 27.19 Let's consider again the consequential space showed in Figure 27.3 before, and suppose that $(f_{1*}, f_{2*}) = (0, -1)$, that is it comprises the worst values in its components. Then Exercise (27.44) is the following:

$$\begin{array}{rcl} f_1(f_2 + 1) & \longrightarrow & max \\ f_2 & \leq & f_1 \\ f_2 & \leq & 3 - 2f_1 \\ f_2 & \geq & -\frac{1}{2} f_1 \,. \end{array}$$

It's easy to see that the optimal solution is $f_1 = f_2 = 1$.

Notice that problem (27.44) is a distance dependent method, where we maximize the geometric distance from the (f_{1*}, f_{2*}) point. The algorithm is the solution of the (27.44) optimum problem.

Condition (vi) requires that the two decision makers must be treated equally. However in many practical cases this is not an actual requirement if one of them is in stronger position than the other.

Theorem 27.7 *Requirements (i)-(v) are satisfied by infinite number of functions, but every solution function comprises such* $0 \le \alpha \le 1$, *that the solution is given by as the*

$$\begin{array}{rcll} (f_1 - f_{1*})\alpha(f_2 - f_{2*})^{1-\alpha} & \longrightarrow & max & ((f_1, f_2) \in H) \\ f_1 & \ge & f_{1*} & \\ f_2 & \ge & f_{2*} & \end{array} \tag{27.45}$$

optimum problem unique solution.

Notice that in the case of $\alpha = \frac{1}{2}$, problem (27.45) reduces to problem (27.44). The algorithm is the solution of the (27.45) optimum problem.

Many author criticized Nash's original axioms, and beside the modification of the axiom system, more and more new solution concepts and methods were introduced. Without expose the actual axioms, we will show the methods judged to be of the utmost importance by the literature.

In the case of the ***Kalai–Smorodinsky solution*** we determine firstly the ideal point, which coordinates are:

$$f_i^* = \max\{f_i | (f_1, f_2) \in H, (f_1, f_2) \ge \mathbf{f}_*\} ,$$

then we will accept the last mutual point of the half-line joining $\mathbf{f}_*$ to the ideal point and H as solution. Figure 27.18. shows the method. Notice that this is an direction dependent method, where the half-line shows the direction of growing and $\mathbf{f}_*$ is the chosen start point.

The algorithm is the solution of the following optimum problem.

$$t \longrightarrow max$$

provided that

$$\mathbf{f}_* + t(\mathbf{f}^* - \mathbf{f}_*) \in H .$$

Example 27.20 In the case of the previous example $\mathbf{f}_* = (0, -1)$ and $\mathbf{f}^* = (2, 1)$. We can see in Figure 27.19, that the last point of the half-line joining $\mathbf{f}_*$ to $\mathbf{f}^*$ in H is the intersection point of the half-line and the section joining $(1, 1)$ to $(2, -1)$.

The equation of the half-line is

$$f_2 = f_1 - 1 ,$$

while the equation of the joining section is

$$f_2 = -2f_1 + 3 ,$$

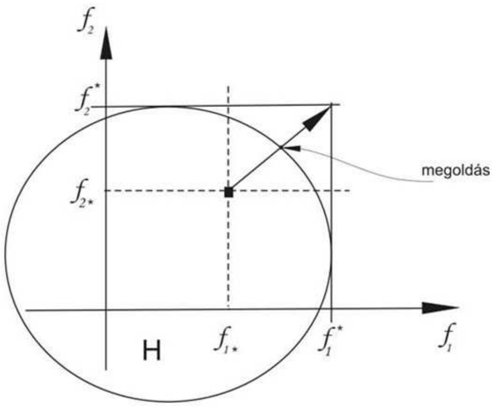


Figure 27.18 Kalai–Smorodinsky solution.

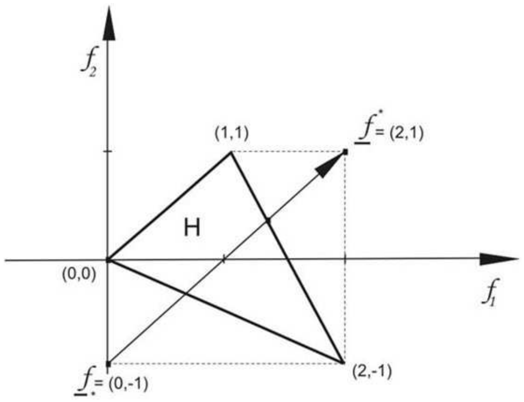


Figure 27.19 Solution of Example 27.20

so the intersect point: $f_1 = \frac{4}{3}, f_2 = \frac{1}{3}$.

In the case of the ***equal-loss method*** we assume, that starting from the ideal point the two decision makers reduce the objective function values equally until they find a possible solution. This concept is equivalent to the solution of the

$$t \longrightarrow min \qquad ((f_1^* - t, f_2^* - t) \in H) \tag{27.46}$$

optimum problem. Let t^* denote the minimal t value, then the $(f_1^* - t^*, f_2^* - t^*)$ point is the solution of the conflict. The algorithm is the solution of the (27.46) optimum problem.

Example 27.21 In the case of the previous example $\mathbf{f}^* = (2, 1)$, so starting from this point going by the 45° line, the first possible solution is the $f_1 = \frac{4}{3}, f_2 = \frac{1}{3}$ point again.

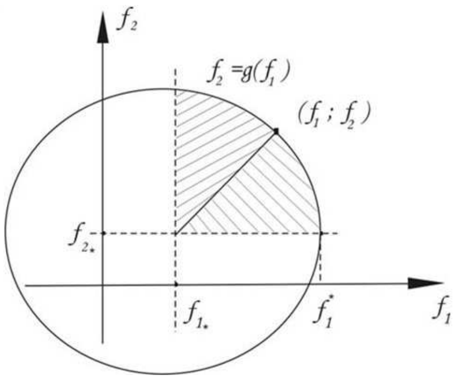


Figure 27.20 The method of monotonous area.

In the case of the method of ***monotonous area*** the (f_1, f_2) solution is given by as follows. The linear section joining (f_{1*}, f_{2*}) to (f_1, f_2) divides the set H into two parts, if (f_1, f_2) is a Pareto optimal solution. In the application of this concept we require the two areas being equal. Figure 27.20 shows the concept. The two areas are given by as follows:

$$\int_{f_{1*}}^{f_1} (g(t) - f_{2*})dt - \frac{1}{2}(f_1 - f_{1*})(g(f_1) - f_{2*})$$

and

$$\frac{1}{2}(f_1 - f_{1*})(g(f_1) - f_{2*}) + \int_{f_1}^{f_1^*} (g(t) - f_2^*)dt$$

where we suppose that $f_2 = g(f_1)$ defines the graph of the Pareto optimal solution. Thus we get a simple equation to determine the unknown value of f_1.

The algorithm is the solution of the following nonlinear, univariate equation:

$$\int_{f_{1*}}^{f_1} (g(t) - f_{2*})dt - \int_{f_1}^{f_{1*}} (g(t) - f_2^*)dt - (f_1 - f_{1*})(g(f_1) - f_{2*}) = 0\,.$$

Any commonly known (bisection, secant, Newton's method) method can be used to solve the problem.

Exercises

27.6-1 Consider that $H = \{(f_1, f_2)|f_1, f_2 \geq 0, f_1 + 2f_2 \leq 4\}$. Be $f_{1*} = f_{2*} = 0$. Use the (27.44) optimum problem.

27.6-2 Assume that the two decision makers are not equally important in the previous exercise. $\alpha = \frac{1}{3}, 1 - \alpha = \frac{2}{3}$. Solve the (27.45) optimum problem.

27.6-3 Use the Kalai–Smorodinsky solution for Exercise 27.6-1

27.6-4 Use the equal-loss method for Exercise 27.6-1

27.6-5 Use the method of monotonous area for Exercise 27.6-1

Problems

27-1 Első feladat címe
Prove that the solution of problem (27.9) is Pareto optimal for any positive $\alpha_1, \alpha_2, \ldots, \alpha_I$ values.

27-2 Masodik feladat címe
Prove that the distance dependent methods always give Pareto optimal solution for ϱ_1. Is it also true for ϱ_∞?

27-3 Harmadik feladat címe
Find a simple example for which the direction dependent methods give non Pareto optimal solution.

27-4 Negyedik feladat címe
Suppose in addition to the conditions of 27.4. that all of the f_i functions are strictly concave in x_i. Give an example for which there are more than one equilibrium.

27-5 Ötödik feladat címe
Prove that the Shapley values result imputation and satisfy the (27.35)–(27.36) conditions.

27-6 Hatodik feladat címe
Solve such a group decision making table where the method of paired comparison doesn't satisfy the requirement of transitivity. That is there are such i, j, k alternatives for which $i \succ j$, $j \succ k$, but $k \succ i$.

27-7 Hetedik feladat címe
Construct such an example, where the application of Borda measure equally qualifies all of the alternatives.

27-8 Nyolcadik feladat címe
Prove that using the Kalai–Smorodinsky solution for non convex H, the solution is not necessarily Pareto optimal.

27-9 Kilencedik feladat címe
Show that for non convex H, neither the equal-loss method nor the method of monotonous area can guarantee Pareto optimal solution.

Chapter Notes

Readers interested in multi-objective programming can find addition details and methods related to the topic in the [142] book. There are more details about the method of equilibrium and the solution concepts of the cooperative games in the [50] monograph. The [147] monograph comprises additional methods and formulas from the methodology of group decision making. Additional details to Theorem 27.6 originates from Hash can be found in [109]. One can read more details about the weakening of the conditions of this theorem in [61]. Details about the Kalai–Smorodinsky solution, the equal-loss method and the method of monotonous area can found respectively in [81], [31] and [2]. Note finally that the [149] summary paper discuss the axiomatic introduction and properties of these and other newer methods.

The results discussed in this chapter can be found in the book of Molnár Sándor and Szidarovszky Ferenc [104] in details.

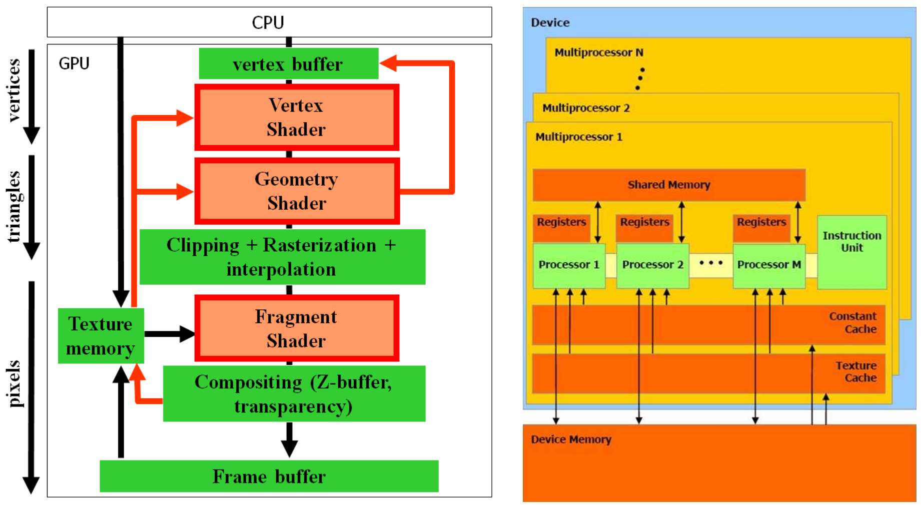
28. General Purpose Computing on Graphics Processing Units

GPGPU stands for **G**eneral-**P**urpose computation on **G**raphics **P**rocessing **U**nits, also known as GPU Computing. Graphics Processing Units (***GPU***) are highly parallel, multithreaded, manycore processors capable of very high computation and data throughput. Once specially designed for computer graphics and programmable only through graphics APIs, today's GPUs can be considered as general-purpose parallel processors with the support for accessible programming interfaces and industry-standard languages such as C.

Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations. This difference comes from the high floating point performance and peak memory bandwidth of GPUs. This is because the GPU is specialized for compute-intensive, highly parallel computation—exactly what graphics rendering is about—and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. From the developer's point of view this means that hardware latencies are not hidden, they must be managed explicitly, and writing an efficient GPU program is not possible without the knowledge of the architecture.

Another reason of discussing GPGPU computing as a specific field of computer science is that although a GPU can be regarded as a parallel system, its architecture is not a clean implementation of parallel computing models (see Chapter 15 of this book titled *Parallel Computations*). Instead, it has the features of many different models, like pipelines, vector or array processors, ***Single-Instruction Multiple-Data*** (***SIMD***) machines, stream-processors, multi-processors connected via shared memory, hard-wired algorithms, etc. So, when we develop solutions for this special architecture, the ideas applicable for many different architectures should be combined in creative ways.

GPUs are available as graphics cards, which must be mounted into computer systems, and a runtime software package must be available to drive the computations. A graphics card has programmable processing units, various types of memory and cache, and fixed-function units for special graphics tasks. The hardware operation must be controlled by a program running on the host computer's CPU through ***Application Programming Interfaces*** **(*API*).** This includes uploading programs to GPU units and feeding them with data. Programs might be written and compiled



Graphics API programming model CUDA programming model

Figure 28.1 GPU programming models for shader APIs and for CUDA. We depict here a Shader Model 4 compatible GPU. The programmable stages of the shader API model are red, the fixed-function stages are green.

from various programming languages, some originally designed for graphics (like ***Cg*** [111] or ***HLSL*** [102]) and some born by the extension of generic programming languages (like CUDA C). The programming environment also defines a ***programming model*** or ***virtual parallel architecture*** that reflects how programmable and fixed-function units are interconnected. Interestingly, different programming models present significantly different virtual parallel architectures (Figure 28.1). Graphics APIs provide us with the view that the GPU is a pipeline or a stream-processor since this is natural for most of the graphics applications. ***CUDA*** [112] or ***OpenCL*** [83], on the other hand, gives the illusion that the GPU is a collection of multiprocessors where every multiprocessor is a wide SIMD processor composed of scalar units, capable of executing the same operation on different data. The number of multiprocessors in a single GPU can range nowadays up to a few hundreds and a single multiprocessor typically contains 8 or 16 scalar units sharing the instruction decoder.

The total number of scalar processors is the product of the number of multiprocessors and the number of SIMD scalar processors per multiprocessor, which can be well over a thousand. This huge number of processors can execute the same program on different data. A single execution of the program is called the ***thread.*** A multiprocessor executes a ***thread block.*** All processors have some fast local memory, which is only accessible to threads executed on the same processor, i.e. to a thread block. There is also global device memory to which data can be uploaded or downloaded from by the host program. This memory can be accessed from multiprocessors through different caching and synchronization strategies. Compared to the CPU, this means less transistors for caching, less cache performance in general, but more control for the programmer to make use of the memory architecture in an efficient way.

The above architecture favours the parallel execution of short, coherent computations on compact pieces of data. Thus, the main challenge of porting algorithms to the GPU is that of parallelization and decomposition to independent computational steps. GPU programs, which perform such a step when executed by the processing units, are often called ***kernels*** or ***shaders,*** the former alludes to the parallel data processing aspect and the latter is a legacy of the fundamental graphics task: the simulation of light reflection at object surfaces, better known as shading.

GPU programming languages and control APIs have grown pretty similar to each other in both capabilities and syntax, but they can still be divided into graphics and GPGPU solutions. The two approaches can be associated with two different programmer attitudes. While GPGPU frameworks try to add some constructs to programming languages to prepare regular code for parallel execution, graphics APIs extend previously very limited parallel shader programs into flexible computational tools. This second mindset may seem obsolete or only relevant in specific graphics-related scenarios, but in essence it is not about graphics at all: it is about the implicit knowledge of how parallel algorithms work, inherent to the incremental image synthesis pipeline. Therefore, we first discuss this pipeline and how the GPU device is seen by a graphics programmer. This will not only make the purpose and operation of device components clear, but also provides a valid and tried approach to general purpose GPU programming, and what GPU programs should ideally look like. Then we introduce the GPGPU approach, which abandons most of the graphics terminology and neglects task-specific hardware elements in favour of a higher abstraction level.

28.1. The graphics pipeline model

The graphics pipeline model provides an abstraction over the GPU hardware where we view it as a device which performs ***incremental image synthesis*** [143] (see Chapter 22 of this book, titled *Computer Graphics* of this book). Incremental image synthesis aims to render a virtual world defined by a numerical model by transforming it into linear primitives (points, lines, triangles), and rasterizing these primitives to pixels of a discrete image. The process is composed of several algorithmic steps, which are grouped in pipeline stages. Some of these stages are realized by dedicated hardware components while others are implemented through programs run by GPUs. Without going into details, let us recap the image synthesis process (Figure 28.2):

- The virtual world is a collection of model instances. The models are approximated using triangle meshes. This is called .
- In order to perform shading, the objects have to be transformed into the coordinate system where the camera and lights are specified. This is either the ***world space*** or the ***camera space.***

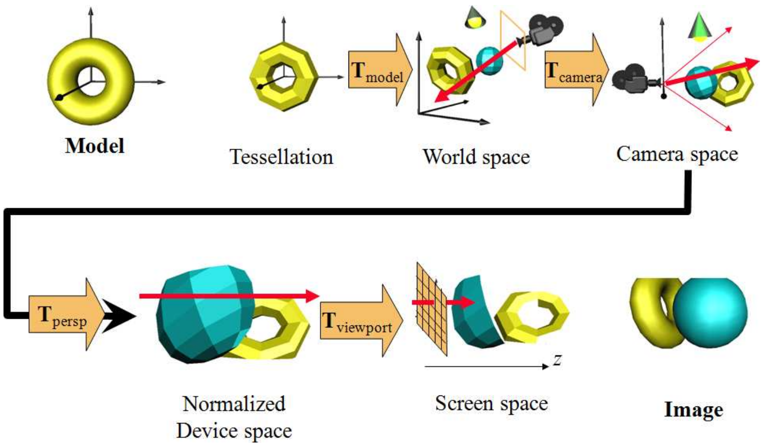


Figure 28.2 Incremental image synthesis process.

- Triangle vertices are projected on-screen according to the camera settings. Where a vertex should appear on the screen is found by applying the ***camera transformation,*** the ***perspective transformation,*** and finally the ***viewport transformation.*** In camera space the camera is in the origin and looks at the $-z$ direction. Rays originating at the camera focus, called the ***eye*** position, and passing through points on the window that represent the pixels of our display form a perspective bundle. The role of perspective transformation is to convert this perspective bundle into a parallel bundle of rays, thus to replace perspective projection by a parallel projection. After perspective transformation, the vertices are in ***normalized device space*** where the visible volume is an axis aligned cube defined by inequalities $-1 \le x \le 1$, $-1 \le y \le 1$, $-1 \le z \le 1$. Parts of the geometric primitives that are outside of this volume are removed by ***clipping.*** Normalized device space is further transformed to ***screen space,*** where the target image resolution and position are taken into account. Points of normalized device space coordinates $x = -1, y = -1$ are mapped to the lower left corner of the viewport rectangle on the screen. Points of $x = 1, y = 1$ are projected to the upper right corner. Meanwhile, the z range of $-1 \le z \le 1$ is converted to $[0, 1]$.
- In screen space every projected triangle is rasterized to a set of pixels. When an internal pixel is filled, its properties, including the z coordinate, also called the ***depth value,*** and shading data are computed via incremental linear interpolation from the vertex data. For every pixel, a shading color is computed from the interpolated data. The shading color of a pixel inside the projection of the triangle might be the color interpolated from the vertex colors. Alternatively, we can map images called ***textures*** onto the meshes. Texture images are 2D arrays of color records. An element of the texture image is called the ***texel.*** How the texture should be mapped onto triangle surfaces is specified by texture coordinates assigned to every vertex.

- Pixel colors are finally written to the ***frame buffer*** that is displayed on the computer screen. Besides the frame buffer, we maintain a ***depth buffer*** (also called ***z-buffer*** or ***depth stencil texture***), containing screen space depth, which is the z coordinate of the point whose color value is in the frame buffer. Whenever a triangle is rasterized to a pixel, the color and the depth are overwritten only if the new depth value is less than the depth stored in the depth buffer, meaning the new triangle fragment is closer to the viewer. As a result, we get a rendering of triangles correctly occluding each other in 3D. This process is commonly called the ***depth buffer algorithm.*** The depth buffer algorithm is also an example of a more general operation, which computes the pixel data as some function of the new data and the data already stored at the same location. This general operation is called ***merging.***

28.1.1. GPU as the implementation of incremental image synthesis

The GPU architecture as presented by the graphics API is the direct implementation of the image synthesis pipeline (left part of Figure 28.1). This pipeline is configured by the CPU via graphics API calls, and its operation is initiated by the ***draw call.*** A sequence of draw calls during which the configuration of the pipeline does not change (but the inputs do) is called a ***pass.*** A single draw call operates on a sequence of vertices, the attributes of which are stored in a ***vertex buffer.***

Vertices are expected to be specified in modeling space with homogeneous coordinates. A point of ***Cartesian coordinates*** (x, y, z) can be defined by the quadruple of ***homogeneous coordinates*** $[xw, yw, zw, w]$ using an arbitrary, non-zero scalar w (for more details see Chapter 21 *Computer Graphics* of this book). This representation owns its name to the fact that if the elements of the quadruple are multiplied by the same scalar, then the represented point will not change. From homogeneous quadruple $[X, Y, Z, w]$ the Cartesian coordinates of the same point can be obtained by ***homogeneous division,*** that is as $(X/w, Y/w, Z/w)$. Homogeneous coordinates have several advantages over Cartesian coordinates. When homogeneous coordinates are used, even parallel lines have an intersection (an ***ideal point,***) thus the singularity of the Euclidean geometry caused by parallel lines is eliminated. Homogeneous linear transformations include ***perspective projection*** as well, which has an important role in rendering, but cannot be expressed as a linear function of Cartesian coordinates. Most importantly, the widest class of transformations that preserve lines and planes are those which modify homogeneous coordinates linearly.

Having set the vertex buffer, vertices defined by their coordinates and attributes like texture coordinates or color begin their journey down the graphics pipeline, visiting processing stages implemented by programmable shader processors or fixed-function hardware elements. We consider these stages one-by-one.

Tessellation If the vertices do not directly define the final triangle mesh, but they are control points of a parametric surface or define just a coarse version of the mesh, the first step is the development of the final mesh, which is called ***tessellation.*** As the programmability of this stage is limited and its GPGPU potential is small, we do not discuss this stage further but assume that the vertex buffer contains the fine mesh that needs no tessellation.

Vertex processing Objects must be transformed to normalized device space for clipping, which is typically executed by a homogeneous linear transformation. Additionally, GPUs may also take the responsibility of illumination computation at the vertices of the triangle mesh. These operations are executed in the vertex shader. From a more general point of view, the vertex shader gets a single vertex at a time, modifies its attributes, including position, color, and texture coordinates, and outputs the modified vertex. Vertices are processed independently and in parallel.

The geometry shader The geometry shader stage receives vertex records along with primitive information. It may just pass them on as in the fixed-function pipeline, or spawn new vertices. Optionally, these may be written to an output buffer, which can be used as an input vertex buffer in a consecutive pass. A typical application of the geometry shader is procedural modeling, when a complex model is built from a single point or a triangle [99].

While vertex shaders have evolved from small, specialized units to general stream processors, they have kept the one record of output for every record of input scheme. The geometry shader, on the other hand, works on vertex shader output records (processed vertices), and outputs a varying (but limited) number of similar records.

Clipping The hardware keeps only those parts of the primitives that are inside an axis aligned cube of corners $(-1,-1,-1)$ and $(1,1,1)$ in normalized device space. In homogeneous coordinates, a point should meet the following requirements to be inside:

$$-w \le x \le w, \;\; -w \le y \le w, \;\; -w \le z \le w\,.$$

This formulation complies to the OpenGL [110] convention. It is valid e.g. in the Cg language when compiling for an OpenGL vertex shader profile. The last pair of inequalities can also be defined as $0 \le z \le w$, as Direct3D assumes. This is the case for Cg Direct3D profiles and in the HLSL standard. The difference is hidden by compilers which map vertex shader output to what is expected by the clipping hardware.

Clipping is executed by a fixed-function hardware of the GPU, so its operation can neither be programmed nor modified. However, if we wish our primitives to continue their path in further stages of the pipeline, the conditions of the clipping must be satisfied. In GPGPU, the clipping hardware is considered as a ***stream filter***. If it turns out that a data element processed by vertex and geometry shader programs needs to be discarded, vertices should be set to move the primitive out of the clipping volume. Then the clipping hardware will delete this element from the pipeline.

After clipping the pipeline executes , that is, it converts homogeneous coordinates to Cartesian ones by dividing the first three homogeneous coordinates by the fourth (w). The points are then transformed to where the first two Cartesian coordinates select the pixel in which this point is visible.

Rasterization with linear interpolation The heart of the pipeline is the non-programmable rasterization stage. This is capable of converting linear primitives (triangles, line segments, points) into discrete ***fragments*** corresponding to digital image pixels. More simply put, it draws triangles if the screen coordinates of the vertices are given. Pipeline stages before the rasterizer have to compute these vertex coordinates, stages after it have to process the fragments to find pixel colors.

Even though the base functionality of all stages can be motivated by rasterization, GPGPU applications do not necessarily make use of drawing triangles. Still, the rasterizer can be seen to work as a stream expander, launching an array of fragment computations for all primitive computations, only the triangles have to be set up cleverly.

Rasterization works in ***screen space*** where the x, y coordinates of the vertices are equal to those integer pixel coordinates where the vertices are projected. The vertices may have additional properties, such as a z coordinate in screen space, texture coordinates and color values. When a triangle is rasterized, all those pixels are identified which fall into the interior of the projection of the triangle. The properties of the individual pixels are obtained from the vertex properties using linear interpolation.

Fragment shading The fragment properties interpolated from vertex properties are used to find the fragment color and possibly a modified depth value. The classical operation for this includes fetching the texture memory addressed by the interpolated texture coordinates and modulating the result with the interpolated color.

Generally, fragment shader programs get the interpolated properties of the fragment and output the color and optionally modify the depth of the fragment. Like the vertex shader, the fragment shader is also one-record-in, one-record-out type processor. The fragment shader is associated with the target pixel, so it cannot write its output anywhere else.

Merging When final fragment colors are computed, they may not directly be written to the image memory, but the ***output merger*** stage is responsible for the composition. First, the depth test against the depth buffer is performed. Note that if the fragment shader does not modify the z value, depth testing might be moved before the execution of the fragment shader. This ***early z-culling*** might improve performance by not processing irrelevant fragments.

Finally, the output merger blends the new fragment color with the existing pixel color, and outputs the result. This feature could implement ***blending*** needed for ***transparent surface rendering*** (Figure 28.3).

In GPGPU, blending is mainly useful if we need to find the sum, minimum or maximum of results from consecutive computations without a need of reconfiguring the pipeline between them.

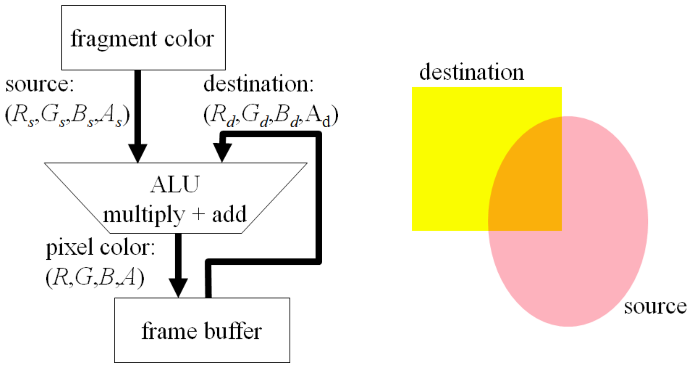


Figure 28.3 Blending unit that computes the new pixel color of the frame buffer as a function of its old color (destination) and the new fragment color (source).

28.2. GPGPU with the graphics pipeline model

In general purpose programming, we are used to concepts like input data, temporary data, output data, and functions that convert input data to temporary and finally to output data according to their parameters. If we wish to use the GPU as presented by a graphics API, our programming concepts should be mapped onto the concepts of incremental image synthesis, including geometric primitives, vertex/primitive/fragment processing, rasterization, texturing, merging, and final image. There are many different possibilities to establish this correspondence, and their comparative advantages also depend on the actual algorithm. Here we consider a few general approaches that have proven to be successful in high performance computing applications. First, we discuss how our general purpose programming concepts can be related to GPU features.

28.2.1. Output

GPUs render images, i.e. two-dimensional arrays of pixels. The ***render target*** can be the frame buffer that is displayed or an output texture (in the latter case, the pixel is often referred to as a texel). In GPGPU the output is usually a texture since texels can be stored in floating point format unlike the final frame buffer values that are unsigned bytes. Furthermore, textures can be used later on as inputs of subsequent computation passes, i.e. the two-dimensional output texture can be interpreted as one or two-dimensional input texture in the next rendering pass, or as a single layer of a three-dimensional texture. In older GPUs, a pixel was capable of storing at most five floating point values since a color is typically identified by red, green, blue, and opacity values, and hidden surface elimination needed a single distance value, which is the z coordinate of the point in screen coordinates. Later, with the emergence of ***multiple render targets,*** a pixel could be associated with several, e.g. four textures, which means that the maximum size of an output record could grow to 17 floats. In current, most advanced Shader Model 5.0 GPUs even this limitation has been lifted, so a single pixel can store a list of varying number of values.

Which pixel is targeted by the rendering process is determined by the geometric elements. Each primitive is transformed to screen space and its projection is rasterized which means that those pixels are targeted that are inside the projection. If more than one element is sent down the pipeline, their projections may overlap, so the pixel value is calculated multiple times. The merging unit combines these partial results, it may keep only one, e.g. the fragment having minimal screen space z coordinate if depth testing is enabled, or it may add up partial results using blending.

An important property of the render target is that it can be read directly by none of the shader processors, and only the fragment shader processor can indirectly write into it via the possible merging operation. Different fragment shaders are assigned to different parts of the render target, so no synchronization problem may occur.

28.2.2. Input

In image synthesis the inputs are the geometry stream and the textures used to color the geometry. As a triangle mesh geometry has usually no direct meaning in a GPGPU application, we use the geometry stream only as a control mechanism to distribute the computational load among the shader processors. The real GPGPU input will be the data stored in textures. The texture is a one-, two- or three-dimensional array of color data elements, which can store one, two, three or four scalars. In the most general case, the color has red, green, blue and opacity channels. These color values can be stored in different formats including, for example, unsigned bytes or 32 bit floats. From the point of view of GPGPU, 32 bit floats are the most appropriate.

A one-dimensional float texture is similar to the linear CPU memory where the usual data structures like arrays, lists, trees etc. can be encoded. However, the equivalence of the CPU memory and the GPU texture fails in two important aspects. In one, the texture is poorer, in the other, it is better than the linear CPU memory.

An apparent limitation is that a texture is parallel read-only for all programmable shaders with the exception of the render target that cannot be read by the shaders and is accessible only for the merger unit. Read-modify-write cycles, which are common in the CPU memory, are not available in shader programs. GPU designers had a good reason not to allow read-modify-write cycles and to classify textures as parallel read-only and exclusive write-only. In this way, the writes do not have to be cached and during reads caches get never invalidated.

On the other hand, the texture memory has much more addressing modes than a linear memory, and more importantly, they are also equipped with built-in ***texture filters.*** With the filters, a texture is not only an array of elements, but also a ***finite element*** representation of a one-, two-, or three-dimensional spatial function (refer to Section 28.7 to learn more of the relation between finite element representations and textures).

For one-dimensional textures, we can use linear filtering, which means that if the texture coordinate u points to a location in between two texels of coordinates U and $U + 1$, then the hardware automatically computes a linear interpolation of the two texel values. Let these texels be $T(U)$ and $T(U+1)$. The filtered value returned for u is then

$$T(u) = (1-u^*)T(U) + u^*T(U+1), \quad \text{where} \quad u^* = u - U\,.$$

Two-dimensional textures are filtered with ***bi-linear filtering*** taking the four texels closest to the interpolated texture coordinate pair (u, v). Let these be $T(U, V)$, $T(U+1, V)$, $T(U+1, V+1)$, and $T(U, V+1)$. The filtered value returned for (u, v) is then

$$T(U,V)u^*v^* + T(U+1,V)(1-u^*)v^* + T(U+1,V+1)(1-u^*)(1-v^*)$$
$$+T(U,V+1)u^*(1-v^*),$$

where $u^* = u - U$ and $v^* = v - V$.

For three-dimensional textures, ***tri-linear filtering*** is implemented.

28.2.3. Functions and parameters

As the primitives flow through the pipeline, shader processors and fixed-function elements process them, determining the final values in each pixel. The programs of shader processors are not changed in a single rendering pass, so we can say that each pixel is computed by the very same program. The difference of pixel colors is due to data dependencies. So, in conclusion a GPU can be regarded as a hardware that computes an array of records.

In the GPU, primitives are processed by a series of processors that are either programmable or execute fixed algorithms while output pixels are produced. It means that GPUs can also be seen as ***stream processors.*** Vertices defining primitives enter a single virtual stream and are first processed by the vertex shader. With stream processing terminology, the vertex shader is a ***mapping*** since it applies a function to the vertex data and always outputs one modified vertex for each input vertex. So, the data frequency is the same at the output as it was at the input. The geometry shader may change the topology and inputting a single primitive, it may output different primitives having different number of vertices. The data frequency may decrease, when the stream operation is called ***reduction,*** or may increase, when it is called ***expansion.*** The clipping unit may keep or remove primitives, or may even change them if they are partially inside of the clipping volume. If we ignore partially kept primitives, the clipping can be considered as a ***.*** By setting the coordinates of the vertices in the vertex shader to be outside of the clipping volume, we can filter this primitive out of the further processing steps. Rasterization converts a primitive to possibly many fragments, so it is an expansion. The fragment shader is also a mapping similarly to the vertex shader. Finally, merging may act as a selection, for example, based on the z coordinate or even as an ***accumulation*** if blending is turned on.

Shader processors get their stream data via dedicated registers, which are filled by the shader of the preceding step. These are called ***varying input.*** On the other hand, parameters can also be passed from the CPU. These parameters are called ***uniform input*** since they are identical for all elements of the stream and cannot be changed in a pass.

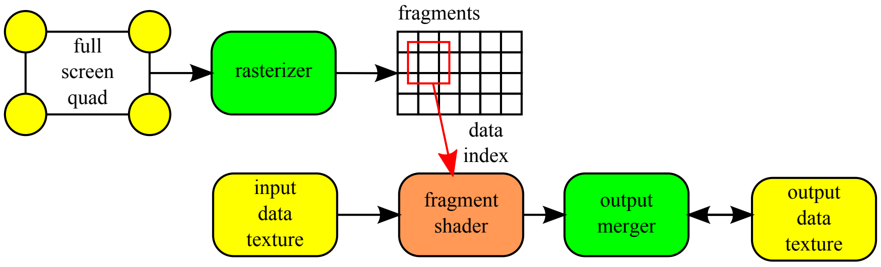


Figure 28.4 GPU as a vector processor.

28.3. GPU as a vector processor

If the computation of the elements is done independently and without sharing temporary results, the parallel machines are called ***vector processors*** or ***array processors.*** As in the GPU hardware the fragment shader is associated with the elements of the output data, we use the fragment shader to evaluate output elements. Of course, the evaluation in a given processor must also be aware which element is being computed, which is the fundamental source of data dependency (it would not make sense to compute the very same data many times on a parallel machine). In the fragment shader, the index of the data element is in fact the pair of the pixel coordinates. This is available in screen space as a pair of two integers specifying the row and the column where the pixel is located.

In the simplest, but practically the most important case, we wish to have a result in all pixels in a single rendering pass. So we have to select a geometric primitive that is mapped to all pixels in screen space and a single pixel is mapped only once. Such a geometric primitive is the virtual display itself, thus we should render a rectangle or a quadrilateral that represents the window of our virtual camera. In screen space, this is the viewport rectangle, in clipping space, this is a square on the x, y plane and having corners in homogeneous coordinates $(-1,-1,0,1)$, $(1,-1,0,1)$, $(1,1,0,1)$, $(-1,1,0,1)$. This rectangle is also called the ***full screen quad*** and is processed by the hardware as two triangles (Figure 28.4).

Suppose that we want to compute an output array $\mathbf{y}$ of dimension N from an input array $\mathbf{x}$ of possibly different dimension M and a global parameter p with function F:

$$\mathbf{y}_i = F(i, \mathbf{x}, p), \qquad i = 1, \ldots, N\,.$$

To set up the GPU for this computation, we assign output array $\mathbf{y}$ to the output texture that is the current render target. Texture size is chosen according to the output size, and the viewport is set to cover the entire render target. A two-dimensional array of H horizontal resolution and V vertical resolution is capable of storing $H \times V$ elements. If $H \times V \geq N$, then it is up to us how horizontal and vertical resolutions are found. However, GPUs may impose restrictions, e.g. they cannot be larger than 2^{12} or, if we wish to use them as input textures in the next pass or compute binary reductions, the resolutions are preferred to be powers of two. If power of two dimensions are advantageous but the dimension of the array is different, we can extend the array by additional void elements.

According to vector processing principles, different output values are computed independently without sharing temporary results. As in the GPU hardware the fragment shader is associated with the elements of the output data and can run independently of other elements, we use the fragment shader to evaluate function F. To find its parameters, we need to know i, i.e. which element is currently computed, and should have an access to input array $\mathbf{x}$. The simplest way is to store the input array as an input texture (or multiple input textures if that is more convenient) since the fragment shader can access textures.

The only responsibility of the CPU is to set the uniform parameters, specify the viewport and send a full screen quad down the pipeline. Uniform parameters select the input texture and define global parameter p. Assuming the OpenGL API, the corresponding CPU program in C would look like the following:

```
StartVectorOperation( ) {
    Set uniform parameters p and arrayX identifying the input texture

    glViewport(0, 0, H, V);     // Set horizontal and vertical resolutions, H and V
    glBegin(GL_QUADS);          // The next four vertices define a quad
        glVertex4f(-1,-1, 0, 1);   // Vertices assuming normalized device space
        glVertex4f(-1, 1, 0, 1);
        glVertex4f( 1, 1, 0, 1);
        glVertex4f( 1,-1, 0, 1);
    glEnd( );
}
```

Note that this program defines the rectangle directly in normalized device space using homogeneous coordinates passed as input parameters of the *glVertex4f* functions. So in the pipeline we should make sure that the vertices are not transformed.

For the shader program, the varying inputs are available in dedicated registers and outputs must also be written to dedicated registers. All of these registers are of type *float4*, that is, they can hold 4 float values. The role of the register is explained by its name. For example, the current value of the vertex position can be fetched from the ***POSITION*** register. Similar registers can store the texture coordinates or the color associated with this vertex.

The vertex shader gets the position of the vertex and is responsible for transforming it to the normalized device space. As we directly defined the vertices in normalized device space, the vertex shader simply copies the content of its input *POSITION* register to its output *POSITION* register (the input and output classification is given by the ***in*** and ***out*** keywords in front of the variable names assigned to registers):

```
void VertexShader( in float4  inputPos   : POSITION,
                   out float4 outputPos  : POSITION )
{
    outputPos = inputPos;
}
```

The geometry shader should keep the rectangle as it is without changing the vertex coordinates. As this is the default operation for the geometry shader, we do not specify any program for it. The rectangle leaving the geometry shader goes to the clipping stage, which keeps it since we defined our rectangle to be inside the clipping region. Then, Cartesian coordinates are obtained from the homogeneous ones by dividing the first three coordinates by the fourth one. As we set all fourth homogeneous coordinates to 1, the first three coordinates are not altered. After homogeneous division, the fixed-function stage transforms the vertices of the rectangle to the vertices of a screen space rectangle having the x, y coordinates equal to the corners of the viewport and the $z = 0$ coordinate to 0.5. Finally, this rectangle is rasterized in screen space, so all pixels of the viewport are identified as a target, and the fragment shader is invoked for each of them.

The fragment shader is our real computing unit. It gets the input array and global parameter p as uniform parameters and can also find out which pixel is being computed by reading the ***WPOS*** register:

```
float FragmentShaderF(
        in float2 index : WPOS,     // target pixel coordinates
        uniform samplerRECT arrayX, // input array
        uniform float p             // global parameter p
        ) : COLOR   // output is interpreted as a pixel color
{
    float yi = F(index, arrayX, p);  // F is the function to be evaluated
    return yi;
}
```

In this program two input parameters were declared as uniform inputs by the ***uniform*** keyword, a float parameter p and the texture identification *arrayX.* The type of the texture is ***samplerRECT*** that determines the addressing modes how a texel can be selected. In this addressing mode, texel centers are on a two-dimensional integer grid. Note that here we used a different syntax to express what the output of the shader is. Instead of declaring a register as *out,* the output is given as a return value and the function itself, and is assigned to the output *COLOR* register.

28.3.1. Implementing the SAXPY BLAS function

To show concrete examples, we first implement the level 1 functionality of the ***Basic Linear Algebra Subprograms*** (***BLAS***) library (http://www.netlib.org/blas/) that evaluates vector functions of the following general form:

$$\mathbf{y} = p\mathbf{x} + \mathbf{y},$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors and p is a scalar parameter. This operation is called ***SAXPY*** in the BLAS library. Now our fragment shader inputs two textures, vector $\mathbf{x}$ and the original version of vector $\mathbf{y}$. One fragment shader processor computes a single element of the output vector:

```
float FragmentShaderSAXPY(
        in float2 index : WPOS,      // target pixel coordinates
        uniform samplerRECT arrayX,  // input array x
        uniform samplerRECT arrayY,  // original version of y
        uniform float p              // global parameter p
        ) : COLOR                    // output is interpreted as a pixel color
{
    float yoldi = texRECT(arrayY, index);    // yoldi = arrayY[index]
    float xi    = texRECT(arrayX, index);    // xi = arrayX[index]
    float yi = p * xi + yoldi;
    return yi;
}
```

Note that instead of indexing an array of CPU style programming, here we fetch the element from a texture that represents the array by the ***texRECT*** Cg function.

The first parameter of the *texRECT* function is the identification number of a two-dimensional texture, which is passed from the CPU as a uniform parameter, and the second is the texture address pointing to the texel to be selected.

Here we can observe how we can handle the limitation that a shader can only read textures but is not allowed to write into it. In the operation, vector $\mathbf{y}$ is an input and simultaneously also the output of the operation. To resolve this, we assign two textures to vector $\mathbf{y}$. One is the original vector in texture *arrayY,* and the other one is the render target. While we read the original value, the new version is produced without reading back from the render target, which would not be possible.

28.3.2. Image filtering

Another important example is the discrete convolution of two textures, an image and a filter kernel, which is the basic operation in many image processing algorithms:

$$\tilde{L}(X,Y) \approx \sum_{i=-M}^{M} \sum_{j=-M}^{M} L(X-i, Y-j) w(i,j) , \tag{28.1}$$

where $\tilde{L}(X,Y)$ is the filtered value at pixel X, Y, $L(X,Y)$ is the original image, and $w(x,y)$ is the ***filter kernel,*** which spans over $(2M+1) \times (2M+1)$ pixels.

Now the fragment shader is responsible for the evaluation of a single output pixel according to the input image given as texture *Image* and the filter kernel stored in a smaller texture *Weight.* The half size of the filter kernel M is passed as a uniform variable:

```
float3 FragmentShaderConvolution(
        in float2 index : WPOS,     // target pixel coordinates
        uniform samplerRECT Image,  // input image
        uniform samplerRECT Weight, // filter kernel
        uniform float M             // size of the filter kernel
        ) : COLOR  // a pixel of the filtered image
{
    float3 filtered = float3(0, 0, 0);

    for(int i = -M; i <= M; i++)
        for(int j = -M; j <= M; j++) {
            float2 kernelIndex = float2(i, j);
            float2 sourceIndex = index + kernelIndex;
            filtered += texRECT(Image, sourceIndex) * texRECT(Weight, kernelIndex);
        }
    }
    return filtered;
}
```

Note that this example was a linear, i.e. convolution filter, but non-linear filters (e.g. median filtering) could be implemented similarly. In this program we applied arithmetic operators (*, +=, =) for *float2* and *float3* type variables storing two and three floats, respectively. The Cg compiler and the GPU will execute these instructions independently on the float elements.

Note also that we did not care what happens at the edges of the image, the texture is always fetched with the sum of the target address and the shift of the filter kernel. A CPU implementation ignoring image boundaries would obviously be wrong, since we would over-index the source array. However, the texture fetching hardware implementing, for example, the *texRECT* function automatically solves this problem. When the texture is initialized, we can specify what should happen if the texture coordinate is out of its domain. Depending on the selected option, we get the closest texel back, or a default value, or the address is interpreted in a periodic way.

Exercises

28.3-1 Following the vector processing concept, write a pixel shader which, when a full screen quad is rendered, quantizes the colors of an input texture to a few levels in all three channels, achieving a *cell shading* effect.

28.3-2 Following the gathering data processing scheme, write a pixel shader which, when a full screen quad is rendered, performs *median filtering* on an input grayscale image, achieving dot noise reduction. The shader should fetch nine texel values from a neighborhood of 3×3, outputting the fifth largest.

28.3-3 Implement an *anisotropic, edge preserving low-pass image filter* with the gathering data processing scheme. In order to preserve edges, compute the Euclidean distance of the original pixel color and the color of a neighboring pixel, and include the neighbor in the averaging operation only when the distance is below a threshold.

28.3-4 Write a parallel *Mandelbrot set rendering* program by assuming that pixel x, y corresponds to complex number $c = x + \mathbf{i}y$ and deciding whether or not the $z_n = z_{n-1}^2 + c$ iteration diverges when started from $z_0 = c$. The divergence may be checked by iterating $n = 10^6$ times and examining that $|z_n|$ is large enough. Divergent points are depicted with white, non-divergent points with black.

28.4. Beyond vector processing

Imagining the GPU as a vector processor is a simple but efficient application of the GPU hardware in general parallel processing. If the algorithm is suitable for vector processing, then this approach is straightforward. However, some algorithms are not good candidates for vector processing, but can still be efficiently executed by the GPU. In this section, we review the basic approaches that can extend the vector processing framework to make the GPU applicable for a wider range of algorithms.

28.4.1. SIMD or MIMD

Vector processors are usually SIMD machines, which means that they execute not only the same program for different vector elements but always the very same machine instruction at a time. It means that vector operations cannot contain data dependent conditionals or loop lengths depending on the actual data. There is only one control sequence in a SIMD parallel program.

Of course, writing programs without if conditionals and using only constants as loop cycle numbers are severe limitations, which significantly affects the program structure and the ease of development. Early GPU shaders were also SIMD type processors and placed the burden of eliminating all conditionals from the program on the shoulder of the programmer. Current GPUs and their compilers solve this problem automatically, thus, on the programming level, we can use conditionals and variable length loops as if the shaders were MIMD computers. On execution level, additional control logic makes it possible that execution paths of scalar units diverge: in this case it is still a single instruction which is executed at a time, possibly with some scalar units being idle. Operations of different control paths are serialized so that all of them are completed. The overhead of serialization makes performance strongly dependent on the coherence of execution paths, but many transistors of control logic can be spared for more processing units.

The trick of executing all branches of conditionals with possibly disabled writes is called ***predication.*** Suppose that our program has an if statement like

```
    if (condition(i)) {
        F( );
    } else {
        G( );
    }
```

Depending on the data, on some processors the $condition(i)$ may be true, while it is false on other processors, thus our vector machine would like to execute function F of the first branch in some processors while it should evaluate function G of the second branch in other processors. As in SIMD there can be only one control path, the parallel system should execute both paths and disable writes when the processor is not in a valid path. This method converts the original program to the following conditional free algorithm:

```
    enableWrite = condition(i);
    F( );
    enableWrite = !enableWrite;
    G( );
```

This version does not have conditional instructions so it can be executed by the SIMD machine. However, the computation time will be the the sum of computation times of the two functions.

This performance bottleneck can be attacked by decomposing the computation into multiple passes and by the exploitation of the feature. The early z-cull compares the z value of the fragment with the content of the depth buffer, and if it is smaller than the stored value, the fragment shader is not called for this fragment but the fragment processor is assigned to another data element. The early z-cull is enabled automatically if we execute fragment programs that do not modify the fragment's z coordinate (this is the case in all examples discussed so far).

To exploit this feature, we decompose the computation into three passes. In the first pass, only the *condition* is evaluated and the depth buffer is initialized with the values. Recall that if the z value is not modified, our full screen quad is on the xy plane in normalized device space, so it will be on the $z = 0.5$ plane in screen space. Thus, to allow a discrimination according to the condition, we can set values in the range $(0.5, 1)$ if the condition is true and in $(0, 0.5)$ if it is false.

The fragment shader of the first pass computes just the condition values and stores them in the depth buffer:

```
float FragmentShaderCondition(
        in float2 index : WPOS,     // target pixel coordinates
        uniform samplerRECT Input,  // input vector
        ) : DEPTH  // the output goes to the depth buffer
{
    bool condition = ComputeCondition( texRECT(Input, index) );
    return (condition) ? 0.8 : 0.2; // 0.8 is greater than 0.5; 0.2 is smaller than 0.5
}
```

Then we execute two passes for the evaluation of functions F and G, respectively. In the first pass, the fragment shader computes F and the depth comparison is set to pass those fragments where their $z = 0.5$ coordinate is less than the depth value stored in the depth buffer. In this pass, only those fragments are evaluated where the depth buffer has 0.8 value, i.e. where the previous condition was true. Then, in the second pass, the fragment shader is set to compute G while the depth buffer is turned to keep those fragments where the fragment's depth is greater than the stored value.

In Subsection 28.7.1 we exploit early z-culling to implement a variable length loop in fragment processors.

28.4.2. Reduction

The vector processing principle assumes that the output is an array where elements are obtained independently. The array should be large enough to keep every shader processor busy. Clearly, if the array has just one or a few elements, then only one or a few shader processors may work at a time, so we loose the advantages of parallel processing.

In many algorithms, the final result is not a large array, but is a single value computed from the array. Thus, the algorithm should reduce the dimension of the output. Doing the in a single step by producing a single texel would not benefit from the parallel architecture. Thus, reduction should also be executed in parallel, in multiple steps. This is possible if the operation needed to compute the result from the array is associative, which is the case for the most common operations, like sum, average, maximum, or minimum.

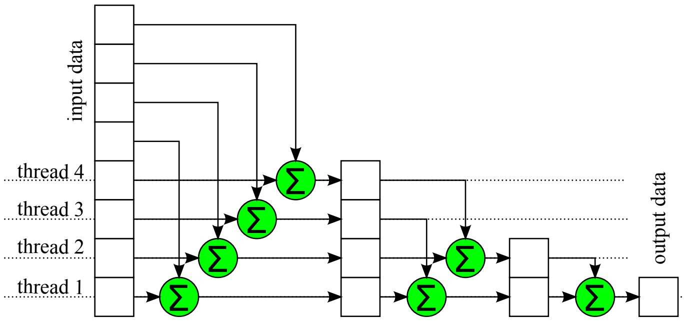


Figure 28.5 An example for parallel reduction that sums the elements of the input vector.

Suppose that the array is encoded by a two-dimensional texture. At a single phase, we downsample the texture by halving its linear resolution, i.e. replacing four neighboring texels by a single texel. The fragment shaders will compute the operation on four texels. If the original array has $2^n \times 2^n$ resolution, then n reduction steps are needed to obtain a single 1×1 output value. In the following example, we compute the sum of the elements of the input array (Figure 28.5). The CPU program renders a full screen quad in each iteration having divided the render target resolution by two:

```
Reduction( ) {
    Set uniform parameter arrayX to identify the input texture

    for(N /= 2 ; N >= 1; N /= 2) {  // log_2 N iterations
        glViewport(0, 0, N, N);     // Set render target dimensions to hold NxN elements
        glBegin(GL_QUADS);          // Render a full screen quad
             glVertex4f(-1,-1, 0, 1);
             glVertex4f(-1, 1, 0, 1);
             glVertex4f( 1, 1, 0, 1);
             glVertex4f( 1,-1, 0, 1);
        glEnd( );

        Copy render target to input texture arrayX
    }
}
```

The fragment shader computes a single reduced texel from four texels as a summation in each iteration step:

```
float FragmentShaderSum( ) (
        in float2 index : WPOS,      // target pixel coordinates
        uniform samplerRECT arrayX,  // input array x
        ) : COLOR                    // output is interpreted as a pixel color
{
    float sum = texRECT(arrayX, 2 * index);
    sum += texRECT(arrayX, 2 * index + float2(1, 0));
    sum += texRECT(arrayX, 2 * index + float2(1, 1));
    sum += texRECT(arrayX, 2 * index + float2(0, 1));
    return sum;
}
```

Note that if we exploited the bi-linear filtering feature of the texture memory, then we could save three texture fetch operations and obtain the average in a single step.

28.4.3. Implementing scatter

In vector processing a processor is assigned to each output value, i.e. every processor should be aware which output element it is computing and it is not allowed to deroute its result to somewhere else. Such a static assignment is appropriate for ***gathering*** type computations. The general structure of gathering is that we may rely on a dynamically selected set of input elements but the variable where the output is stored is known a-priory:

```
    index = ComputeIndex( );     // index of the input data
    y = F(x[index]);
```

Opposed to gathering, algorithms may have ***scattering*** characteristics, i.e. a given input value may end up in a variable that is selected dynamically. A simple scatter operation is:

```
    index = ComputeIndex( );     // index of the output data
    y[index] = F(x);
```

Vector processing frameworks and our fragment shader implementation are unable to implement scatter since the fragment shader can only write to the pixel it

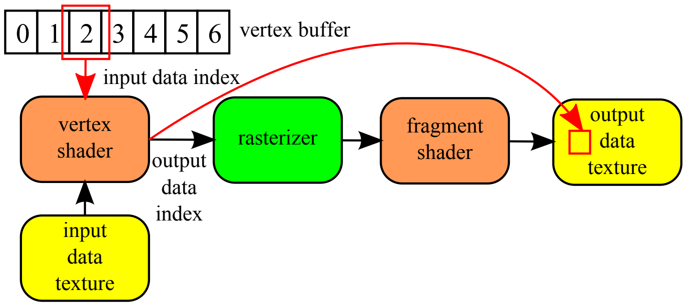


Figure 28.6 Implementation of scatter.

has been assigned to.

If we wish to solve a problem having scattering type algorithm on the GPU, we have two options. First, we can restructure the algorithm to be of gathering type. Converting scattering type parallel algorithms to gathering type ones requires a change of our viewpoint how we look at the problem and its solution. For example, when integral equations or transport problems are considered, this corresponds to the solution of the adjoint problem [144]. Secondly, we can move the index calculation up to the pipeline and use the rasterizer to establish the dynamic correspondence between the index and the render target (Figure 28.6).

Let us consider a famous scattering type algorithm, ***histogram generation.*** Suppose we scan an input array $\mathbf{x}$ of dimension M, evaluate function F for the elements, and calculate output array $\mathbf{y}$ of dimension N that stores the number of function values that are in bins equally subdividing range $(Fmin, Fmax)$.

A scalar implementation of histogram generation would be:

```
Histogram( x ) {
    for(int i = 0; i < M; i++) {
        index = (int)((F(x[i]) - Fmin)/(Fmax - Fmin) * N); // bin
        index = max(index, 0);
        index = min(index, N-1);
        y[index] = y[index] + 1;
    }
}
```

We can see that the above function writes to the output array at random locations, meaning it cannot be implemented in a fragment shader which is only allowed to write the render target at its dedicated index. The problem of scattering will be solved by computing the index in the vertex shader but delegating the responsibility of incrementing to the rest of the pipeline. The indices are mapped to output pixels by the rasterization hardware. The problem of read-modify-write cycles might be solved by starting a new pass after each increment operation and copying the current render target as an input texture of the next rendering pass. However, this solution would have very poor performance and would not utilize the parallel hardware at all. A much better solution uses the arithmetic capabilities of the merging unit. The fragment shader generates just the increment (i.e. value 1) where the histogram needs to be updated and gives this value to the merging unit. The merging unit, in turn, adds the increment to the content of the render target.

The CPU program generates a point primitive for each input data element. Additionally, it sets the render target to match the output array and also enables the merging unit to execute add operations:

```
ScanInputVector( ) {
    Set uniform parameters Fmin, Fmax, N

    glDisable(GL_DEPTH_TEST);     // Turn depth buffering off
    glBlendFunc(GL_ONE, GL_ONE); // Blending operation: dest = source * 1 + dest * 1;
    glEnable(GL_BLEND);           // Enable blending

    glViewport(0, 0, N, 1);       // Set render target dimensions to hold N elements
    glBegin(GL_POINTS);           // Assign a point primitive to each input elements
    for(int i = 0; i < M; i++) {
        glVertex1f( x[i] );       // an input element as a point primitive
    }
    glEnd( );
}
```

The vertex positions in this level are not important since it turns out later where this point will be mapped. So we use the first coordinate of the vertex to pass the current input element $x[i]$.

The vertex shader gets the position of the vertex currently storing the input element, and finds the location of this point in normalized device space. First, function F is evaluated and the bin index is obtained, then we convert this index to the $[-1,1]$ range since in normalized device space these will correspond to the extremes of the viewport:

```
void VertexShaderHistogram(
    in float   inputPos   : POSITION,
    out float4 outputPos  : POSITION,
    uniform float Fmin,
    uniform float Fmax,
    uniform float N )
{
    float xi = inputPos;
    int index = (int)((F(xi) - Fmin)/(Fmax - Fmin) * N); // bin
    index = max(index, 0);
    index = min(index, N-1);
    float nindex = 2.0 * index / N - 1.0;    // normalized device space
    outputPos = float4(nindex, 0, 0, 1);     // set output coordinates
}
```

The above example is not optimized. Note that the index calculation and the normalization could be merged together and we do not even need the size of the output array N to execute this operation.

The fragment shader will be invoked for the pixel on which the point primitive is mapped. It simply outputs an increment value of 1:

```
float FragmentShaderIncr( ) : COLOR   // output is interpreted as a pixel color
{
    return 1;  // increment that is added to the render target by merging
}
```

28.4.4. Parallelism versus reuse

Parallel processors running independently offer a linear speed up over equivalent scalar processor implementations. However, scalar processors may benefit from recognizing similar parts in the computation of different output values, so they can

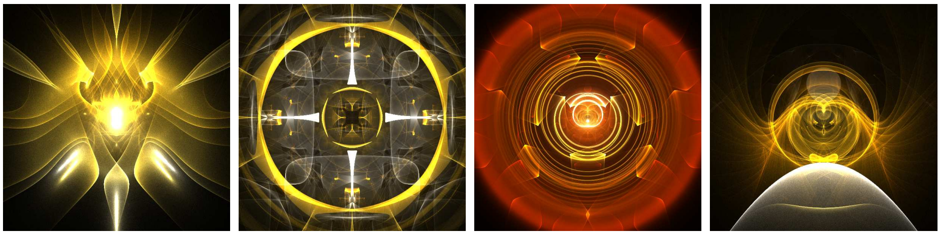


Figure 28.7 Caustics rendering is a practical use of histogram generation. The illumination intensity of the target will be proportional to the number of photons it receives (images courtesy of Dávid Balambér).

increase their performance utilizing ***reuse.*** As parallel processors may not reuse data generated by other processors, their comparative advantages become less attractive.

GPUs are parallel systems of significant streaming capabilities, so if data that can be reused are generated early, we can get the advantages of both independent parallel processing and the reuse features of scalar computing.

Our main stream expander is the rasterization. Thus anything happens before rasterization can be considered as a global computation for all those pixels that are filled with the rasterized version of the primitive. Alternatively, the result of a pass can be considered as an input texture in the next pass, so results obtained in the previous pass can be reused by all threads in the next pass.

Exercises

28.4-1 Implement a *parallel regula falsi equation solver* for $(2-a-b)x^3 + ax^2 + bx - 1 = 0$ that searches for roots in $[0,1]$ for many different a and b parameters. The a and b parameters are stored in a texture and the pixel shader is responsible for iteratively solving the equation for a particular parameter pair. Terminate the iteration when the error is below a given threshold. Take advantage of the early z-culling hardware to prevent further refinement of the terminated iterations. Analyze the performance gain.

28.4-2 Based on the reduction scheme, write a program which applies simple linear *tone mapping* to a high dynamic range image stored in a floating-point texture. The scaling factor should be chosen to map the maximum texel value to the value of one. Find this maximum using iterative reduction of the texture.

28.4-3 Based on the concept of scatter, implement a *caustics renderer* program (Figure 28.7). The scene includes a point light source, a glass sphere, and a diffuse square that is visualized on the screen. Photons with random directions are generated by the CPU and passed to the GPU as point primitives. The vertex shader traces the photon through possible reflections or refractions and decides where the photon will eventually hit the diffuse square. The point primitive is directed to that pixel and the photon powers are added by additive alpha blending.

28.4-4 Based on the concept of scatter, given an array of GSM transmitter tower coordinates, compute cell phone signal strength on a 2D grid. Assume signal strength diminishes linearly with the distance to the nearest transmitter. Use the rasterizer to render circular features onto a 2D render target, and set up blending to pick the maximum.

28.5. GPGPU programming model: CUDA and OpenCL

The ***Compute Unified Device Architecture*** (***CUDA***) and the interfaces provide the programmer with a programming model that is significantly different from the graphics pipeline model (right of Figure 28.1). It presents the GPU as a collection of multiprocessors where each multiprocessor contains several SIMD scalar processors. Scalar processors have their own registers and can communicate inside a multiprocessor via a fast ***shared memory.*** Scalar processors can read cached textures having built-in filtering and can read or write the slow global memory. If we wish, even read-modify-write operations can also be used. Parts of the global memory can be declared as a texture, but from that point it becomes read-only.

Unlike in the graphics API model, the write to the global memory is not exclusive and ***atomic add*** operations are available to support semaphores and data consistency. The fixed-function elements like clipping, rasterization, and merging are not visible in this programming model.

Comparing the GPGPU programming model to the graphics API model, we notice that it is cleaner and simpler. In the GPGPU programming model, parallel processors are on the same level and can access the global memory in an unrestricted way, while in the graphics API model, processors and fixed-function hardware form streams and write is possible only at the end of the stream. When we program through the GPGPU model, we face less restrictions than in the graphics pipeline model. However, care should be practiced since the graphics pipeline model forbids exactly those features that are not recommended to use in high performance applications.

The art of programming the GPGPU model is an efficient decomposition of the original algorithm to parallel threads that can run with minimum amount of data communication and synchronization, but always keep most of the processors busy. In the following sections we analyze a fundamental operation, the matrix-vector multiplication, and discuss how these requirements can be met.

28.6. Matrix-vector multiplication

Computational problems are based on mathematical models and their numerical solution. The numerical solution methods practically always rely on some kind of linearization, resulting in algorithms that require us to solve linear systems of equations and perform matrix-vector multiplication as a core of the iterative solution. Thus, matrix-vector multiplication is a basic operation that can be, if implemented efficiently on the parallel architecture, the most general building block in any numerical algorithm. We define the basic problem to be the computation of the result vector $\mathbf{y}$ from input matrix $\mathbf{A}$, vectors $\mathbf{x}$ and $\mathbf{b}$, as

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}\,.$$

We call this the MV problem. Let $N \times M$ be the dimensions of matrix $\mathbf{A}$. As every input vector element may contribute to each of the output vector elements, a scalar CPU implementation would contain a double loop, one loop scans the input elements while the other the output elements. If we parallelize the algorithm by assigning output elements to parallel threads, then we obtain a gathering type algorithm where a thread gathers the contributions of all input elements and aggregates them to the thread's single output value. On the other hand, if we assigned parallel threads to input elements, then a thread would compute the contribution of this input element to all output elements, which would be a scatter operation. In case of gathering, threads share only input data but their output is exclusive so no synchronization is needed. In case of scattering, multiple threads may add their contribution to the same output element, so atomic adds are needed, which may result in performance degradation.

An implementation of the matrix-vector multiplication on a scalar processor looks like the following:

```
void ScalarMV(int N, int M, float* y, const float* A, const float* x, const float* b)
{
    for(int i=0; i<N; i++) {
        float yi = b[i];
        for(int j=0; j<M; j++) yi += A[i * M + j] * x[j];
        y[i] = yi;
    }
}
```

The first step of porting this algorithm to a parallel machine is to determine what a single thread would do from this program. From the options of gathering and scattering, we should prefer gathering since that automatically eliminates the problems of non-exclusive write operations. In a gathering type solution, a thread computes a single element of vector $\mathbf{y}$ and thus we need to start N threads. A GPU can launch a practically unlimited number of threads that are grouped in thread blocks. Threads of a block are assigned to the same multiprocessor. So the next design decision is how the N threads are distributed in blocks. A multiprocessor typically executes 32 threads in parallel, so the number of threads in a block should be some multiple of 32. When the threads are halted because of a slow memory access, a hardware scheduler tries to continue the processing of other threads, so it is wise to assign more than 32 threads to a multiprocessor to always have threads that are ready to run. However, increasing the number of threads in a single block may also mean that at the end we have just a few blocks, i.e. our program will run just on a few multiprocessors. Considering these, we assign 256 threads to a single block and hope that $N/256$ exceeds the number of multiprocessors and thus we fully utilize the parallel hardware.

There is a slight problem if N is not a multiple of 256. We should assign the last elements of the vector to some processors as well, so the thread block number should be the ceiling of $N/256$. As a result of this, we shall have threads that are not associated with vector elements. It is not a problem if the extra threads can detect it and cause no harm, e.g. they do not over-index the output array.

Similarly to the discussed vector processing model, a thread should be aware which output element it is computing. The CUDA library provides implicit input parameters that encode this information: *blockIdx* is the index of the thread block, *blockDim* is the number of threads in a block, and *threadIdx* is the index of the thread inside the block.

The program of the CUDA kernel computing a single element of the output vector is now a part of a conventional CPU program:

```
__global__ void cudaSimpleMV(int N, int M, float* y, float* A, float* x, float* b)
{
    // Determine element to process from thread and block indices
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N) {  // if the index is out of the range of the output array, skip.
        float yi = b[i];
        for(int j=0; j<M; j++) yi += A[i * M + j] * x[j];
        y[i] = yi;
    }
}
```

The ***global*** keyword tells the compiler that this function will run not on the CPU but on the GPU and it may be invoked from the CPU as well. The parameters are passed according to the normal C syntax. The only special feature is the use of the implicit parameters to compute the identification number of this thread, which is the index of the output array.

The kernels are started from a CPU program that sets the parameters and also defines the number of thread blocks and the number of threads inside a block.

```
__host__ void run_cudaSimpleMV()
{
    int threadsPerBlock = 256; // number of threads per block
    int blockNum = (N +  threadsPerBlock - 1)/threadsPerBlock; // number of blocks
    cudaSimpleMV<<<blockNum, threadsPerBlock>>>(N, M, y, A, x, b);
}
```

The compiler will realize that this function runs on the CPU by reading the ***host*** keyword. The parallel threads are started like a normal C function call with the exception of the *<blockNum, threadsPerBlock>* tag, which defines how many threads should be started and how they are distributed among the multiprocessors.

28.6.1. Making matrix-vector multiplication more parallel

So far, we assigned matrix rows to parallel threads and computed scalar product $\mathbf{A}_i\mathbf{x}$ serially inside threads. If the number of matrix rows is less than the number of parallel scalar processors, this amount of parallelization is not enough to supply all processing units with work to do, and the execution of individual threads will be lengthy. Reformulating the scalar product computation is a well known, but tougher parallelization problem, as the additions cannot be executed independently, and we require a single scalar to be written for every row of the matrix. However, parts of the summation can be executed independently, and then the results added. This is a classic example of . It is required that the threads whose results are to be added both finish execution and write their results to where they are accessible for the thread that needs to add them. Thus, we use ***thread synchronization*** and available only for the threads of the same block.

Let us assume first—unrealistically—that we can have M threads processing a row and the shared memory can hold M floating point values. Let $\mathbf{Q}$ be the vector of length M residing in shared memory. Then, every thread can compute one element Q_j as $A_{ij}x_j$.. Finally, elements of $\mathbf{Q}$ must be reduced by summation. Let us further assume that $M = 2^k$.. The reduction can be carried out in k steps, terminating half of the threads, while each surviving thread adds the value in $\mathbf{Q}$ computed by a terminated one to its own. The final remaining thread outputs the value to the global memory.

```
#define M THE_NUMBER_OF_MATRIX_COLUMNS
__global__ void cudaReduceMV(int N, float* y, float* A, float* x, float* b)
{
    int i = blockIdx.x;
    int j = threadIdx.x;

    __shared__ float Q[M];  // in the shader memory inside a multiprocessor

    Q[j] = A[i * M + j] * x[j];  // a parallel part of matrix-vector multiplication

    for(int stride = M / 2; stride > 0; stride >>= 1) // reduction
    {
        __syncthreads();    // wait until all other threads of the block arrive this point
        if(j + stride < M)
            Q[j] += Q[j + stride];
    }

    if(j == 0)              // reduced to a single element
        y[i] = Q[0] + b[i];
}

__host__ void run_cudaReduceMV()
{
    cudaReduceMV<<< N, M >>>(N, y, A, x, b);
}
```

For practical matrix dimensions ($M > 10^4$), neither the number of possible threads of a single multiprocessor nor the size of the shared memory is enough to process all elements in parallel. In our next example, we use a single block of threads with limited size to process a large matrix. First, we break the output vector into segments of size T. Elements within such a segment are evaluated in parallel, then the threads proceed to the next segment. Second, for every scalar product computation, we break the vectors $\mathbf{A}_i$ and $\mathbf{x}$ into segments of length Z. We maintain a shared vector $\mathbf{Q}_t$ of length Z for every row being processed in parallel. We can compute the elementwise product of the $\mathbf{A}_i$ and $\mathbf{x}$ segments in parallel, and add it to $\mathbf{Q}_t$. As T rows are being processed by Z threads each, the block will consist of $T \times Z$ threads. From one thread's perspective this means it has to loop over $\mathbf{y}$ with a stride of T, and for every such element in $\mathbf{y}$, loop over $\mathbf{A}_i$ and $\mathbf{x}$ with a stride of Z. Also for every element in $\mathbf{y}$, the contents of $\mathbf{Q}_t$ must be summed by reduction as before. The complete kernel which works with large matrices would then be:

```
__global__ void cudaLargeMV(int N, int M, float* y, float* A, float* x, float* b)
{
    __shared__ float Q[T * Z]; // stored in the shared memory inside a multiprocessor

    int t = threadIdx.x / Z;
    int z = threadIdx.x % Z;

    for(int i = t; i < N; i += T)
    {
        Q[t * Z + z] = 0;
        for(int j = z; j < M; j += Z)
            Q[t * Z + z] += A[i * M + j] * x[j];

        for(int stride = Z / 2; stride > 0; stride >>= 1)
        {
            __syncthreads();
            if(z + stride < Z)
                Q[t * Z + z] += Q[t * Z + z + stride];
        }

        if(z == 0)
            y[i] = Q[t * Z + 0] + b[i];
    }
}

__host__ void run_cudaLargeMV()
{
    cudaReduceMV<<< 1, T*Z >>>(N, M, y, A, x, b);
}
```

This can easily be extended to make use of multiple thread blocks by restricting the outer loop to only a fraction of the matrix rows based on the *blockIdx* parameter.

The above algorithm uses shared memory straightforwardly and allows us to align memory access of threads through a proper choice of block sizes. However, every element of vector **x** must be read once for the computation of every row. We can improve on this if we read values of **x** into the shared memory and have threads in one block operate on multiple rows of the matrix. This, however, means we can use less shared memory per line to parallelize summation. The analysis of this trade-off is beyond the scope of this chapter, but a block size of 64×8 has been proposed in [53]. With such a strategy it is also beneficial to access matrix **A** as a texture, as data access will exhibit 2D locality, supported by texture caching hardware.

Even though matrix-vector multiplication is a general mathematical formulation for a wide range of computational problems, the arising matrices are often large, but sparse. In case of sparse matrices, the previously introduced matrix-vector multiplication algorithms will not be efficient as they explicitly compute multiplication with zero elements. Sparse matrix representations and MV algorithms are discussed in [15].

Exercises

28.6-1 Implement matrix-vector multiplication for large matrices in CUDA. Compare results to a CPU implementation.

28.6-2 Implement an inverse iteration type Julia set renderer. The Julia set is the attractor of the $z_n = z_{n-1}^2 + c$ iteration where z_n and c are complex numbers. Inverse iteration starts from a fixed point of the iteration formula, and iterates the inverse mapping, $z_n = \pm\sqrt{z_n - c}$ by randomly selecting either $\sqrt{z_n - c}$ or $-\sqrt{z_n - c}$ from the two possibilities. Threads must use pseudo-random generators that are initialized with different seeds. Note that CUDA has no built-in random number generator, so implement one in the program.

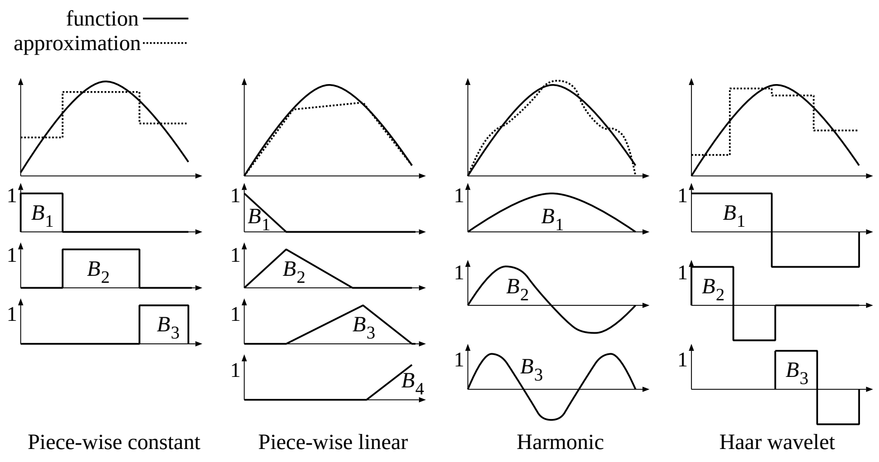


Figure 28.8 Finite element representations of functions. The texture filtering of the GPU directly supports finite element representations using regularly placed samples in one-, two-, and three-dimensions and interpolating with piece-wise constant and piece-wise linear basis functions.

28.7. Case study: computational fluid dynamics

Problems emerging in physics or engineering are usually described mathematically as a set of partial differential or integral equations. As physical systems expand in space and time, derivatives or integrals should be evaluated both in temporal and spatial domains.

When we have to represent a value over space and time, we should use functions having the spatial position and the time as their variables. The representation of general functions would require infinite amount of data, so in numerical methods we only approximate them with finite number of values. Intuitively, these values can be imagined as the function values at discrete points and time instances. The theory behind this is the ***finite element method.*** If we need to represent function $f(\vec{r})$ with finite data, we approximate the function in the following finite series form (Figure 28.8):

$$f(\vec{r}) \approx \tilde{f}(\vec{r}) = \sum_{i=1}^{N} f_i B_i(\vec{r}),$$

where $B_1(\vec{r}), \ldots, B_N(\vec{r})$ are pre-defined ***basis functions*** and $f_1, \ldots, f_N$ are the coefficients that describe $\tilde{f}$.

A particularly simple finite element representation is the piece-wise linear scheme that finds possibly regularly placed sample points $\vec{r}_1, \ldots, \vec{r}_N$ in the domain, evaluates the function at these points to obtain the coefficients $f_i = f(\vec{r}_i)$ and linearly interpolates between $\vec{r}_i$ and $\vec{r}_{i+1}$.

When the system is dynamic, solution f will be time dependent, so a new finite element representation is needed for every time instance. We have basically two options for this. We can set sample points $\vec{r}_1, \ldots, \vec{r}_N$ in a static way and allow only coefficients f_i to change in time. This approach is called ***Eulerian.*** On the other hand, we can also allow the sample points to move with the evaluation of the system, making also sample points $\vec{r}_i$ time dependent. This is the ***Lagrangian*** approach, where sample locations are also called ***particles.***

Intuitive examples of Eulerian and Lagrangian discretization schemes are how temperature and other attributes are measured in meteorology. In ground stations, these data are measured at fixed locations. However, meteorological balloons can also provide the same data, but from varying positions that follow the flow of the air.

In this section we discuss a case study for GPU-based scientific computation. The selected problem is ***computational fluid dynamics.*** Many phenomena that can be seen in nature like smoke, cloud formation, fire, and explosion show fluid-like behavior. Understandably, there is a need for good and fast fluid solvers both in engineering and in computer animation.

The mathematical model of the fluid motion is given by the Navier-Stokes equation. First we introduce this partial differential equation, then discuss how GPU-based Eulerian and Langrangian solvers can be developed for it.

A fluid with constant density and temperature can be described by its velocity $\vec{v} = (v_x, v_y, v_z)$ and pressure p fields. The velocity and the pressure vary both in space and time:

$$\vec{v} = \vec{v}(\vec{r}, t), \qquad p = p(\vec{r}, t)\,.$$

Let us focus on a fluid element of unit volume that is at point $\vec{r}$ at time t. At an earlier time instance $t - \mathrm{d}t$, this fluid element was in $\vec{r} - \vec{v}\mathrm{d}t$ and, according to the fundamental law of dynamics, its velocity changed according to an acceleration that is equal to total force $\vec{F}$ divided by mass ρ of this unit volume fluid element:

$$\vec{v}(\vec{r}, t) = \vec{v}(\vec{r} - \vec{v}\mathrm{d}t, t - \mathrm{d}t) + \frac{\vec{F}}{\rho}\mathrm{d}t\,.$$

Mass ρ of a unit volume fluid element is called the ***fluid density.*** Moving the velocity terms to the left side and dividing the equation by $\mathrm{d}t$, we can express the ***substantial derivative*** of the velocity:

$$\frac{\vec{v}(\vec{r}, t) - \vec{v}(\vec{r} - \vec{v}\mathrm{d}t, t - \mathrm{d}t)}{\mathrm{d}t} = \frac{\vec{F}}{\rho}\,.$$

The total force can stem from different sources. It may be due to the pressure differences:

$$\vec{F}_{\text{pressure}} = -\vec{\nabla} p = -\left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}\right)\,,$$

where $\vec{\nabla} p$ is the ***gradient*** of the pressure field. The minus sign indicates that the pressure accelerates the fluid element towards the low pressure regions. Here we used the ***nabla operator,*** which has the following form in a Cartesian coordinate system:

$$\vec{\nabla} = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)\,.$$

Due to friction, the fluid motion is damped. This damping depends on the ***viscosity*** ν of the fluid. Highly viscous fluids like syrup stick together, while low-viscosity fluids flow freely. The total damping force is expressed as a diffusion term since the viscosity force is proportional to the ***Laplacian*** of the velocity field:

$$\vec{F}_{\text{viscosity}} = \nu \vec{\nabla}^2 \vec{v} = \nu \left(\frac{\partial^2 \vec{v}}{\partial x^2} + \frac{\partial^2 \vec{v}}{\partial y^2} + \frac{\partial^2 \vec{v}}{\partial z^2} \right) .$$

Finally, an external force field $\vec{F}_{\text{external}}$ may also act on our fluid element causing acceleration. In the gravity field of the Earth, assuming that the vertical direction is axis z, this external acceleration is $(0, 0, -g)$ where $g = 9.8$ $[m/s^2]$.

Adding the forces together, we can obtain the ***Navier-Stokes equation*** for the velocity of our fluid element:

$$\rho \frac{\vec{v}(\vec{r}, t) - \vec{v}(\vec{r} - \vec{v}\mathrm{d}t, t - \mathrm{d}t)}{\mathrm{d}t} = -\vec{\nabla} p + \nu \vec{\nabla}^2 \vec{v} + \vec{F}_{\text{external}} .$$

In fact, this equation is the adaptation of the fundamental law of dynamics for fluids. If there is no external force field, the momentum of the dynamic system must be preserved. This is why this equation is also called ***momentum conservation equation.***

Closed physics systems preserve not only the momentum but also the mass, so this aspect should also be built into our fluid model. Simply put, the mass conservation means that what flows into a volume must also flow out, so the divergence of the mass flow is zero. If the fluid is incompressible, then the fluid density is constant, thus the mass flow is proportional to the velocity field. For incompressible fluids, the mass conservation means that the velocity field is ***divergence*** free:

$$\vec{\nabla} \cdot \vec{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} = 0 . \tag{28.2}$$

28.7.1. Eulerian solver for fluid dynamics

The Eulerian approach tracks the evolution of the velocity and pressure fields on fixed, uniform grid points. The grid allows a simple approximation of spatial derivatives by finite differences. If the grid points are in distances Δx, Δy, and Δz along the three coordinate axes and the values of scalar field p and vector field $\vec{v}$ at grid point (i, j, k) are $p^{i,j,k}$ and $\vec{v}^{i,j,k}$, respectively, then the gradient, the divergence and the Laplacian operators can be approximated as:

$$\vec{\nabla} p \approx \left(\frac{p^{i+1,j,k} - p^{i-1,j,k}}{2\Delta x}, \frac{p^{i,j+1,k} - p^{i,j-1,k}}{2\Delta y}, \frac{p^{i,j,k+1} - p^{i,j,k-1}}{2\Delta x} \right) , \tag{28.3}$$

$$\vec{\nabla} \cdot \vec{v} \approx \frac{v_x^{i+1,j,k} - v_x^{i-1,j,k}}{2\Delta x} + \frac{v_y^{i,j+1,k} - v_y^{i,j-1,k}}{2\Delta y} + \frac{v_z^{i,j,k+1} - v_z^{i,j,k-1}}{2\Delta z} , \tag{28.4}$$

$$\vec{\nabla}^2 p \approx \frac{p^{i+1,j,k} - 2p^{i,j,k} + p^{i-1,j,k}}{(\Delta x)^2} + \frac{p^{i,j+1,k} - 2p^{i,j,k} + p^{i,j-1,k}}{(\Delta x)^2}$$

$$+\frac{p^{i,j,k+1} - 2p^{i,j,k} + p^{i,j,k-1}}{(\Delta x)^2}\,. \tag{28.5}$$

The Navier-Stokes equation and the requirement that the velocity is divergence free define four scalar equations (the conservation of momentum is a vector equation) with four scalar unknowns (v_x, v_y, v_z, p). The numerical solver computes the current fields advancing the time in discrete steps of length Δt:

$$\vec{v}(\vec{r},t) = \vec{v}(\vec{r} - \vec{v}\Delta t, t - \Delta t) + \frac{\nu \Delta t}{\rho}\vec{\nabla}^2\vec{v} + \frac{\Delta t}{\rho}\vec{F}_{\text{external}} - \frac{\Delta t}{\rho}\vec{\nabla}p\,.$$

The velocity field is updated in several steps, each considering a single term on the right side of this equation. Let us consider these steps one-by-one.

Advection To initialize the new velocity field at point $\vec{r}$, we fetch the previous field at position $\vec{r} - \vec{v}\Delta t$ since the fluid element arriving at point $\vec{r}$ was there in the previous time step [141]. This step computes advection, i.e. the phenomenon that the fluid carries its own velocity field:

$$\vec{w}_1(\vec{r}) = \vec{v}(\vec{r} - \vec{v}\Delta t, t - \Delta t)\,.$$

Diffusion To damp the velocity field, we could update it proportionally to a diffusion term:

$$\vec{w}_2 = \vec{w}_1 + \frac{\nu \Delta t}{\rho}\vec{\nabla}^2\vec{w}_1\,.$$

However, this type of ***forward Euler integrator*** is numerically unstable. The reason of instability is that forward methods predict the future based on the present values, and as time passes, each simulation step adds some error, which may accumulate and exceed any limit.

Unlike forward integrators, a backward method can guarantee stability. A backward looking approach is stable since while predicting the future, it simultaneously corrects the past. Thus, the total error converges to a finite value and remains bounded. Here a backward method means that the Laplacian is obtained from the future, yet unknown velocity field, and not from the current velocity field:

$$\vec{w}_2 = \vec{w}_1 + \frac{\nu \Delta t}{\rho}\vec{\nabla}^2\vec{w}_2\,. \tag{28.6}$$

At this step of the computation, the advected field $\vec{w}_1$ is available at the grid points, the unknowns are the diffused velocity $\vec{w}_2^{i,j,k}$ for each of the grid points. Using (28.5) to compute the Laplacian of the x, y, z coordinates of unknown vector field $\vec{w}_2$ at grid point (i, j, k), we observe that it will be a linear function of the $\vec{w}_2$ velocities in the (i, j, k) grid point and its neighbors. Thus, (28.6) is a sparse linear system of equations:

$$\mathbf{w}_2 = \mathbf{w}_1 + \mathbf{A}\cdot\mathbf{w}_2 \tag{28.7}$$

where vector $\mathbf{w}_1$ is the vector of the known velocities obtained by advection, $\mathbf{w}_2$ is the vector of unknown velocities of the grid points, and matrix-vector multiplication $\mathbf{A}\cdot\mathbf{w}_2$ represents the discrete form of $(\nu\Delta t/\rho)\nabla^2\vec{w}_2(\vec{r})$.

Such systems are primary candidates for ***Jacobi iteration*** (see Chapter 12 of this book, titled *Scientific Computation*). Initially we fill vector $\mathbf{w}_2$ with zero and evaluate the right side of (28.7) iteratively, moving the result of the previous step to vector $\mathbf{w}_2$ of the right side. Thus, we traced back the problem to a sequence of sparse vector-matrix multiplications. Note that matrix $\mathbf{A}$ needs not be stored. When velocity field $\vec{w}_2$ is needed at a grid point, the neighbors are looked up and the simple formula of (28.5) gives us the result.

Updating a value in a grid point according to its previous value and the values of its neighbors are called ***image filtering.*** Thus, a single step of the Jacobi iteration is equivalent to an image filtering operation, which is discussed in Section 28.3.2.

External force field The external force accelerates the velocity field at each grid point:

$$\vec{w}_3 = \vec{w}_2 + \frac{\Delta t}{\rho}\vec{F}_{\text{external}} \,.$$

Projection So far, we calculated an updated velocity field $\vec{w}_3$ without considering the unknown pressure field. In the projection step, we compute the unknown pressure field p and update the velocity field with it:

$$\vec{v}(t) = \vec{w}_3 - \frac{\Delta t}{\rho}\vec{\nabla}p \,.$$

The pressure field is obtained from the requirement that the final velocity field must be divergence free. Let us apply the divergence operator to both sides of this equation. After this, the left side becomes zero since we aim at a divergence free vector field for which $\vec{\nabla}\cdot\vec{v} = 0$:

$$0 = \vec{\nabla}\cdot\left(\vec{w}_3 - \frac{\Delta t}{\rho}\vec{\nabla}p\right) = \vec{\nabla}\cdot\vec{w}_3 - \frac{\Delta t}{\rho}\vec{\nabla}^2 p \,.$$

Assuming a regular grid where vector field $\vec{w}_3$ is available, searching the unknown pressure at grid positions, and evaluating the divergence and the Laplacian with finite differences of equations (28.4) and (28.5), respectively, we again end up with a sparse linear system for the discrete pressure values and consequently for the difference between the final velocity field $\vec{v}$ and $\vec{w}_3$. This system is also solved with Jacobi iteration. Similarly to the diffusion step, the Jacobi iteration of the projection is also a simple image filtering operation.

Eulerian simulation on the GPU The discretized velocity and pressure fields can be conveniently stored in three-dimensional textures, where discrete variables are defined at the centers of elemental cubes, called ***voxels*** of a grid [60]. At each time step, the content of these data sets should be refreshed (Figure 28.9).

The representation of the fields in textures has an important advantage when the advection is computed. The advected field at voxel center $\vec{r}_i$ is obtained by copying the field value at position $\vec{r}_i - \vec{v}_i\Delta t$. Note that the computed position is not necessarily a voxel center, but it can be between the grid points. According to the finite element concept, this value can be generated from the finite element

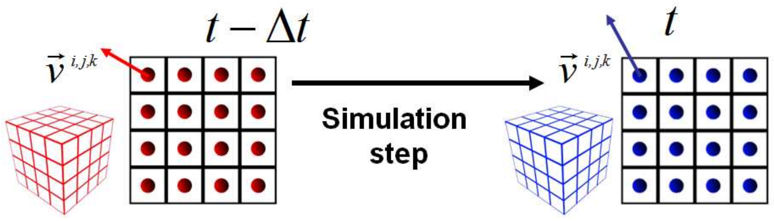
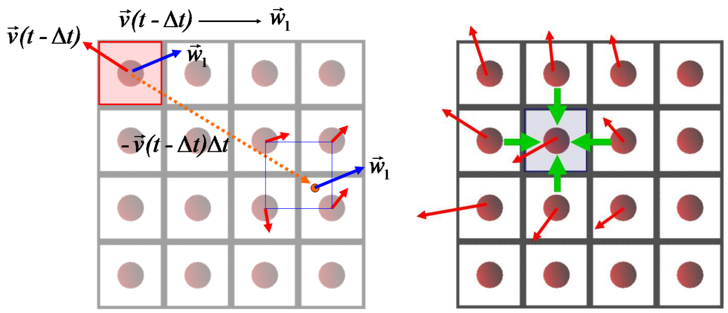


Figure 28.9 A time step of the Eulerian solver updates textures encoding the velocity field.


$\vec{v}(t - \Delta t)$
$\vec{v}(t - \Delta t) \longrightarrow \vec{w}_1$
$\vec{w}_1$
$-\vec{v}(t - \Delta t)\Delta t$
$\vec{w}_1$

Advection Jacobi iteration

Figure 28.10 Computation of the simulation steps by updating three-dimensional textures. Advection utilizes the texture filtering hardware. The linear equations of the viscosity damping and projection are solved by Jacobi iteration, where a texel (i.e. voxel) is updated with the weighted sum of its neighbors, making a single Jacobi iteration step equivalent to an image filtering operation.

representation of the data. If we assume piece-wise linear basis functions, then the texture filtering hardware automatically solves this problem for us at no additional computation cost.

The disadvantage of storing vector and scalar fields in three-dimensional textures is that the GPU can only read these textures no matter whether we take the graphics API or the GPGPU approach. The updated field must be written to the render target in case of the graphics API approach, and to the global memory if we use a GPGPU interface. Then, for the next simulation step, the last render target or global memory should be declared as an input texture.

In order to avoid write collisions, we follow a gathering approach and assign threads to each of the grid points storing output values. If GPUs fetch global data via textures, then the new value written by a thread becomes visible when the pass or the thread run is over, and the output is declared as an input texture for the next run. Thus, the computation of the time step should be decomposed to elemental update steps when the new output value of another grid point is needed. It means that we have and advection pass, a sequence of Jacobi iteration passes of the diffusion step, an external force calculation pass, and another sequence of Jacobi iteration passes of the projection step. With a GPGPU framework, a thread may directly read the

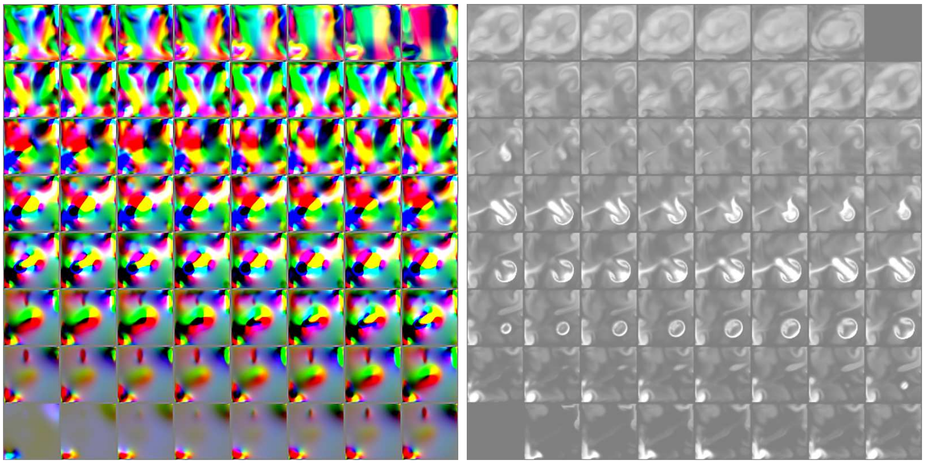


Figure 28.11 Flattened 3D velocity (left) and display variable (right) textures of a simulation.

data produced by another thread, but then synchronization is needed to make sure that the read value is already valid, so not the old but the new value is fetched. In such cases, synchronization points have the same role and passes or decomposed kernels.

In case of graphics APIs, there is one additional limitation. The render target can only be two-dimensional, thus either we flatten the layers of the three-dimensional voxel array into a large two-dimensional texture, or update just a single layer at a time. Flattened three-dimensional textures are shown by Figure 28.11. Once the textures are set up, one simulation step of the volume can be done by the rendering of a quad covering the flattened grid.

The graphics API approach has not only drawbacks but also an advantage over the GPGPU method, when the linear systems are solved with Jacobi iteration. The graphics API method runs the fragment shader for each grid point to update the solution in the texel associated with the grid point. However, if the neighbor elements of a particular grid point are negligible, we need less iteration steps than in a grid point where the neighbor elements are significant. In a quasi-SIMD machine like the GPU, iterating less in some of the processors is usually a bad idea. However, the exploitation of the ***early z-culling*** hardware helps to sidestep this problem and boosts the performance [146]. The z coordinate in the depth value is set proportionally to the maximum element in the neighborhood and to the iteration count. This way, as the iteration proceeds, the GPU processes less and less number of fragments, and can concentrate on important regions. According to our measurements, this optimization reduces the total simulation time by about 40 %.

When we wish to visualize the flow, we can also assume that the flow carries a scalar *display variable* with itself. The display variable is analogous with some paint or confetti poured into the flow. The display variable is stored in a float voxel array.

Using the advection formula for display variable D, its field can also be updated

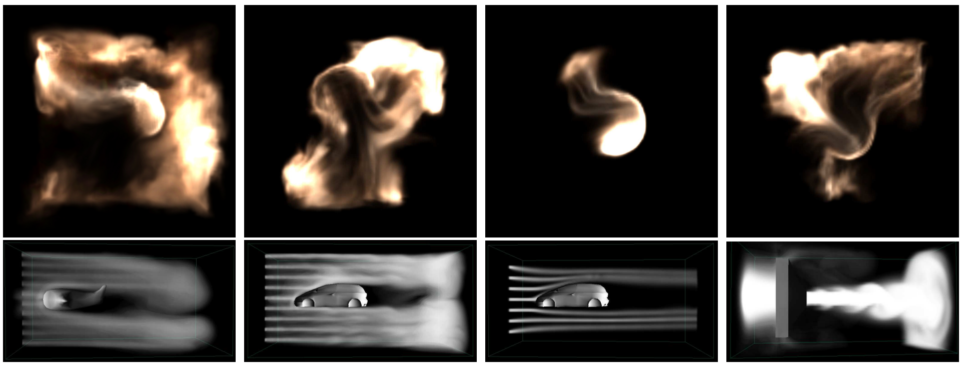


Figure 28.12 Snapshots from an animation rendered with Eulerian fluid dynamics.

in parallel with the simulation of time step Δt:

$$D(\vec{r}, t) = D(\vec{r} - \vec{v}\Delta t, t - \Delta t)\,.$$

At a time, the color and opacity of a point can be obtained from the display variable using a user controlled transfer function.

We can use a 3D texture slicing rendering method to display the resulting display variable field, which means that we place semi-transparent polygons perpendicular to the view plane and blend them together in back to front order (Figure 28.12). The color and the opacity of the 3D texture is the function of the 3D display variable field.

28.7.2. Lagrangian solver for differential equations

In the Lagrangian approach, the space is discretized by identifying **,** i.e. following just finite number of fluid elements. Let us denote the position and the velocity of the ith discrete fluid element by $\vec{r}_i$ and $\vec{v}_i$, respectively. We assume that all particles represent fluid elements of the same mass m, but as the density varies in space and will be the attribute of the particle, every particle is associated with a different volume $\Delta V_i = m/\rho_i$ of the fluid. The momentum conservation equation has the following form in this case:

$$\frac{\mathrm{d}\vec{r}_i}{\mathrm{d}t} = \vec{v}_i\,,$$

$$m\frac{\mathrm{d}\vec{v}_i}{\mathrm{d}t} = \left(-\vec{\nabla}p(\vec{r}_i) + \nu\vec{\nabla}^2\vec{v}(\vec{r}_i) + \vec{F}_{\mathrm{external}}(\vec{r}_i)\right)\Delta V_i\,. \tag{28.8}$$

If particles do not get lost during the simulation, the mass is automatically conserved. However, temporarily this mass may concentrate in smaller parts of the volume, so the simulated fluid is not incompressible. In Lagrangian simulation, we usually assume compressible gas.

From the knowledge of the system at discrete points, attributes are obtained at an arbitrary point via interpolation. Suppose we know an attribute A at the particle locations, i.e. we have $A_1, \ldots, A_N$. Attribute A is interpolated at location $\vec{r}$ by a weighted sum of contributions from the particles:

$$A(\vec{r}) = \sum_{i=1}^{N} A_i \Delta V_i W(|\vec{r} - \vec{r}_i|) ,$$

where ΔV_i is the volume represented by the particle in point $\vec{r}_i$, and $W(d)$ is a ***smoothing kernel,*** also called ***radial basis function,*** that depends on distance d between the particle location and the point of interest. From a different point of view, the smoothing kernel expresses how quickly the impact of a particle diminishes farther away. The smoothing kernel is normalized if smoothing preserves the total amount of the attribute value, which is the case if the kernel has unit integral over the whole volumetric domain. An example for the possible kernels is the *spiky kernel* of maximum radius h:

$$W(d) = \frac{15}{\pi h^6}(h-d)^3, \text{ if } 0 \le d \le h \text{ and zero otherwise} .$$

For normalized kernels, the particle density at point $\vec{r}_j$ is approximated as:

$$\rho_j = \rho(\vec{r}_j) = \sum_{i=1}^{N} mW(|\vec{r}_j - \vec{r}_i|) .$$

As each particle has the same mass m, the volume represented by particle j is

$$\Delta V_j = \frac{m}{\rho_j} = \frac{1}{\sum_{i=1}^{N} W(|\vec{r}_j - \vec{r}_i|)} .$$

According to the ***ideal gas law,*** the pressure is inversely proportional to the volume on constant temperature, thus at particle j the pressure is

$$p_j = \frac{k}{\Delta V_j} ,$$

where constant k depends on the temperature.

The pressure at an arbitrary point $\vec{r}$ is

$$p(\vec{r}) = \sum_{i=1}^{N} p_i \Delta V_i W(|\vec{r} - \vec{r}_i|) .$$

The acceleration due to pressure differences requires the computation of the gradient of the pressure field. As spatial variable $\vec{r}$ shows up only in the smoothing kernel, the gradient can be computed by using the gradient of the smoothing kernel:

$$\vec{\nabla} p(\vec{r}) = \sum_{i=1}^{N} p_i \Delta V_i \vec{\nabla} W(|\vec{r} - \vec{r}_i|) .$$

Thus, our first guess for the pressure force at particle j is:

$$\vec{F}_{\text{pressure},j} = -\vec{\nabla} p(\vec{r}_j) = -\sum_{i=1}^{N} p_i \Delta V_i \vec{\nabla} W(|\vec{r}_j - \vec{r}_i|) .$$

However, there is a problem here. Our approximation scheme could not guarantee to satisfy the physical rules including symmetry of forces and consequently the conservation of momentum. We should make sure that the force on particle i due to particle j is always equal to the force on particle j due to particle i. The symmetric relation can be ensured by modifying the pressure force in the following way:

$$\vec{F}_{\text{pressure},j} = -\sum_{i=1}^{N} \frac{p_i + p_j}{2} \Delta V_i \vec{\nabla} W(|\vec{r}_j - \vec{r}_i|) .$$

The viscosity term contains the Laplacian of the vector field, which can be computed by using the Laplacian of the smoothing kernel:

$$\vec{F}_{\text{viscosity},j} = \nu \vec{\nabla}^2 \vec{v} = \nu \sum_{i=1}^{N} \vec{v}_i \Delta V_i \vec{\nabla}^2 W(|\vec{r}_j - \vec{r}_i|) .$$

Similarly to the pressure force, a symmetrized version is used instead that makes the forces symmetric:

$$\vec{F}_{\text{viscosity},j} = \nu \sum_{i=1}^{N} (\vec{v}_i - \vec{v}_j) \Delta V_i \vec{\nabla}^2 W(|\vec{r}_j - \vec{r}_i|) .$$

External forces can be directly applied to particles. Particle-object collisions are solved by reflecting the velocity component that is perpendicular to the surface.

Having computed all forces, and approximating the time derivatives of (28.8) by finite differences, we may obtain the positions and velocities of each of the particles in the following way:

$$\begin{aligned} \vec{r}_i(t + \Delta t) &= \vec{r}_i(t) + \vec{v}_i(t)\Delta t , \\ \vec{v}_i(t + \Delta t) &= \vec{v}_i(t) + (\vec{F}_{\text{pressure},i} + \vec{F}_{\text{viscosity},i} + \vec{F}_{\text{external},i}) \Delta V_i \Delta t / m . \end{aligned}$$

Note that this is also a forward Euler integration scheme, which has stability problems. Instead of this, we should use a stable version, for example, the ***Verlet integration*** [40].

The Lagrangian approach tracks a finite number of particles where the forces acting on them depend on the locations and actual properties of other particles. Thus, to update a system of N particles, $O(N^2)$ interactions should be examined. Such tasks are generally referred to as the ***N-body problem.***

Lagrangian solver on the GPU In a GPGPU framework, the particle attributes can be stored in the global memory as a one-dimensional array or can be fetched via one-dimensional textures. In graphics API frameworks, particle attributes can only be represented by textures. The advantage of reading the data via textures is only the better caching since now we cannot utilize the texture filtering hardware. A gathering type method would assign a thread to each of the controlled particles, and a thread would compute the effect of other particles on its own particle. As the smoothing kernel has finite support, only those particles can interact with the considered one, which are not farther than the maximum radius of the smoothing filter.

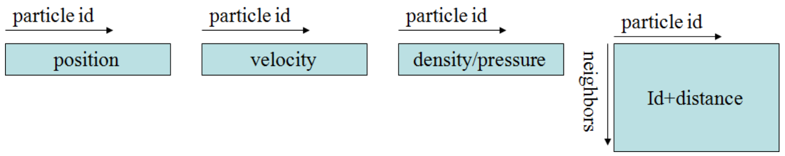


Figure 28.13 Data structures stored in arrays or textures. One-dimensional float3 arrays store the particles' position and velocity. A one-dimensional float2 texture stores the computed density and pressure. Finally, a two-dimensional texture identifies nearby particles for each particle.

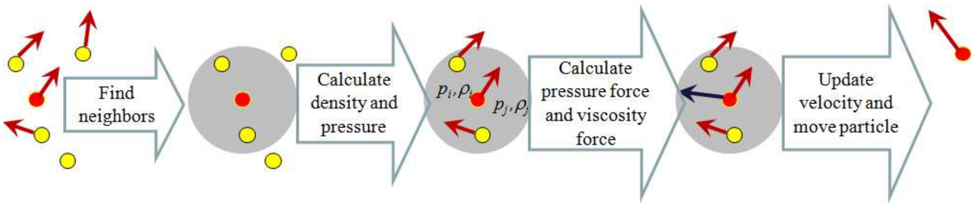

Find neighbors
Calculate density and pressure
p_i, ρ_i
p_j, ρ_j
Calculate pressure force and viscosity force
Update velocity and move particle

Figure 28.14 A time step of the Lagrangian solver. The considered particle is the red one, and its neighbors are yellow.

It is worth identifying these particles only once, storing them in a two-dimensional texture of in the global memory, and using this information in all subsequent kernels.

A GPGPU approach would need three one-dimensional arrays representing the particle position, velocity, density and pressure, and a two-dimensional array for the neighboring particles (Figure 28.13). In a graphics API approach, these are one- or two-dimensional textures. We can run a kernel or a fragment shader for each of the particles. In a GPGPU solution it poses no problem for the kernel to output a complete column of the neighborhood array, but in the fragment shaders of older GPUs the maximum size of a single fragment is limited. To solve this, we may limit the number of considered neighbor particles to the number that can be outputted with the available multiple render target option.

The processing of a single particle should be decomposed to passes or kernel runs when we would like to use the already updated properties of other particles (Figure 28.14). The first pass is the identification of the neighbors for each particles, i.e. those other particles that are closer than the support of the smoothing kernel. The output of this step is a two-dimensional array where columns are selected by the index of the considered particle and the elements in this column store the index and the distance of those particles that are close by.

The second pass calculates the density and the pressure from the number and the distance of the nearby particles. Having finished this pass, the pressure of every particle will be available for all threads. The third pass computes the forces from the pressure and the velocity of nearby particles. Finally, each particle gets its updated velocity and is moved to its new position.

Having obtained the particle positions, the system can be visualized by different

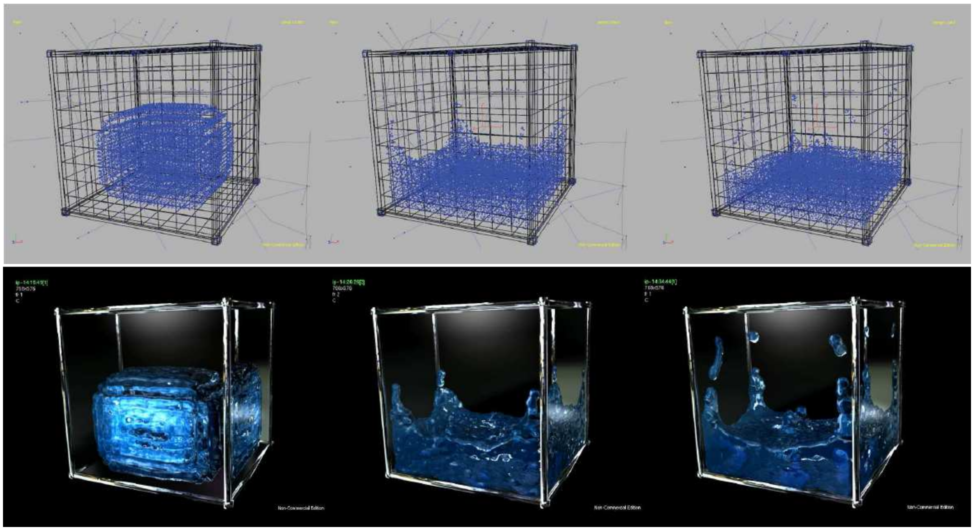


Figure 28.15 Animations obtained with a Lagrangian solver rendering particles with spheres (upper image) and generating the isosurface (lower image) [64].

methods. For example, we can render a point or a small sphere for each particle (upper image of Figure 28.15). Alternatively, we can splat particles onto the screen, resulting in a rendering style similar to that of the Eulerian solver (Figure 28.12). Finally, we can also find the surface of the fluid and compute reflections and refractions here using the laws of geometric optics (lower image of Figure 28.15). The surface of fluid is the isosurface of the density field, which is the solution of the following implicit equation:

$$\rho(\vec{r}) = \rho_{\text{iso}} \,.$$

This equation can be solved for points visible in the virtual camera by ***ray marching.*** We trace a ray from the eye position through the pixel and make small steps on it. At every sample position $\vec{r}_s$ we check whether the interpolated density $\rho(\vec{r}_s)$ has exceeded the specified isovalue ρ_{iso}. The first step when this happens is the intersection of the ray and the isosurface. The rays are continued from here into the reflection and refraction directions. The computation of these directions also requires the normal vector of the isosurface, which can be calculated as the gradient of the density field.

Exercises

28.7-1 Implement a game-of-life in CUDA. On a two-dimensional grid of cells, every cell is either *populated* of *unpopulated*. In every step, all cell states are re-evaluated. For populated cells:

- Each cell with one or no neighbors dies, as if by loneliness.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.

For unpopulated cells:

- Each cell with three neighbors becomes populated.

Store cell states in arrays accessible as textures. Always compute the next iteration state into a different output array. Start with a random grid and display results using the graphics API.

28.7-2 Implement a *wave equation solver*. The wave equation is a partial differential equation:

$$\frac{\partial^2 z}{\partial t^2} = c^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right),$$

where $z(x, y, t)$ is the wave height above point x, y in time t, and c is the speed of the wave.

Chapter Notes

The fixed transformation and multi-texturing hardware of GPUs became programmable vertex and fragment shaders about a decade ago. The high floating point processing performance of GPUs has quickly created the need to use them not only for incremental rendering but for other algorithms as well. The first GPGPU algorithms were also graphics related, e.g. ***ray tracing*** or the simulation of natural phenomena. An excellent review about the early years of GPGPU computing can be found in [113]. Computer graphics researchers have been very enthusiastic to work with the new hardware since its general purpose features allowed them to implement algorithms that are conceptually different from the incremental rendering, including the physically plausible light transport, called global illumination [143], physics simulation of rigid body motion with accurate collision detection, fluid dynamics etc., which made realistic simulation and rendering possible in real-time systems and games. The GPU Gems book series [47, 96, 123] and the ShaderX (currently GPU Pro [43]) series provide a huge collection of such methods.

Since the emergence of GPGPU platforms like CUDA and OpenCL, GPU solutions have showed up in all fields of high performance computing. Online warehouses of papers and programs are the gpgpu.org homepage and the NVIDIA homepage [111, 112], which demonstrate the wide acceptance of this approach in many fields. Without aiming at completeness, successful GPU applications have targeted high performance computing tasks including simulation of all kinds of physics phenomena, differential equations, tomographic reconstruction, computer vision, database searches and compression, linear algebra, signal processing, molecular dynamics and docking, financial informatics, virus detection, finite element methods, Monte Carlo methods, simulation of computing machines (CNN, neural networks, quantum computers), pattern matching, DNA sequence alignment, cryptography, digital holography, quantum chemistry, etc.

To get a scalable system that is not limited by the memory of a single GPU card, we can build GPU clusters. A single PC can be equipped with four GPUs and the number of interconnected PCs is unlimited [155]. However, in such systems the communication will be the bottleneck since current communication channels cannot compete with the computing power of GPUs.

29. Perfect Arrays

An (n, a, b)-perfect double cube is a $b \times b \times b$ sized n-ary periodic array containing all possible $a \times a \times a$ sized n-ary array exactly once as subarray. A growing cube is an array whose $c_j \times c_j \times c_j$ sized prefix is an (n_j, a, c_j)-perfect double cube for $j = 1, 2, \ldots$, where $c_j = n_j^{v/3}$, $v = a^3$ and $n_1 < n_2 < \cdots$. We construct the smallest possible perfect double cube (a $256 \times 256 \times 256$ sized 8-ary array) and growing cubes for any a.

29.1. Basic concepts

Cyclic sequences in which every possible sequence of a fixed length occurs exactly once have been studied for more than a hundred years [48]. The same problem, which can be applied to position localization, was extended to arrays [44].

Let $\mathbb{Z}$ be the set of integers. For $u,\ v \in \mathbb{Z}$ we denote the set $\{j \in \mathbb{Z} \mid u \leq j \leq v\}$ by $[u..v]$ and the set $\{j \in \mathbb{Z} \mid j \geq u\}$ by $[u..\infty]$. Let $d \in [1..\infty]$ and $k,\ n \in [2..\infty]$, $b_i,\ c_i,\ j_i \in [1..\infty]$ $(i \in [1..d])$ and $a_i,\ k_i \in [2..\infty]$ $(i \in [1..d])$. Let $\mathbf{a} = \langle a_1, a_2, \ldots, a_d \rangle$, $\mathbf{b} = \langle b_1, b_2, \ldots, b_d \rangle$, $\mathbf{c} = \langle c_1, c_2, \ldots, c_d \rangle$, $\mathbf{j} = \langle j_1, j_2, \ldots, j_d \rangle$ and $\mathbf{k} = \langle k_1, k_2, \ldots, k_d \rangle$ be vectors of length d, $\mathbf{n} = \langle n_1, n_2, \ldots \rangle$ an infinite vector with $2 \leq n_1 < n_2 < \cdots$.

A ***d*-dimensional *n*-ary array *A*** is a mapping $A : [1..\infty]^d \to [0, n-1]$.

If there exist a vector $\mathbf{b}$ and an array M such that

$$\forall \mathbf{j} \in [1..\infty]^d : A[\mathbf{j}] = M[(j_1 \bmod b_1) + 1, (j_2 \bmod b_2) + 1, \ldots, (j_d \bmod b_d) + 1],$$

then A is a **b*periodic array*** and M is a period of A.

The **a*-sized subarrays of A*** are the **a**-periodic n-ary arrays.

Although our arrays are infinite we say that a **b**-periodic array is **b*-sized.***

Indexset A_{index} of a **b**-periodic array A is the Cartesian product

$$A_{index} = \times_{i=1}^{d} [1..b_i] \,.$$

A d dimensional **b**-periodic n-ary array A is called ***$(n, d, \mathbf{a}, \mathbf{b})$-perfect,*** if all possible n-ary arrays of size **a** appear in A exactly once as a subarray.

Here n is the alphabet size, d gives the number of dimensions of the "window" and the perfect array M, the vector **a** characterizes the size of the window, and the vector **b** is the size of the perfect array M.

An $(n, d, \mathbf{a}, \mathbf{b})$-perfect array A is called ***c-cellular,*** if c_i divides b_i for $i \in [1..d]$.

A cellular array consists of $b_1/c_1 \times b_2/c_2 \times \cdots \times b_d/c_d$ disjoint subarrays of size $\mathbf{c}$, called ***cells.*** In each cell the element with smallest indices is called the ***head*** of the cell. The contents of the cell is called ***pattern.***

The product of the elements of a vector $\mathbf{a}$ is called the ***volume*** of the vector and is denoted by $|\mathbf{a}|$. The number of elements of the perfect array M is called the ***volume*** of M and is denoted by $|M|$.

If $b_1 = b_2 = \cdots = b_d$, then the $(n, d, \mathbf{a}, \mathbf{b})$-perfect array A is called **symmetric**. If A is symmetric and $a_1 = a_2 = \cdots = a_d$, then A is called **doubly symmetric.** If A is doubly symmetric and

1. $d = 1$, then A is called a ***double sequence;***
2. $d = 2$, then A is called a ***double square;***
3. $d = 3$, then A is called a ***double cube.***

According to this definition, all perfect sequences are doubly symmetric. In the case of symmetric arrays we use the notion $(n, d, \mathbf{a}, b)$ and in the case of doubly symmetric arrays we use (n, d, a, b) instead of $(n, d, \mathbf{a}, \mathbf{b})$.

The first known result originates from Flye-Sainte [48] who proved the existence of $(2, 1, a, 2^a)$-perfect sequences for all possible values of a in 1894.

One dimensional perfect arrays are often called de Bruijn [21] or Good [56] sequences. Two dimensional perfect arrays are called also perfect maps [115] or de Bruijn tori [65, 66, 69].

De Bruijn sequences of even length – introduced in [80] – are useful in construction of perfect arrays when the size of the alphabet is an even number and the window size is 2×2. Their definition is as follows.

If n is an even integer then an $(n, 1, 2, n^2)$-perfect sequence $M = (m_1, m_2, \ldots, m_{n^2})$ is called ***even,*** if $m_i = x$, $m_{i+1} = y$, $x \neq y, m_j = y$ and $m_{j+1} = x$ imply $j - i$ is even.

Iványi and Tóth [80] and later Hurlbert and Isaak [66] provided a constructive proof of the existence of even sequences.

Lexicographic indexing of an array $M = [m_{j_1 j_2 \ldots j_d}] = [m_{\mathbf{j}}]$ $(1 \leq j_i \leq b_i)$ for $i \in [1..d]$ means that the index $I(m_{\mathbf{j}})$ is defined as

$$I(m_{\mathbf{j}}) = j_1 - 1 + \sum_{i=2}^{d} \left((j_i - 1) \prod_{m=1}^{i-1} b_m \right) .$$

The concept of perfectness can be extended to infinite arrays in various ways. In ***growing arrays*** [66] the window size is fixed, the alphabet size is increasing and the prefixes grow in all d directions.

Let a and d be positive integers with $a \geq 2$ and $\mathbf{n} = \langle n_1, n_2, \ldots \rangle$ be a strictly increasing sequence of positive integers. An array $M = [m_{i_1 i_2 \ldots i_d}]$ is called ***$(\mathbf{n}, d, a)$-growing,*** if the following conditions hold:

1. $M = [m_{i_1 i_2 \ldots i_d}]$ $(1 \leq i_j < \infty)$ for $j \in [1..d]$;
2. $m_{i_1 i_2 \ldots i_d} \in [0..n - 1]$;

3. the prefix $M_k = [m_{i_1 i_2 \ldots i_d}]$ $(1 \leq i_j \leq n_k^{a^d/d} \mathit{for}\ j \in [1..d])$ of M is $(n_k, d, a, n_k^{a^d/d})$-perfect array for $k \in [0..\infty]$.

For the growing arrays we use the terms growing sequence, growing square and growing cube.

For $a, n \in [2..\infty]$ the ***new alphabet size*** $N(n, a)$ is

$$N(n,a) = \begin{cases} n, & \mathit{if\ any\ prime\ divisor\ of}\ a\ \mathit{divides}\ n\,, \\ nq, & \mathit{otherwise}\,, \end{cases} \tag{29.1}$$

where q is the product of the prime divisors of a not dividing n.

Note, that alphabet size n and new alphabet size N have the property that $n \mid N$, furthermore, $n = N$ holds in the most interesting case $d = 3$ and $n = a_1 = a_2 = a_3 = 2$.

The aim of this chapter is to prove the existence of a double cube. As a side-effect we show that there exist $(\mathbf{n}, d, a)$-growing arrays for any n, d and a.

29.2. Necessary condition and earlier results

Since in the period M of a perfect array A each element is the head of a pattern, the volume of M equals the number of the possible patterns. Since each pattern – among others the pattern containing only zeros – can appear only once, any size of M is greater then the corresponding size of the window. So we have the following necessary condition [32, 66]: If M is an $(n, d, \mathbf{a}, \mathbf{b})$-perfect array, then

$$|\mathbf{b}| = n^{|\mathbf{a}|} \tag{29.2}$$

and

$$b_i > a_i \text{ for } i \in [1..d]\,. \tag{29.3}$$

Different construction algorithms and other results concerning one and two dimensional perfect arrays can be found in the fourth volume of *The Art of Computer Programming* written by D. E. Knuth [84]. E.g. a (2,1,5,32)-perfect array [84, page 22], a 36-length even sequence whose 4-length and 16-length prefixes are also even sequences [84, page 62], a (2,2,2,4)-perfect array [84, page 38] and a (4,2,2,16)-perfect array [84, page 63].

It is known [21, 84] that in the one-dimensional case the necessary condition (29.2) is sufficient too. There are many construction algorithms, like the ones of Cock [32], Fan, Fan, Ma and Siu [44], Martin [100] or any algorithm for constructing of directed Euler cycles [97].

Chung, Diaconis and Graham [30] posed the problem to give a necessary and sufficient condition of the existence of $(n, 2, \mathbf{a}, \mathbf{b})$-perfect arrays.

The conditions (2) and (3) are sufficient for the existence of (2,2,**a**,**b**)-perfect arrays [44] and (n,2,a,b)-perfect arrays [114]. Later Paterson in [115, 116] supplied further sufficient conditions.

Hurlbert and Isaak [66] gave a construction for one and two dimensional growing arrays.

29.3. One-dimensional arrays

In the construction of one-dimensional perfect arrays we use the following algorithms.

Algorithm Martin generates one-dimensional perfect arrays. Its inputs are the alphabet size n and the window size a. Its output is an n-ary perfect sequence of length n^a. The output begins with a zeros and always continues with the maximal permitted element of the alphabet.

29.3.1. Pseudocode of the algorithm Quick-Martin

A natural implementation of Martin's algorithm can be found in the chapter *Complexity of words* of this book. The following effective implementation of Martin is due to M. Horváth and A. Iványi.

```
Quick-Martin($n, a$)

1 for $i = 0$ to $n^{a-1} - 1$
2       $C[i] = n - 1$
3 for $i = 1$ to $a$
4       $w[i] = 0$
5 for $i = a + 1$ to $n^a$
6       $k = w[i - a + 1]$
7       for $j = 1$ to $a - 1$
8             $k = kn + w[i - a + j]$
9       $w[i] = C[k]$
10      $C[k] = C[k] - 1$
11 return w
```

This algorithm runs in $\Theta(an^a)$ time. The following implementation of Martin algorithm requires even smaller time.

29.3.2. Pseudocode of the algorithm Optimal-Martin

```
Optimal-Martin($n, a$)

1 for $i = 0$ to $n^{a-1} - 1$
2       $C[i] = n - 1$
3 for $i = 1$ to $a$
4       $w[i] = 0$
5 for $i = a + 1$ to $n^a$
6       $k = w[i - a + 1]$
7       for $j = 1$ to $a - 1$
8             $k = kn + w[i - a + j]$
9       $w[i] = C[k]$
10      $C[k] = C[k] - 1$
11 return w
```

The running time of any algorithm which constructs a on??? perfect array is $\Omega(n^a)$, since the sequance contains n^a elements. The running time of Optimal-

Martin is $\Theta(n^a)$.

29.3.3. Pseudocode of the algorithm Shift

Algorithm Shift proposed by Cook in 1988 is a widely usable algorithm to construct perfect arrays. We use it to transform cellular $(N, d, a, \mathbf{b})$-perfect arrays into $(N, d+1, a, \mathbf{c})$-perfect arrays.

```
Shift($N, d, a, P_d, P_{d+1}$)

1 Martin($N^{a^d}, a-1, \mathbf{w}$)
2 for $j = 0$ to $N^{a^d - a^{d-1}} - 1$
3       transform $w_i$ to an $a^d$ digit $N$-ary number
4       produce the $(j+1)$-st layer of the output $P_{d+1}$ by multiple shifting
        the $j$th layer of $P_d$ by the transformed number (the first $a$ digits
        give the shift size for the first direction, then the next $a^2 - a$ digits
        in the second direction etc.)
5 return $P_{d+1}$
```

29.3.4. Pseudocode of the algorithm Even

If N is even, then this algorithm generates the N^2-length prefix of an even growing sequence [66].

```
Even($N, \mathbf{w}$)

1 if $N == 2$
2     $w[1] = 0$
3     $w[2] = 0$
4     $w[3] = 1$
5     $w[4] = 1$
6     return $\mathbf{w}$
7 for $i = 1$ to $N/2 - 1$
8       for $j = 0$ to $2i - 1$
9             $w[4i^2 + 2j + 1] = j$
10      for $j = 0$ to $i - 1$
11            $w[4i^2 + 2 + 4j] = 2i$
12      for $j = 0$ to $i - 1$
13            $w[4i^2 + 4 + 4j] = 2i + 1$
14      for $j = 0$ to $4i - 1$
15            $w[4i^2 + 4i + 1 + j] = w[4i^2 + 4i - j]$
16      $w[4i^2 + 8i + 1] = 2i + 1$
17      $w[4i^2 + 8i + 2] = 2i$
18      $w[4i^2 + 8i + 3] = 2i$
19      $w[4i^2 + 8i + 4] = 2i + 1$
20 return $\mathbf{w}$
```

Algorithm Even [66] produces even de Bruijn sequences.

29.4. One dimensional words with fixed length

29.5. Two-dimensional infinite arrays

Chung, Diaconis and Graham posed the problem to give a necessary and sufficient condition of the existence of $(n, 2, \mathbf{a}, \mathbf{b})$-perfect arrays.

As Fan, Fan and Siu proved in 1985, the conditions (2) and (3) are sufficient for the existence of (2,2,**a**,**b**)-perfect arrays. Paterson proved the same in 1994 for $(n, 2, a, b)$-perfect arrays. leter Paterson supplied further sufficient conditions.

Hurlbert and Isaak in 1993 gave a construction for one and two dimensional growing arrays.

29.5.1. Pseudocode of the algorithm Mesh

The following implementation of Mesh is was proposed by Iványi and Tóth in 1988.

```
Mesh($N, \mathbf{w}, S$)

1 for $i = 1$ to $N^2$
2     for $j = 1$ to $N^2$
3         if   $i + j$ is even
4              $S[i, j] = w[i]$
5         else $S[i, j] = w[j]$
6 return $S$
```

29.5.2. Pseudocode of the algorithm Cellular

This is an extension and combination of the known algorithms Shift, Martin, Even and Mesh.

Cellular results cellular perfect arrays. Its input data are n, d and $\mathbf{a}$, its output is an $(N, d, \mathbf{a}, \mathbf{b})$-perfect array, where $b_1 = N^{a_1}$ and $b_i = N^{a_1 a_2 \ldots a_i - a_1 a_2 \ldots a_{i-1}}$ for $i = 2, 3, \ldots, d$. Cellular consists of five parts:

1. *Calculation* (line 1 in the pseudocode) determining the new alphabet size N using formula (29.1);

2. *Walking* (lines 2–3) if $d = 1$, then construction of a perfect symmetric sequence S_1 using algorithm Martin (walking in a de Bruijn graph);

3. *Meshing* (lines 4–6) if $d = 2$, N is even and $a = 2$, then first construct an N-ary even perfect sequence $\mathbf{e} = \langle e_1, e_2, \ldots, e_{N^2} \rangle$ using Even, then construct an $N^2 \times N^2$ sized N-ary square S_1 using meshing function (**??**);

4. *Shifting* (lines 7–12) if $d > 1$ and (N is odd or $a > 2$), then use Martin once, then use Shift $d - 1$ times, receiving a perfect array P;

5. *Combination* (lines 13–16) if $d > 2$, N is even and $a = 2$, then construct an even sequence with Even, construct a perfect square by Mesh and finally use of Shift $d - 2$ times, results a perfect array P.

```
Cellular($n, d, a, N, A$)

$N = N(n, a)$
if $d = 1$
    Martin($N, d, a, A$)
return $A$
if $d == 2$ and $a == 2$ and $N$ is even
    Mesh($N, a, A$)
    return $A$
if $N$ is odd or $a \neq 2$
    Martin($N, a, P_1$)
    for $i = 1$ to $d - 1$
        Shift($N, i, P_i, P_{i+1}$)
        $A = P_1$
    return $A$
Mesh($N, a, P_1$)
for $i = 2$ to $d - 1$
    Shift($N, i, P_i, P_{i+1}$)
$A \leftarrow P_d$
return $P_d$
```

29.6. Three-dimensional infinite cubes

29.6.1. Pseudocode of the algorithm Colour

Colour transforms cellular perfect arrays into larger cellular perfect arrays. Its input data are

- $d \geq 1$ – the number of dimensions;
- $N \geq 2$ – the size of the alphabet;
- $\mathbf{a}$ – the window size;
- $\mathbf{b}$ – the size of the cellular perfect array A;
- A – a cellular $(N, d, \mathbf{a}, \mathbf{b})$-perfect array.
- $k \geq 2$ – the multiplication coefficient of the alphabet;
- $\langle k_1, k_2, \ldots, k_d \rangle$ – the extension vector having the property $k^{|\mathbf{a}|} = k_1 \times k_2 \times \cdots \times k_d$.

The *output* of Colour is

- a (kN)-ary cellular perfect array P of size $\mathbf{b} = \langle k_1 a_1, k_2 a_2, \ldots, k_d a_d \rangle$.

Colour consists of three steps:

1. *Blocking:* (line 1) arranging $k^{|\mathbf{a}|}$ copies (blocks) of a cellular perfect array A into a rectangular array R of size $\mathbf{k} = k_1 \times k_2 \times \cdots \times k_d$ and indexing the blocks lexicographically (by $0,\ 1, \ldots, k^{|\mathbf{a}|} - 1$);

2. *Indexing:* (line 2) the construction of a lexicographic indexing scheme I containing the elements $0, 1, \ldots k^{|a|} - 1$ and having the same structure as the array R, then construction of a colouring matrix C, transforming the elements of I into k-ary numbers consisting of $|\mathbf{a}|$ digits;

3. *Colouring:* (lines 3-4) colouring R into a symmetric perfect array P using the colouring array C that is adding the N-fold of the j-th element of C to each cell of the j-th block in R (considering the elements of the cell as lexicographically ordered digits of a number).

The output P consists of blocks, blocks consist of cells and cells consists of elements. If $e = P[\mathbf{j}]$ is an element of P, then the lexicographic index of the block containing e is called the ***blockindex*** of e, the lexicographic index of the cell containing e is called the ***cellindex*** and the lexicographic index of e in the cell is called ***elementindex***. E.g. the element $S_2[7, 6] = 2$ in Table 3 has blockindex 5, cellindex 2 and elementindex 1.

Input parameters are $N,\ d,\ a,\ k,\ \mathbf{k}$, a cellular $(N, d, a, \mathbf{b})$-perfect array A, the output is a $(kN, d, \mathbf{a}, \mathbf{c})$-perfect array P, where $\mathbf{c} = \langle a_1k_1, a_2k_2, \ldots, a_dk_d\rangle$.

```
Colour($N, d, \mathbf{a}, k, \mathbf{k}, A, P$)
1 arrange the copies of $P$ into an array $R$ of size
  $k_1 \times k_2 \times \cdots \times k_d$ blocks
2 construct a lexicographic indexing scheme $I$ containing the elements
  of $[0..k^{a^d} - 1]$ and having the same structure as $R$
3 construct an array $C$ transforming the elements of $I$ into $k$-ary
  numbers of $v$ digits and multiplying them by $N$
4 produce the output $S$ adding the $j$-th ($j \in [0..k^{a^d} - 1]$) element of $C$
  to each cell of the $j$-th block in $R$ for each block of $R$
5 return $S$
```

29.6.2. Pseudocode of the algorithm Growing

Finally, algorithm Growing generates a prefix S_r of a growing array G. Its input data are r, the number of required doubly perfect prefixes of the growing array G, then n, d and $\mathbf{a}$. It consists of the following steps:

1. *Initialization*: construction of a cellular perfect array P using Cellular;

2. *Resizing:* if the result of the initialization is not doubly symmetric, then construction of a symmetric perfect array S_1 using Colour, otherwise we take P as S_1;

3. *Iteration*: construction of the further $r - 1$ prefixes of the growing array G repeatedly, using Colour.

Input parameters of Growing are $n,\ d,\ a$ and r, the output is a doubly symmetric perfect array S_r, which is the rth prefix of an $(\mathbf{n}, d, a)$-growing array.

Growing(n, d, a, r, S_r)

```
CELLULAR($n, d, a, N, P$)
calculation of $N$ using formula (29.1)
if $P$ is symmetric
    $S_1 = P$
if $P$ is not symmetric
    $n_1 = N^{d/\gcd(d,a^d)}$
    $k = n_1/N$
    $k_1 = (n_1)^{a^d/3}/N^a$
    for $i = 2$ to $d$
        $k_i = (n_1)^{a^d/d}/N^{a^i - a^{i-1}}$
        COLOUR($n_1, d, a, k, \mathbf{k}, P, S_1$)
$k = N^d/\gcd(d, a^d)$
for $i = 1$ to $d$
    $k_i = (n_2)^{a^d/d}/N^{a^i - a^{i-1}}$
for $i = 2$ to $r$
    $n_i = N^{di/\gcd(d,a^d)}$
    COLOUR($n_i, d, \mathbf{a}, k, \mathbf{k}, S_{i-1}, S_i$)
return $S_r$
```

29.7. Examples of constructing growing arrays using colouring

In this section particular constructions are presented.

29.7.1. Construction of growing sequences

As the first example let $n = 2$, $a = 2$ and $r = 3$. CELLULAR calculates $N = 2$ and MARTIN produces the cellular (2,1,2,4)-perfect sequence $P = 00|11$.

Since P is symmetric, $S_1 = P$. Now GROWING chooses multiplication coefficient $k = n_2/n_1 = 2$, extension vector $\mathbf{k} = \langle 4\rangle$ and uses COLOUR to construct a 4-ary perfect sequence.

COLOUR arranges $k_1 = 4$ copies into a 4 blocks sized arrray receiving

$$R = 00|11 \mid\mid 00|11 \mid\mid 00|11 \mid\mid 00|11. \tag{29.4}$$

COLOURING receives the indexing scheme $I = 0\ 1\ 2\ 3$, and the colouring matrix C transforming the elements of I into a digit length k-ary numbers: $C = 00 \mid\mid 01 \mid\mid 10 \mid\mid 11$.

Finally we colour the matrix R using C – that is multiply the elements of C by n_1 and adding the j-th ($j = 0, 1, 2, 3$) block of $C_1 = n_1C$ to both cells of the j-th copy in R:

$$S_2 = 00|11 \mid\mid 02|13 \mid\mid 20|31 \mid\mid 22|33. \tag{29.5}$$

Since $r = 3$, we use COLOUR again with $k = n_3/n_2 = 2$ and get the (8,1,2,64)-perfect sequence S_3 repeating S_2 4 times, using the same indexing array I and

29.1. Table a) A (2,2,4,4)-square b) Indexing scheme I of size 4×4

column/row	1	2	3	4
1	0	0	0	1
2	0	0	1	0
3	1	0	1	1
4	0	1	1	1

column/row	1	2	3	4
1	0	1	2	3
2	4	5	6	7
3	8	9	10	11
4	12	13	14	15

29.2. Table Binary colouring matrix C of size 8×8

column/row	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	1	1	0	1	1
3	0	1	0	1	0	1	0	1
4	0	0	0	1	1	0	1	1
5	1	0	1	0	1	0	1	0
6	0	0	0	1	1	0	1	1
7	1	1	1	1	1	1	1	1
8	0	0	0	1	1	0	1	1

colouring array $C' = 2C$.

Another example is $a = 2$, $n = 3$ and $r = 2$. To guarantee the cellular property now we need a new alphabet size $N = 6$. Martin produces a (6,1,2,36)-perfect sequence S_1, then Colour results a (12,1,2,144)-perfect sequence S_2.

29.7.2. Construction of growing squares

Let $n = a = 2$ and $r = 3$. Then $N(2,2) = 2$. We construct the even sequence $W_4 = e_1e_2e_3e_4 = 0\ 0\ 1\ 1$ using Even and the symmetric perfect array A in Table 29.1.a using the meshing function (**??**). Since A is symmetric, it can be used as S_1. Now the greatest common divisor of a and a^d is 2, therefore indeed $n_1 = N^{2/2} = 2$.

Growing chooses $k = n_1/N = 2$ and Colour returns the array R repeating the array A $k^2 \times k^2 = 4 \times 4$ times.

Colour uses the indexing scheme I containing k^4 indices in the same 4×4 arrangement as it was used in R. Table 29.1.b shows I.

Transformation of the elements of I into 4-digit k-ary form results the colouring matrix C represented in Table 29.2.

Colouring of array R using the colouring array $2C$ results the (4,2,2,16)-square S_2 represented in Table 29.3.

In the next iteration Colour constructs an 8-ary square repeating S_2 4×4 times, using the same indexing scheme I and colouring by $4C$. The result is S_3, a $(8, 2, 2, 64)$-perfect square.

29.3. Table A (4,2,2,16)-square generated by colouring

column/row	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	2	1	2	2	0	3	0	2	2	3	2
3	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
4	0	1	1	1	0	3	1	3	2	1	3	1	2	3	3	3
5	0	2	0	3	0	2	0	3	0	2	0	3	0	2	0	3
6	0	0	1	0	0	2	1	2	2	0	3	0	2	2	3	2
7	1	2	1	3	1	2	1	3	1	2	1	3	1	2	1	3
8	0	1	1	1	0	3	1	3	2	1	3	1	2	3	3	3
9	2	0	2	1	2	0	2	1	2	0	2	1	2	0	2	1
10	0	0	1	0	0	2	1	2	2	0	3	0	2	2	3	2
11	3	0	3	1	3	0	3	1	3	0	3	1	3	0	3	1
12	0	1	1	1	0	3	1	3	2	1	3	1	2	3	3	3
13	2	2	2	3	2	2	2	3	2	2	2	3	2	2	2	3
14	0	0	1	0	0	2	1	2	2	0	3	0	2	2	3	2
15	3	2	3	3	3	2	3	3	3	2	3	3	3	2	3	3
16	0	1	1	1	0	3	1	3	2	1	3	1	2	3	3	3

29.4. Table 8 layers of a (2,3,2,16)-perfect array

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Layer 7
0 0 0 1	0 0 0 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1
0 0 1 0	0 0 1 0	0 0 0 1	0 0 0 1	0 1 1 1	0 1 1 1	0 1 1 1	0 1 1 1
1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1
0 1 1 1	0 1 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	0 1 1 1

29.7.3. Construction of growing cubes

If $d = 3$, then the necessary condition (2) is $b^3 = (n)^{a^3}$ for double cubes, implying n is a cube number or a is a multiple of 3. Therefore, either $n \geq 8$ and then $b \geq 256$, or $a \geq 3$ and so $b \geq 512$, that is, the smallest possible perfect double cube is the (8, 3, 2, 256)-cube.

As an example, let $n = 2$, $a = 2$ and $r = 2$. Cellular computes $N = 2$, Mesh constructs the $(2, 2, 2, 4)$-perfect square in Table 29.1.a, then Shift uses Martin with $N = 16$ and $a = 1$ to get the shift sizes for the layers of the $(2, 3, 2, \mathbf{b})$-perfect output P of Cellular, where $\mathbf{b} = \langle 4, 4, 16\rangle$. Shift uses P as zeroth layer and the jth ($j \in [1 : 15]$) layer is generated by cyclic shifting of the previous layer downwards by w_i (div 4) and right by w_i (mod 4), where $\mathbf{w} = \langle 0\ 15\ 14\ 13\ 12\ 11\ 10\ 9\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\rangle$. 8 layers of P are shown in Table 29.4.

Let A_3 be a $4 \times 4 \times 16$ sized perfect, rectangular matrix, whose 0. layer is the matrix represented in Table 29.1, and the $(2, 3, \mathrm{a}, \mathrm{b})$-perfect array P in Table 29.4, where $\mathrm{a} = (2, 2, 2)$ and $\mathrm{b} = (4, 4, 8)$.

Growing uses Colour to retrieve a doubly symmetric cube. $n_1 = 8$, thus

$b = 256$, $k = n_1/N = 4$ and $\mathbf{k} = \langle 256/4, 256/4, 256/64\rangle$, that is we construct the matrix R repeating P $64 \times 64 \times 16$ times.

I has the size $64 \times 64 \times 16$ and $I[i_1, i_2, i_3] = 64^2(i_1 - 1) + 64(i_2 - 1) + i_3 - 1$. Colour gets the colouring matrix C by transforming the elements of I into 8-digit 4-ary numbers – and arrange the elements into $2 \times 2 \times 2$ sized cubes in lexicographic order – that is in order (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1). Finally colouring results a double cube S_1.

S_1 contains 2^{24} elements therefore it is presented only in electronic form (on the homepage of the corresponding author).

If we repeat the colouring again with $k = 2$, then we get a 64-ary $65536 \times 64536 \times 64536$ sized double cube S_2.

29.8. Proof of the main result

The main result of this paper can be formulated as follows.

Theorem 29.1 *If $n \geq 2$, $d \geq 1$, $a \geq 2$, $n_j = N^{dj/\gcd(d,a^d)}$ with $N = N(n, a)$ given by (1) for $j \in [0..\infty]$, then there exists an $(\mathbf{n}, d, a)$-growing array.*

The proof is based on the following lemmas.

Lemma 29.2 (Cellular lemma) *If $n \geq 2$, $d \geq 1$ and $a \geq 2$, then algorithm* Cellular *produces a cellular $(N, d, a, \mathbf{b})$-perfect array A, where N is determined by formula (29.1), $b_1 = N^a$ and $b_i = N^{a^i - a^{i-1}}$* $(i \in [2..d])$.

Proof It is known that algorithms Even+Mesh and Martin+Shift result perfect outputs.

Since Mesh is used only for even alphabet size and for 2×2 sized window, the sizes of the constructed array are even numbers and so the output array is cellular.

In the case of Shift we exploit that all prime divisors of a divide the new alphabet size N, and $b_i = N^{(a-1)(a^{i-1})}$ and $(a - 1)(a^{i-1}) \geq 1$. ■

Lemma 29.3 (Indexing lemma) *If $n \geq 2$, $d \geq 2$, $k \geq 2$, C is a d dimensional $\mathbf{a}$-cellular array with $|\mathbf{b}| = k^{|\mathbf{a}|}$ cells and each cell of C contains the corresponding cellindex as an $|\mathbf{a}|$ digit k-ary number, then any two elements of C having the same elementindex and different cellindex are heads of different patterns.*

Proof Let P_1 and P_2 be two such patterns and let us suppose they are identical. Let the head of P_1 in the cell have cellindex g and head of P_2 in the cell have cellindex h (both cells are in array C). Let $g - h = u$.

We show that $u = 0 \pmod{k^{|b|}}$. For example in Table 2 let the head of P_1 be $(2, 2)$ and the head of P_2 be $(2, 6)$. Then these heads are in cells with cellindex 0 and 2 so here $u = 2$.

In both cells, let us consider the position containing the values having local value 1 of some number (in our example they are the elements (3,2) and (3,6) of C.) Since these elements are identical, then $k|u$. Then let us consider the positions with local values k (in our example they are (3,1) and (3,5).) Since these elements are also identical so $k^2|u$. We continue this way up to the elements having local value $k^{|b|}$ and get $k^{|b|}|u$, implying $u = 0$.

This contradicts to the conditon that the patterns are in different cells. ■

Lemma 29.4 (Colouring lemma) *If* $k \geq 2$, $k_i \in [2..\infty]$ $(i \in [1..d])$, A *is a cellular* $(n, d, \mathbf{a}, \mathbf{b})$*-perfect array, then algorithm* Colour$(N, d, \mathbf{a}, k, \mathbf{k}, A, S)$ *produces a cellular* $(kN, d, \mathbf{a}, \mathbf{c})$*-perfect array* P*, where* $\mathbf{c} = \langle k_1a_1, k_2a_2, \dots, k_da_d\rangle$.

Proof The input array A is N-ary, therefore R is also N-ary. The colouring array C contains the elements of $[0..N(k-1)]$, so elements of P are in $[0..kN-1]$.

The number of dimensions of S equals to the number of dimensions of P that is, d.

Since A is cellular and c_i is a multiple of b_i $(i \in [1..d])$, P is cellular.

All that has to be shown is that the patterns in P are different.

Let's consider two elements of P as heads of two windows and their contents – patterns p and q. If these heads have different cellindex, then the considered patterns are different due to the periodicity of R. E.g. in Table 29.3 $P[11, 9]$ has cellindex 8, the pattern headed by $P[9, 11]$ has cellindex 2, therefore they are different (see parity of the elements).

If two heads have identical cellindex but different blockindex, then the indexing lemma can be applied. ■

Proof of the main theorem. Lemma 18 implies that the first call of Colour in line 10 of Growing results a doubly symmetric perfect output S_1. In every iteration step (in lines 14–16 of Growing) the nzeroth block of S_i is the same as S_{i-1}, since the zeroth cell of the colouring array is filled up with zeros.

Thus S_1 is transformed into a doubly symmetric perfect output S_r having the required prefixes S_1, S_2, $\dots, S_{r-1}$. ■

29.9. Multi-dimensional infinite arrays

Chapter Notes

For Section 29.3:

For Section 29.4:

For Section 29.5:

For Section 29.6:

[5] [30] [32]

[33] [34]

[21] [38] [48] [56] [63]

[65] [66] [67] [68]

[69] [73]

[74] [75] [76] [80] [85]
[97] [100] [103] [114]
[115] [116] [117] [118]
[131] [153]
For Section 29.9:

30. Score Sets and Kings

The idea of comparison-based ranking has been discussed earlier in the chapter *Comparison based ranking*, where score sequence was introduced as a way of ranking vertices in a tournament. Oriented graphs are generalizations of tournaments. In fact, just like one can think of a tournament as expressing the results of a round-robin competition without ties (with vertices representing players and arrows pointing to the defeated players), one can think of an oriented graph as a round-robin competition with ties allowed (ties are represented by not drawing the corresponding arcs). Figure 30.1 shows the results of a round-robin competition involving 4 players a, b, c

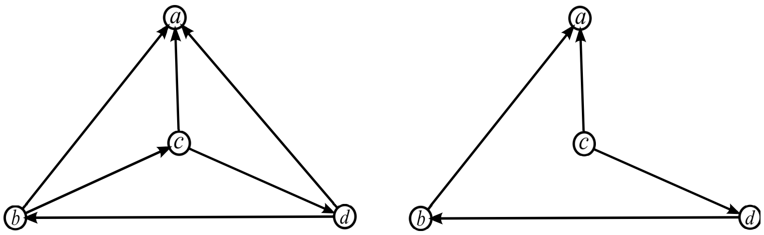


Figure 30.1 A round-robin competition involving 4 players.

and d, with (a) ties not allowed and (b) ties allowed. In the first instance there is always a winner and a loser whenever two players square off, while in the latter case player a ties with player d and player b ties with player c.

In 2009 Antal Iványi studied directed graphs, in which every pair of different vertices is connected with at least a and at most b arcs. He named them (a, b, n)-tournaments or simply (a, b)-tournament.

If $a = b = k$, then the (a, b)-tournaments are called k-tournaments. In this chapter we deal first of all with 1-tournaments and $(0, 1)$-tournaments. $(0, 1)$-tournaments are in some sense equivalent with $(2, 2)$-tournaments. We use the simple notations 1-tournament T_n^1, 2-tournament T_n^2, ..., k-tournament T_n^k, ... It is worth mentioning that T_n^1 is a classical tournament, while oriented graphs are $(0, 1)$-tournaments. If we allow loops then every directed graph is some (a, b, n)-tournament (see the Chapter **??** (*Comparison Based Ranking*) of this book).

We discuss two concepts related with (a, b)-tournaments, namely score sets and kings. A *score set* is just the set of different scores (out-degrees) of vertices, while a *king* is a dominant vertex. We shall study both concepts for 1-tournaments first and then extend these to the more general setting of oriented graphs.

Although we present algorithms for finding score sets and kings in 1-tournaments and $(0, 1)$-tournaments, much of the focus is on constructing tournaments with special properties such as having a prescribed score set or a fixed number of kings. Since players in a tournament are represented by vertices, we shall use the words player and vertex interchangeably throughout this chapter without affecting the meaning.

We adopt the standard notation $T(V, A)$ to denote a tournament with vertex set V and arc set A. We denote the number of vertices by n, and the out-degree matrix by $\mathcal{M}$, and the in-degree matrix by N. Furthermore, we use the term n-tournament and the notation T_n^k to represent a tournament with n vertices and exactly k arcs between the elements of any pair of different vertices. In a similar way R_n^k and N_n denote a regular, resp. a null graph. When there is no ambiguity we omit one or even both indices shall refer to the corresponding tournaments as T, R. and N.

In Section 30.1 the score sets of 1-tournaments are discussed, while Section 30.2 deals with the sore sets of oriented graphs. In Section 30.3 the conditions of the unique reconstruction of the score sets are considered at first for k-tournaments, then in more details for 1-tournaments and 2-tournaments. In Section 30.4 and Section 30.5 results connected with different kings of tournaments are presented.

Some long and accessible proofs are omitted. In these cases the Reader can find the coordinates of the proof in *Chapter notes* and *Bibliography*.

30.1. Score sets in 1-tournaments

In a round-robin competition with no ties allowed, what are the sets of nonnegative integers that can arise as scores of players? Note that here we are not interested in the scores of individual players (the score sequence), rather we are looking for the sets of nonnegative integers with each integer being the score of at least one player in the tournament. This question motivates the study of *score sets* of tournaments.

The set of different scores of vertices of a tournament is called the ***score set*** of the tournament. In other words, the score set is actually the score sequence of a tournament with repetitions removed. For example the tournament given in Figure 30.2 has score sequence $[0, 2, 2, 2]$, whereas the score set of this tournament is $\{0, 2\}$. Figure 30.3 shows the out-degree matrix of the tournament represented on Figure 30.2.

30.1.1. Determining the score set

Determining the score set of a tournament $T(V, A)$ is quite easy. The following algorithm SET1 takes the data of a tournament $T(V, A)$ as input and returns the score set $\mathcal{S}$ of T.

The procedures of this chapter are written according to the third edition of the textbook *Introduction to Algorithms* published by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein in 2009.

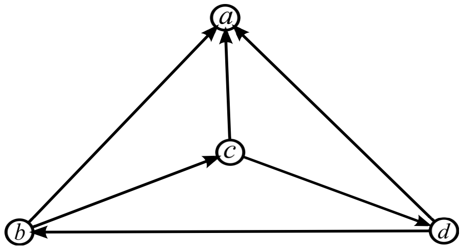


Figure 30.2 A tournament with score set $\{0, 2\}$.

vertex/vertex	a	b	c	d	Score
a	—	0	0	0	0
b	1	—	1	0	2
c	1	0	—	1	2
d	1	1	0	—	2

Figure 30.3 Out-degree matrix of the tournament represented in Figure 30.2.

```
SET1($n, V, A$)
1  $\mathcal{S} = \emptyset$
2  for all vertex $u \in V$
3        $s = 0$
4        for all vertex $v \in V$
5              if $(u, v) \in A$                          // is $(u, v)$ an arc of $T$?
6                   $s = s + 1$
7        if $s \notin \mathcal{S}$                          // is the found score new?
8             $\mathcal{S} = \mathcal{S} \cup \{s\}$
9  return $\mathcal{S}$
```

Since the scores of the vertices depend on $n(n-1)$ out-degrees, any algorithm determining the score set requires $\Omega(n^2)$ time. Due to the embedded loops in lines 02–08 the running time of SET1 is $\Omega(n^2)$ even in the best case. The precise order of the running time depends among others on the implementation of the **if** instruction in line 07. E.g., if line 07 is implemented by the comparison of the actual score with the elements of S, then the running time is $\Theta(n^3)$ for a score sequence containing different elements and is $\Theta(n^2)$ for a regular tournament.

Out-degree matrix $\mathcal{M}_{n\times n} = [m_{ij}]_{n\times n}$ is a useful tool in the implementation of graph algorithms. The input of the following algorithm QUICK-SET1 is n and $\mathcal{M}$, and the output is the score sequence **s** as a nonincreasingly ordered sequence and the score set $\mathcal{S}$ as an increasingly ordered sequence. QUICK-SET1 calls the well-known sorting procedure INSERTION-SORT.

Quick-Set1$(n, \mathcal{M})$

```
 1  $\mathcal{S} = \emptyset$
 2  for $i = 1$ to $n$
 3      $s_i = 0$
 4      for $j = 1$ to $n$
 5          $s_i = s_i + m_{ij}$                    // score sequence is computed
 6  $S_1 = s_1$
 7  Insertion-Sort($\mathbf{s}$) if $s \notin \mathcal{S}$   // sorting of the score vector
 8  for $i = 2$ to $n$
 9      if $s_i \neq s_{i-1}$
10         $S_k = s_i$
11         $k = k + 1$
12  return $\mathbf{s}$, $\mathcal{S}$
```

Since the embedded loops in lines 02–05 need $\Theta(n^2)$ time, and the remaining part of the code requires less, the running time of Quick-Set1 is $\Theta(n^2)$ in all cases.

30.1.2. Tournaments with prescribed score set

Constructing a tournament with a prescribed score set is more difficult than determining the score set. Quite surprisingly, if sufficiently many players participate in a tournament then any finite set of nonnegative integers can arise as a score set. This was conjectured by K. B. Reid in 1978 and turned out to be a relatively challenging problem.

Reid proved the result when $\mid \mathcal{S} \mid = 1, 2$ or 3, or if $\mathcal{S}$ contains consecutive terms of an arithmetic or geometric progression. That is, Reid showed that any set of one, two or three nonnegative integers is a score set of some tournament and additionally, any set of the form $\{s, s+d, s+2d, \ldots, s+pd\}$ for $s > 0, d > 1$ or $\{s, sd, sd^2, \ldots, sd^p\}$ for $s \geq 0, d > 0$, is a score set of some tournament. Hager settled the cases $|\mathcal{S}| = 4$ and $|\mathcal{S}| = 5$ in 1986 and finally in 1987, T. Yao gave an existence proof of the general Reid's conjecture based on arithmetic analysis.

Theorem 30.1 (Yao, 1988) *Every finite nonempty set $\mathcal{S}$ of nonnegative integers is the score set of some tournament.*

Let us try to formulate Reid's conjecture purely as a statement about numbers. Let $\mathcal{S} = \{s_1, \ldots, s_p\}$ be an increasing sequence of nonnegative integers. The conjecture means that there exist positive integers $x_1, \ldots, x_p$ such that

$$\mathbf{S} = (s_1^{x_1}, \ldots, s_2^{x_2} \ldots, s_p^{x_p})$$

is the score sequence of some 1-tournament with $\sum_{i=1}^{p} x_i = n$ vertices. By Landau's theorem, $\mathbf{a} = (a_1, \ldots, a_n)$, with $a_1 \leq \cdots \leq a_n$, is the score sequence of some 1-tournament T_n if and only if $\sum_{i=1}^{k} a_i \geq \binom{k}{2}$, for $k = 1, \ldots, n-1$ and $\sum_{i=1}^{n} a_i = \binom{n}{2}$. Thus it can be readily seen that Reid's conjecture is equivalent to the following statement.

For every nonempty set of nonnegative integers $S = \{s_1, \ldots, s_p\}$, where $s_1 <$

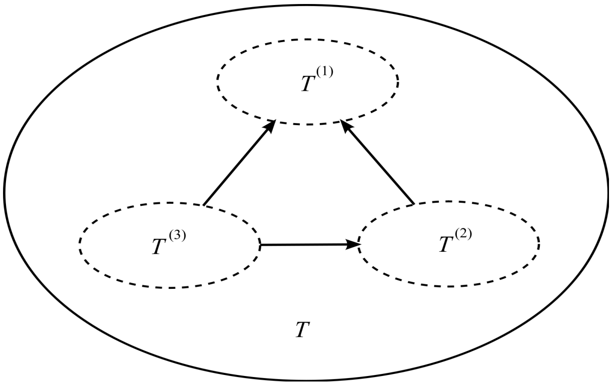


Figure 30.4 Construction of tournament T with odd number of distinct scores.

$\cdots < s_p$, there exist positive integers $x_1, \ldots, x_p$, such that

$$\sum_{i=1}^{k} s_i x_i \geq \binom{\sum_{i=1}^{k} x_i}{2}, \quad \text{for } k = 1, \ldots, p-1\,, \tag{30.1}$$

$$\sum_{i=1}^{p} s_i x_i = \binom{\sum_{i=1}^{p} x_i}{2}\,. \tag{30.2}$$

It is this equivalent formulation of Reid's conjecture that led to Yao's proof. The proof is not combinatorial in nature, but uses first of all some results of number theory. Commenting on Yao's proof Qiao Li wrote in 2006 in the *Annals of New York Academy of Sciences:*

Yao's proof is the first proof of the conjecture, but I do not think it is the last one. I hope a shorter and simpler new proof will be coming in the near future.

However, the prophecized constructive proof has not been discovered yet. This is in sharp contrast with Landau's theorem on score sequences, for which several proofs have emerged over the years. Recently, S. Pirzada and T. A. Naikoo gave a constructive combinatorial proof of a new special case of Reid's theorem. Their proof gives an algorithm for constructing a tournament with the prescribed score set, provided the score increments are increasing.

Theorem 30.2 (Pirzada and Naikoo, 2008) *If $a_1, a_2, \ldots, a_p$ are nonnegative integers with $a_1 < a_2 < \cdots < a_p$, then there exists a 1-tournament T with score set*

$$S = \left\{ s_1 = a_1, s_2 = \sum_{i=1}^{2} a_i, \ldots, s_p = \sum_{i=1}^{p} a_i \right\}. \tag{30.3}$$

Since any set of nonnegative integers can be written in the form of 30.3, the above theorem is applicable to all sets of nonnegative integers $S = \{s_1, s_2, \ldots, s_p\}$

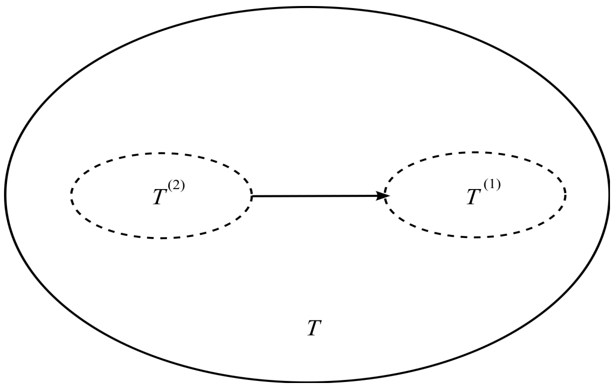


Figure 30.5 Construction of tournament T with even number of distinct scores.

with increasing increments (i.e., $s_1 < s_2 - s_1 < s_3 - s_2 < \cdots < s_p - s_{p-1}$.) The importance of Pirzada-Naikoo proof of Theorem 30.2 is augmented by the fact that Yao's original proof is not constructive and is not accessible to a broad audience[1].

The following recursive algorithm is based on Pirzada and Naikoo's proof of Theorem 30.2. The algorithm takes the set of increments $I_p = \{a_1 < a_2 < \cdots < a_p\}$ of the score set S as input and returns a tournament T whose score set is S. Let $X_t = \{a_1 < a_2 < \cdots < a_t\}$ for $1 \le t \le p$. Let R_n denote the regular tournament on n vertices and let $T^{(1)} \oplus T^{(2)}$ denote the vertex and arc disjoint union of tournaments $T^{(1)}$ and $T^{(2)}$.

```
Score-Reconstruction1($p, I_p$)

1  if   $p$ is odd
2          print Odd($p, I_p$)
3  else print Even($p, I_p$)
```

This algorithm calls one of the two following recursive procedures ODD and Even according to the parity of p. The input of both algorithm is some prefix X_t of the sequence of the increments $a_1, a_2, \ldots, a_t$, and the output is a tournament having the score set corresponding to the given increments.

[1]Yao's proof originally appeared in Chinese in the journal *Kexue Tongbao*. Later in 1989, the proof was published in English in the *Chinese Science Bulletin*. Unfortunately neither are accessible through the world wide web, although the English version is available to subscribers of the *Chinese Science Bulletin*. In Hungary this journal is accessible in the Library of Technical and Economical University of Budapest.

ODD(t, X_t)

```
1  if    $t == 1$
2        return $R_{2a_1+1}$
3  else $T_t^{(3)} = R_{(2(a_t-a_{t-1}+a_{t-2}-a_{t-3}+\cdots+a_3-a_2+a_1)+1)}$
4        $T_t^{(2)} = R_{2(a_{t-1}-a_{t-2}+a_{t-3}-a_{t-2}+\cdots+a_4-a_3+a_2-a_1-1)+1}$
5        $t = t - 2$
6        $T_t^{(1)} = $ ODD$(t, X_t)$
7        $T_t = T_t^{(3)} \oplus T_t^{(2)} \oplus T_t^{(1)}$
8        $T_t = T+$ arcs such that
9                 $T_t^{(2)}$ dominates $T_t^{(1)}$
10                $T_t^{(3)}$ dominates $T_t^{(1)}$
11                $T_t^{(3)}$ dominates $T_t^{(2)}$
12 return $T_t$
```

We can remark that the tournament constructed by the first execution of line 03 of ODD contains the vertices whose score is a_p, while the tournament constructed in line 04 contains the vertices whose score is a_{p-1} in the tournament appearing as output. The vertices having smaller scores appear during the later execution of lines 03 and 04 with exception of the vertices having score a_1 since those vertices will be added to the output in line 02.

EVEN(t, X_t)

```
1     $T_t^{(2)} = R_{2(a_t-a_{t-1}+a_{t-2}-a_{t-3}+\cdots+a_4-a_3+a_2-a_1-1)+1}$
2     $t = t - 1$
3     $T_t^{(1)} = $ ODD$(t, X_t)$
4     $T_t = T_t^{(2)} \oplus T_t^{(1)}$
5     $T_t = T+$ arcs such that $T_t^{(2)}$ dominates $T_t^{(1)}$
6  return $T_t$
```

Since the algorithm is complicated, let's consider an example.

Example 30.1 Let $p = 5$ and $I_5 = \{0, 1, 2, 3, 4\}$. Since p is odd, SCORE-RECONSTRUCTION1 calls ODD in line 02 with parameters 5 and I_5.

The first step of ODD is the construction of $T_5^{(3)} = T_{2(4-3+2-1+0)+1} = T_5$ in line 03. Denoting the vertices of this regular 5-tournament by v_1, v_2, v_3, v_4, v_5 and using the result of Exercise 30.1-1 we get the out-degree matrix shown in Figure 30.6.

The second step of ODD is the construction of $T_5^{(2)} = T_{2(3-2+1-0-1)+1} = T_3$. Let v_6, v_7 and v_8 be the vertices of this tournament.

The third step of ODD is the recursive call with parameters $p = 3$ and $X_3 = \{2,\ 1,\ 0\}$. The fourth action of ODD is the construction of $T_3^{(3)} = T_{2(2-1+0)+1} = T_3$. Let v_9, v_{10} and v_{11} be the vertices of this tournament. The fifth step is the construction of $T_3^{(2)} = T_{2(2-1+0-1)+1} = T_1$. Let v_{12} be the only vertex of this graph. The sixth action is the call of ODD with parameters $t = 1$ and $X_1 = \{0\}$. Now the number of increments equals to 1, therefore the algorithm constructs $T_1^{(1)} = T_1$ in line 02.

The seventh step is the construction of T in line 07, then the eighth step is adding new

vertex/vertex	v_1	v_2	v_3	v_4	v_5	Score
v_1	—	1	1	0	0	2
v_2	0	—	1	1	0	2
v_3	0	0	—	1	1	2
v_4	1	0	0	—	1	2
v_5	1	1	0	—	0	2

Figure 30.6 Out-degree matrix of the tournament $T_5^{(3)}$.

vertex/vertex	v_9	v_{10}	v_{11}	v_{12}	v_{13}	Score
v_9	—	1	0	1	1	3
v_{10}	0	—	1	1	1	3
v_{11}	1	0	—	1	1	3
v_{12}	0	0	0	—	1	1
v_{13}	0	0	0	0	—	0

Figure 30.7 Out-degree matrix of the tournament $T_5^{(3)}$.

v/v	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	Score
v_1	—	0	0	0	0	0	0	0	0	0	0	0	0	0
v_2	1	—	0	0	0	0	0	0	0	0	0	0	0	1
v_3	1	1	—	1	0	0	0	0	0	0	0	0	0	3
v_4	1	1	0	—	1	0	0	0	0	0	0	0	0	3
v_5	1	1	1	0	—	0	0	0	0	0	0	0	0	3
v_6	1	1	1	1	1	—	1	0	0	0	0	0	0	6
v_7	1	1	1	1	1	0	—	1	0	0	0	0	0	6
v_8	1	1	1	1	1	1	0	—	0	0	0	0	0	6
v_9	1	1	1	1	1	1	1	1	—	1	0	1	1	10
v_{10}	1	1	1	1	1	1	1	1	0	—	1	1	0	10
v_{11}	1	1	1	1	1	1	1	1	0	0	—	1	1	10
v_{12}	1	1	1	1	1	1	1	1	1	0	0	—	1	10
v_{13}	1	1	1	1	1	1	1	1	1	1	0	0	—	10

Figure 30.8 Out-degree matrix of the tournament T_5.

arcs (according to lines 08–11) to the actual T constructed in line 07 and consisting from 3 regular tournaments having altogether 5 vertices (v_{13}, v_{12}, v_{11}, v_{10}, v_9). The result is shown in Figure 30.7.

Ninth step of Odd is joining the tournaments T_5 and T_3 to T and the final step is adding of the domination arcs. The out-degree matrix of the output T_5 of Odd is shown in Figure 30.8.

Correctness of the algorithm Let $I = \{a_1, a_2, \ldots, a_p\}$ be a set of p nonnegative integers with $a_1 < a_2 < \cdots < a_p$. SCORE-RECONSTRUCTION1 performs two types of recursions: first if p is odd and the second if p is even. Assume p to be odd. For $p = 1$, the set I contains one nonnegative integer a_1 and the algorithm returns the regular tournament T_{2a_1+1} as output. Note that each vertex of T_{2a_1+1} has score $\left(\frac{2a_1+1-1}{2}\right) = a_1$, so that score set of T_{2a_1+1} is $S = \{s_1 = a_1\}$. This shows that the algorithm is correct for $p = 1$.

If $p = 3$, then the set of increments I consists of three nonnegative integers $\{a_1, a_2, a_3\}$ with $a_1 < a_2 < a_3$. Now $a_3 > a_2$, therefore $a_3 - a_2 > 0$, so that $a_3 - a_2 + a_1 > 0$ as $a_1 \geq 0$. Let $T^{(3)}$ be a regular tournament having $2(a_3 - a_2 + a_1) + 1$ vertices. Then each vertex of $T^{(3)}$ has score $\left(\frac{2(a_3-a_2+a_1)+1-1}{2}\right) = a_3 - a_2 + a_1$.

Again, since $a_2 > a_1$, therefore $a_2 - a_1 > 0$, so that $a_2 - a_1 - 1 \geq 0$. Let $T^{(2)}$ be a regular tournament having $2(a_2 - a_1 - 1) + 1$ vertices. Then each vertex of $T^{(2)}$ has score $\left(\frac{2(a_2-a_1-1)+1-1}{2}\right) = a_2 - a_1 - 1$. Also since $a_1 \geq 0$, let $T^{(1)}$ be a regular tournament having $2a_1+1$ vertices. Then each vertex of T_1 has score $\left(\frac{2a_1+1-1}{2}\right) = a_1$.

If $p = 3$, SCORE-RECONSTRUCTION1 outputs a tournament T whose vertex set is the disjoint union of vertex sets of $T^{(1)}$, $T^{(2)}$ and $T^{(3)}$ and whose arc set contains all the arcs of $T^{(1)}$, $T^{(2)}$ and $T^{(3)}$ such that every vertex of $T^{(2)}$ dominates each vertex of $T^{(1)}$, and every vertex of $T^{(3)}$ dominates each vertex of $T^{(1)}$ and $T^{(2)}$. Thus T has $2a_1 + 1 + 2(a_2 - a_1 - 1) + 1 + 2(a_3 - a_2 + a_1) + 1 = 2(a_1 + a_3) + 1$ vertices with score set

$$
\begin{aligned}
S &= \{a_1, a_2 - a_1 - 1 + 2a_1 + 1, a_3 - a_2 + a_1 + 2(a_2 - a_1 - 1) + 1 + 2a_1 + 1\} \\
&= \left\{a_1, \sum_{i=1}^{2} a_i, \sum_{i=1}^{3} a_i\right\} .
\end{aligned}
$$

This shows that the algorithm is correct for $p = 3$ too. When the set I of increments consists of an odd number of nonnegative integers, the algorithm recursively builds the required tournament by using the procedure ODD. To see this, assume that the algorithm works for all odd numbers upto p. That is, if $a_1, a_2, \ldots, a_p$ are p nonnegative integers with $a_1 < a_2 < \cdots < a_p$, then the algorithm outputs a tournament having $2(a_1+a_3+\ldots+a_p)+1$ vertices with score set $\{a_1, \sum_{i=1}^{2} a_i, \ldots, \sum_{i=1}^{p} a_i\}$. Let us call this tournament $T^{(1)}$.

We now show how the algorithm constructs a tournament with $p+2$ vertices with score set $\{a_1, \sum_{i=1}^{2} a_i, \ldots, \sum_{i=1}^{p+2} a_i\}$, where $a_1, a_2, \ldots, a_{p+2}$ are $p + 2$ nonnegative integers with $a_1 < a_2 < \cdots < a_{p+2}$.

Since $a_2 > a_1, a_4 > a_3, \ldots, a_{p-1} > a_{p-2}, a_{p+1} > a_p$. therefore $a_2 - a_1 > 0$, $a_4 - a_3 > 0, \ldots, a_{p-1} - a_{p-2} > 0$, $a_{p+1} - a_p > 0$, so that $a_{p+1} - a_p + a_{p-1} - a_{p-2} + \ldots + a_4 - a_3 + a_2 - a_1 > 0$, that is, $a_{p+1} - a_p + a_p - 1 - a_{p-2} + \ldots + a_4 - a_3 + a_2 - a_1 - 1 \geq 0$.

The procedure ODD constructs $T^{(2)}$ as a regular tournament having $2(a_{p+1} - a_p + a_{p-1} - a_{p-2} + \cdots + a_4 - a_3 + a_2 - a_1 - 1) + 1$ vertices. Each vertex of $T^{(2)}$ has score

$$
\begin{aligned}
&\frac{2(a_{p+1} - a_p + a_{p-1} - a_{p-2} + \ldots + a_4 - a_3 + a_2 - a_1 - 1) + 1 - 1}{2} \\
&= a_{p+1} - a_p + a_{p-1} - a_{p-2} + \cdots + a_4 - a_3 + a_2 - a_1 - 1 .
\end{aligned}
$$

Again, $a_3 > a_2, \dots, a_p > a_{p-1}, a_{p+2} > a_{p+1}$, therefore $a_3 - a_2 > 0, \dots, a_p - a_{p-1} > 0, a_{p+2} - a_{p+1} > 0$, so that $a_{p+2} - a_{p+1} + a_p - a_{p-1} + \cdots + a_3 - a_2 + a_1 > 0$ as $a_1 \geq 0$.

The procedure Odd constructs $T^{(3)}$ as a regular tournament having $2(a_{p+2} - a_{p+1} + a_p - a_{p-1} + \cdots + a_3 - a_2 + a_1) + 1$ vertices. Each vertex of $T^{(3)}$ has score

$$\begin{aligned}&\frac{2(a_{p+2} - a_{p+1} + a_p - a_{p-1} + \cdots + a_3 - a_2 + a_1) + 1 - 1}{2}\\ &= a_{p+2} - a_{p+1} + a_p - a_{p-1} + \cdots + a_3 - a_2 + a_1 \,.\end{aligned}$$

Now Score-Reconstruction1 sets $T = T^{(1)} \oplus T^{(2)} \oplus T^{(3)}$ and adds additional arcs in such a way that every vertex of $T^{(2)}$ dominates each vertex of $T^{(1)}$, and every vertex of $T^{(3)}$ dominates each vertex of $T^{(1)}$ and $T^{(2)}$. Therefore T is a tournament having

$$\begin{aligned}&2(a_1 + a_3 + \cdots + a_p) + 1 + 2(a_{p+1}a_p + a_{p1}a_{p2} + \cdots + a_4a_3 + a_2a_1) + 1\\ &\qquad\qquad +2(a_{p+2}a_{p+1} + a_pa_{p-1} + \cdots + a_3a_2 + a_1) + 1\\ &= 2(a_1 + a_3 + \cdots + a_{p+2}) + 1\end{aligned}$$

vertices with score set

$$S = \left\{ a_1, \sum_{i=1}^{2} a_i, \dots, \sum_{i=1}^{p} a_i, \sum_{i=1}^{p+1} a_i, \sum_{i=1}^{p+2} a_i \right\} .$$

Hence by induction, the algorithm is correct for all odd p.

To prove the correctness for even case, note that if p is odd, then $p+1$ is even. Let $a_1, a_2, \dots, a_{p+1}$ be $p+1$ nonnegative integers with $a_1 < a_2 < \cdots < a_{p+1}$.. Therefore $a_1 < a_2 < \cdots < a_p$, where p is odd. The procedure Even uses the procedure Odd to generate a tournament $T^{(1)}$ having $2(a_1 + a_3 + \cdots + a_p) + 1$ vertices with score set $S = \{a_1, \sum_{i=1}^{2} a_i, \dots, \ \sum_{i=1}^{p} a_i\}$.

Also, since $a_2 > a_1$, $a_4 > a_3, \dots, a_{p-1} > a_{p-2}, a_{p+1} > a_p$, the procedure Even generates a regular tournament $T^{(2)}$ having $2(a_{p+1} - a_p + a_{p-1} - a_{p-2} + \cdots + a_4 - a_3 + a_2 - a_1 - 1) + 1$ vertices such that the score for each vertex is $a_{p+1} - a_p + a_{p-1} - a_{p-2} + \cdots + a_4 - a_3 + a_2 - a_1 - 1$.

Finally the algorithm generates the tournament $T^{(1)} \oplus T^{(2)}$ and adds additional arcs so that every vertex of $T^{(2)}$ dominates each vertex of $T^{(1)}$. The resulting tournament T consists of

$$\begin{aligned}&2(a_1 + a_3 + \cdots + a_{p-2} + a_p) + 1\\ &\qquad +2(a_{p+1} - a_p + a_{p-1} - a_{p-2} + \cdots + a_4 - a_3 + a_2 - a_1 - 1) + 1\\ &= 2(a_2 + a_4 + \cdots + a_{p+1})\end{aligned}$$

vertices and has score set

$$\begin{aligned} S = \{a_1, \sum_{i=1}^{2} a_i, \ldots, \sum_{i=1}^{p} a_i, \\ & a_{p+1} - a_p + a_{p-1} - a_{p-2} + \cdots + a_4 - a_3 + a_2 - a_1 - 1 \\ & \qquad +2(a_1 + a_3 + \cdots + a_{p-2} + a_p) + 1\} \\ = \{a_1, \sum_{i=1}^{2} a_i, \ldots, \sum_{i=1}^{p+1} a_i\} . \end{aligned}$$

This shows that the algorithm is correct for even p as well.

Computational complexity The running time of Score-Reconstruction1 depends on the size of the score set $|S|$ as well as the largest increment $a_p = s_p - s_{p-1}$. The details are left as a problem for the Reader (see Exercise 30.1-1).

Exercises

30.1-1 The out-degree matrix $\mathcal{M}$ of a tournament is defined as a $0-1$ matrix with (i, j) entry equal to 1 if player v_i defeats player v_j and 0 otherwise (see (30.13)). A tournament is completely determined by its out-degree matrix. Write an $O(n^2)$ algorithm to generate the out-degree matrix of a regular tournament on n vertices, where n is any odd positive integer. *Hint.* Circularly place $\binom{n-1}{2}$ ones in each row.

30.1-2 Use Exercise 30.1-1 and the discussion in this section to determine the worst-case running time of Score-Reconstruction1.

30.1-3 Obtain the out-degree matrix of a tournament with score set $\{1, 3, 6\}$. How many vertices does this tournament have? Draw this tournament and give its outdegree-matrix.

30.1-4 Use the tournament obtained in Exercise 30.1-3 to generate the out-degree matrix of a 1-tournament with score set $\{1, 3, 6, 10\}$. Write the score sequence of your tournament.

30.2. Score sets in oriented graphs

Oriented graphs are generalizations of tournaments. Formally, an ***oriented graph*** $D(V, A)$ with vertex set V and arc set A is a digraph with no symmetric pairs of directed arcs and without loops. In other words oriented graph is a directed graph in which every pair of different vertices is connected with at most one arc, or oriented graphs are $(0, 1)$-*tournaments.*

Figure 30.9 shows an oriented graph with score sequence $[1, 3, 3, 5]$ and the coressponding score set $\{1, 3, 5\}$.

Thus tournaments are complete oriented graphs, in the sense that any pair of vertices in a tournament is joined exactly by one arc. Several concepts defined for tournaments can be extended in a meaningful way to oriented graphs. For example score of a player (vertex) in a tournament is defined as its out-degree, as a player

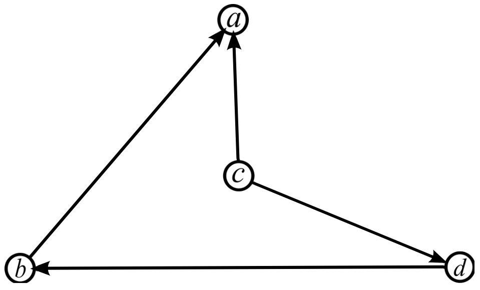


Figure 30.9 An oriented graph with score sequence $[1, 3, 3, 5]$ and score set $\{1, 3, 5\}$.

either wins (and earns one point) or looses (earning no points) a two-way clash. In 1991, Peter Avery introduced the score structure for oriented graphs based on the intuition that in a round-robin competition with ties allowed, a player may earn two, one or no points in case the player wins, looses or makes a tie respectively.

More precisely, the score of a vertex v_i in a k-tournament D with n vertices is defined as

$$a(v_i) = a_i = n - 1 + d^+_{v_i} - d^-_{v_i} ,$$

where $d^+_{v_i}$ and $d^-_{v_i}$ are the out-degree and in-degree, respectively, of v_i. The score sequence of an oriented graph is formed by listing the vertex scores in non-decreasing order. If we denote the number of non-arcs in D containing the vertex v_i as $d^*_{v_i}$, then

$$a_i = 2d^+_{v_i} + d^*_{v_i} .$$

With this score structure, an oriented graph can be interpreted as the result of a round-robin competition in which ties (draws) are allowed, that is, the players play each other once, with an arc from player u to v if and only if u defeats v. A player receives two points for each win, and one point for each tie.

It is worth to remark that this is a sophisticated score structure comparing with the simple and natural structure of 2-tournaments.

Avery gave a complete characterization of score sequences of oriented graphs similar to Landau's theorem.

Theorem 30.3 (Avery, 1991) *A nondecreasing sequence $A = [a_1, \ldots, a_n]$ of non-negative integers is the score sequence of an oriented graph if and only if*

$$\sum_{i=1}^{k} a_i \geq k(k-1) \tag{30.4}$$

for $1 \leq k \leq n$ with equality when $k = n$.

Proof This theorem is a special case of the theorem proved by Moon in 1963 or the theorem proved by Kemnitz and Dulff in 1997 (see the theorem and its proof in Chapter 25, that is chapter *Comparison Based Ranking*). ■

Just as in the case of 1-tournaments, the ***score set of an oriented graph*** is defined as the set of scores of its vertices. It is worth noting that a $(0, 1)$-tournament has different score sets under Avery's and Landau's score structures. In fact, the score of a vertex v under Avery's score structure is twice the score of v under Landau's score structure. This is obviously due to Avery's assumption that a win contributes 2 points to the score.

The score set of an oriented graph can be determined by adapting Quick-Set2 as follows:

Quick-Set2$(n, \mathcal{M})$

```
 1  $S = \emptyset$
 2  for $i = 1$ to $n$
 3       $s_i = 0$
 4       for $j = 1$ to $n$
 5            $s_i = s_i + 2m_{ij}$
 6            if $m_{ij==0}$ and $m_{ji} == 0$
 7                 $s_i = s_i + 1$                  // score sequence is computed
 8  $S_1 = s_1$
 9  $k = 2$
10  for $i = 2$ to $n$
11       if $s_i \neq s_{i-1}$                       // is the found score new?
12            $S_k = s_i$
13            $k = k + 1$
14  return s, $S$
```

The running time of Quick-Set2 is $\Theta(n^2)$ since the nested loop in lines 02–07 requires $\Theta(n^2)$ the remaining lines require $\Theta(n)$ time.

30.2.1. Oriented graphs with prescribed scoresets

In Section **??** we discussed score sets of tournaments and noted that every non-empty set of nonnegative integers is the score set of some tournament. In this section we study the corresponding question for oriented graphs, i.e., which sets of nonnegative integers can arise as score sets of oriented graphs. Pirzada and Naikoo investigated this question and gave two sufficient conditions for a set of nonnegative integers to be the score set of some oriented graph.

Theorem 30.4 (Pirzada, Naikoo, 2008) *Let a, d, n nonnegative integers, and $S = \{a, ad, ad^2, \ldots, ad^n\}$, with $d > 2$ or $d = 2$ and $n > 1$. Then there exists an oriented graph with score set A except for $a = 1, d = 2, n > 0$ and for $a = 1, d = 3, n > 0$.*

Theorem 30.5 (Pirzada, Naikoo, 2008) *If n is a positive integer and $a_1, a_2, \ldots, a_n$ are nonnegative integers with $a_1 < a_2 < \cdots < a_n$, then there exists an oriented graph with $a_n + 1$ vertices and with score set $S = \{a'_1, a'_2, \ldots, a'_n\}$, where*

$$a'_i = \begin{cases} a_{i-1} + a_i + 1 & \text{for } i > 1\,, \\ a_i & \text{for } i = 1\,. \end{cases} \tag{30.5}$$

Thus any set of positive integers whose elements form a geometric progression is the score set of some oriented graph with few exceptions and any set of nonnegative integers whose elements are of the form (30.5) is also a score set. It follows that every singleton set of nonnegative integers is the score set of some oriented graph. On the other hand, for any positive integer n, the sets $\{1, 2, 2^2, \dots, 2^n\}$ and $\{1, 3, 3^2, \dots, 3^n\}$ cannot be the score sets of an oriented graph. Therefore, unlike in the case of tournaments, not all sets of nonnegative integers are score sets of oriented graphs. So far no complete characterization of score sets of oriented graphs is known.

The proof of Theorem 30.4 depends on the following auxiliary assertion.

Lemma 30.6 Naikoo, Pirzada, 2008) *The number of vertices in an oriented graph with at least two distinct scores does not exceed its largest score.*

Proof This assertion is the special case $k = 2$ of Lemma **??** due to Iványi and Phong. ■

Here we omit formal proofs of Theorems 30.4 and 30.5 since they can be found on the internet and since we will implicitly prove these theorems when we check the correctness of Geometric-Construction and Adding-Construction, respectively.

We first present a recursive algorithm that takes positive integers a, d, and n, satisfying the condition of Theorem **??**, as input and generates a 2-tournament $D(V, A)$ with score set $\{a, ad, ad^2, \dots, ad^n\}$. Let N_p denote the null digraph on p vertices, i.e., the digraph with n vertices and no arcs.

```
Geometric-Construction($a, d, n$)

 1  if    $a = 0$ or $n = 0$
 2        $D = N_{a+1}$
 3        return $D$
 4  else
 5        $D^{(1)} =$ Geometric-Construction($a, d, n - 1$)
 6        $U =$ vertex set of $D^{(1)}$
 7        $D = D^{(1)} \oplus N_{ad^n - 2|U| + 1}$
 8        Add arcs to $D$ such that
 9        $N_{ad^n - 2|U| + 1}$ dominates $D^{(1)}$
10        return $D$
```

```
Geometric-Construction($a, d, n$)
1  if $n = 0$
2      $D = N_{a+1}$
3      return $D$
4  if $n = 1$
4  if $n \geq 2$
5      $n = n - 1$
6      $D^(1) =$ Geometric(a,d,n)
6          $U =$ vertex set $D^(1)$
7      $D = D^(1) \oplus N$
```

Example 30.2 Let $a = 2,\ d = 2$ and $n = 2$. Then the prescribed score set is $\{2, 4, 8\}$. The first step is the call of Geometric with parameters $(2, 2, 2)$.

Algorithm description If $n = 0$, then the algorithm returns the null digraph N_{a+1}. Note that N_{a+1} is well-defined as $a + 1 > 0$. Each vertex of N_{a+1} has score $a+1-1+0-0 = a$. Therefore the score set of N_{a+1} is $S = \{a\}$. Thus the algorithm is correct for $n = 0$.

Now we prove the correctness of Geometric by induction. That is, we show that if the algorithm is valid for $n = 0, 1, \ldots, p$ for some integer $p \geq 1$ then it is also valid for $n = p + 1$. Let a and d be positive integers with $a > 0$ and $d > 1$ such that for $a = 1$, $d \neq 2, 3$. By the induction hypothesis the algorithm can construct an oriented graph $D^{(1)}$ with score set $\{a, ad, \ldots, ad^p\}$ and $a, ad, \ldots, ad^p$ are the distinct scores of the vertices of $D^{(1)}$. Let U be the vertex set of $D^{(1)}$.

There are three possibilities:

- $a = 1$ and $d > 3$,
- $a > 1$ and $d = 2$ or
- $a > 1$ and $d > 2$.

Obviously, for $d > 1$ in all the above cases we have $ad^{p+1} \geq 2ad^p$. Also the score set of $D^{(1)}$, namely $\{a, ad, \ldots, ad^p\}$, has at least two distinct scores for $p \geq 1$. Therefore, by Lemma 30.6 we have $|U| \leq ad^p$. Hence $ad^{p+1} \geq 2|U|$ so that $ad^{p+1} - 2|U| + 1 > 0$.

Let $N_{ad^{p+1}-2|U|+1}$ be the null digraph with vertex set X.. The algorithm now generates the vertex and arc disjoint union $D = D^{(1)} \oplus N_{ad^{p+1}-2|U|+1}$ and adds an arc directed from each vertex in $N_{ad^{p+1}-2|U\ vert+1}$ to every vertex of $D^{(1)}$. The output $D(V, A)$ of Geometric-Seq-Construction, therefore, has $|V| = |U| + ad^{p+1} - 2|U| + 1 = ad^{p+1} - |U| + 1$ vertices. Moreover, $a + |X| - |X| = a$,, $ad + |X| - |X| = ad$. $ad^2 + |X| - |X| = ad^2, \ldots, ad^p + |X| - |X| = ad^p$ are the distinct scores of the vertices in U, while $a_x = |U| - 1 + |V| - 0 = ad^{p+1} - |V| + 1 - 1 + |V| = ad^{p+1}$ for all vertices $x \in X$.

Therefore the score set of D is $S = \{a, ad, ad^2, \ldots, ad^p, ad^{p+1}\}$ which shows that the algorithm works for $n = p + 1$. Hence the algorithm is valid for all a, d and n satisfying the hypothesis of Theorem **??**.

The recursive procedure GEOMETRIC runs n times and during its i^{th} run the procedure adds $O(ad^{n+1-i})$ arcs to the oriented graph D. The overall complexity of the algorithm is therefore $O(nad^n)$.

As noted in Theorem **??**, there exists no 1-tournament when either $a = 1,\ d = 2,\ n > 0$ or $a = 1,\ d = 3,\ n > 0$. It is quite interesting to investigate these exceptional cases as it provides more insight into the problem.

Let us assume that $\mathbf{S} = \{1, 2, 2^2, \ldots, 2^n\}$ is a score set of some oriented graph D for $n > 0$. Then there exist positive integers, say $x_1, x_2, x_3, \ldots, x_{n+1}$ such that

$$S_1 = [1^{x_1}, 2^{x_2}, \ldots, (2^2)^{x_3}, \ldots, (2^n)^{x_{n+1}}$$

is the score sequence of D. Therefore, by relations (30.4) of score sequences of 1-tournaments, we have

$$x_1 + 2x_2 + 2^2x_3 + \cdots + 2^nx_{n+1} = \left(\sum_{i=1}^{n+1} x_i\right)\left(\sum_{i=1}^{n+1} x_i - 1\right),$$

which implies that x_1 is even. However, x_1 is a positive integer, therefore $x_1 \geq 2$. Let the scores be $a_1 = 1$, $a_2 = 1$ and $a_3 \geq 1$. By inequalities (30.4) $a_1 + a_2 + a_3 \geq 3(3-1) = 6$, or in other words, $a_3 \geq 4$. This implies that $x_2 = 0$, a contradiction.

The proof of the other exceptional case ($\mathbf{S} = \{1, 3, 3^2, \ldots, 3^n\}$) is left as an exercise (Exercise 30.2-1).

The next algorithm takes the set $I = \{a_1 < a_2 < \cdots < a_n\}$ consisting of n nonnegative integers as input and recursively constructs a 2-tournament $D(V, A)$ the score set $S = \{a'_1, a'_2, \ldots, a'_n\}$ where a'_i are of the form 30.5.

ADDING-CONSTRUCTION(n, I_n)

```
1  if $n = 0$
2      $D = N_{a_1+1}$
3      return $D$
4  $n = n - 1$
5  $D^{(1)} =$ ADDING-CONSTRUCTION$(n, I_n)$
6  $D = D^1 \oplus N_{a_{n+1}-a_n}$
7          Add arcs to $D$ such that
8          $N_n$ dominates $D^(1)$
9  return $D$
```

```
Adding-Construction($n, I_n$)
1  if $n = 0$
2      $D = N_{a_1+1}$
3  else
4      $D^{(1)}$ = Adding-Construction($n-1, I_n - \{a_n\}$)
5      $D = D^{(1)} \oplus N_{a_n - a_{n-1}}$
7      Add arcs to $D$ such that
8      $N_{a_n - a_{n-1}}$ dominates $D^{(1)}$
9  return $D$
```

Algorithm description If $n = 1$, the algorithm returns the null digraph N_{a_1+1}. Each vertex of N_{a_1+1} has the score $a_1 + 1 - 1 + 0 - 0 = a_1 = a'_1$. Therefore the score set of N_{a_1+1} is $S = \{a'_1\}$ as required.

We prove the correctness of General-Construction in general by induction on n. Assume that the algorithm is valid for $n = 1, 2, \ldots, p$, for some integer $p \geq 2$. We show that the algorithm is also valid for $n = p + 1$. Let $a_1, a_2, \ldots, a_{p+1}$ be nonnegative integers with $a_1 < a_2 < \cdots < a_{p+1}$. Since $a_1 < a_2 < \cdots < a_p$, by the induction hypothesis, the algorithm returns an oriented graph $D^{(1)}$ on $a_p + 1$ vertices with score set $\{a'_1, a'_2, \ldots, a'_p\}$, where a'_i is given by equations (30.5). That is, score set of $D^{(1)}$ is $\{a_1, a_1 + a_2 + 1, a_2 + a_3 + 1, \ldots, a_{p-1} + a_p + 1\}$. So a_1, $a_1 + a_2 + 1$, $a_2 + a_3 + 1, \ldots, a_{p-1} + a_p + 1$ are the distinct scores of the vertices of D. Let X be the vertex set of $D^{(1)}$ so that $|X| = a_p + 1$. Since $a_{p+1} > a_p$, $a_{p+1} - a_p > 0$, the algorithm constructs a new oriented graph $D = D^{(1)} \oplus N_{p+1}$ with vertex set $V = X \cup Y$, where Y is the vertex set of N_{p+1} and $|Y| = a_{p+1} - a_p$. Arcs are added to D such that there is an arc directed from each vertex in Y to every vertex in X. Thus D has $|V| = |X| + |Y| = a_p + 1 + a_{p+1} - a_p = a_{p+1} + 1$ vertices. The distinct score of vertices in X are $a_1 + |Y| - |Y| = a_1 = a'_1, a_1 + a_2 + 1 + |Y| - |Y| = a_1 + a_2 + 1 = a'_2, a_2 + a_3 + 1 + |Y| - |Y| = a_2 + a_3 + 1 = a'_3, \ldots, a_{p-1} + ap + 1 + |Y| - |Y| = a_{p-1} + a_p + 1 = a'_p$, while $a_y = |X| - 1 + |V| - 0 = a_{p+1} + 1 - 1 + a_p + 1 = a_p + a_{p+1} + 1 = a'_{p+1}$ for all $y \in Y$.

Therefore the score set of D is $S = \{a'_1, a'_2, \ldots, a'_p, a'_{p+1}\}$ which proves the validity of algorithm for $n = p + 1$. Hence by induction, General-Construction is valid for all n.

The analysis of computational complexity of General-Construction is left as an exercise (Exercise 30.2-2).

Exercises

30.2-1 Prove that there exists no oriented graph with score set $\{1, 3, 3^2, \ldots, 3^n\}$ for any $n > 0$.

30.2-2 Adding-Construction is a recursive algorithm. Analyse its running time and compare its performance with the performance of Geometric-Construction.

30.2-3 Implement Adding-Construction in a suitable programming language and use it to construct an oriented graph with score set $\{2, 4, 8\}$. Write the score sequence of your oriented graph.

30.2-4 Implement Adding-Construction in a suitable programming language and use it to construct an oriented graph with score set $\{1, 4, 6, 9\}$. Write the score sequence of your oriented graph.

30.2-5 Give a proof of Lemma 30.6.

30.2-6 For any nonnegative integer n, what is the score set of the regular tournament T_{2n+1} when considered as an oriented graph.

30.2-7 Determine the score set of the oriented graph $D = T_3 \oplus T_5$, where T_5 dominates T_3, i.e., there is an arc directed from every vertex of T_5 to every vertex of T_3.

30.2-8 Write an $O(n)$ algorithm to determine the score set of directed cycles (i.e., cycles with directed edges). How can we make this algorithm work for directed wheels (note that a *wheel* is a cycle with an additional vertex joined to all the vertices on the cycle).

30.3. Unicity of score sets

k-tournaments (multitournaments) are directed graphs in which each pair of vertices is connected with exactly k arcs.

Reid formulated the following conjecture in [132].

Conjecture 30.7 *Any set of nonnegative integers is the score set of some 1-tournament T.*

Using Landau's theorem this conjecture can be formulated in the following arithmetic form too.

Conjecture 30.8 *If $0 \le r_1 < r_2 < \cdots < r_m$, then there exist such positive integers $x_1, x_2, \ldots, x_m$, that*

$$\sum_{i=1}^{j} x_i r_i \ge \frac{(\sum_{i=1}^{j} x_i)(\sum_{i=1}^{j} x_i - 1)}{2}, \ j \in [1 : m]$$

and

$$\sum_{i=1}^{m} x_i r_i = \frac{(\sum_{i=1}^{m} x_i)(\sum_{i=1}^{m} x_i - 1)}{2} .$$

In this case we say that the sequence $\mathbf{s} = \langle s_1, \ldots, s_n\rangle = \langle r_1^{x_1}, \ldots, r_m^{x_m}\rangle$ realizes the sequence $\mathbf{r} = \langle r_1, \ldots, r_m\rangle$ or $\mathbf{s}$ is a solution for $\mathbf{r}$.

Reid gave a constructive proof of his conjecture for sets containing one, two or three elements [132].

Later Hager published a constructive proof for sets with four and five elements [58] and Yao [154] published the outline of a nonconstructive proof of the general case.

A score set is called ***k-unique,*** if there exists exactly 1 score sequence of k-tournaments generating the given set. In the talk we investigate the following questions:

1. characterization of the unique score sets of 1-tournaments;
2. extension of the Reid's conjecture to 2-tournaments.

30.3.1. 1-unique score sets

At first we formulate a useful necessary condition.

Lemma 30.9 *(Iványi and Phong, 2004) If $k \geq 1$, then for any (n, k)-tournament holds that the sequence $\mathbf{s}$ is a solution for $\mathbf{r}$, then in the case $m = 1$ we have*

$$n = 2r_1 + 1 \tag{30.6}$$

and in the case $m \geq 2$ we have

$$\frac{2r_1}{k} + 1 < n < \frac{2r_m}{k} + 1 \tag{30.7}$$

and

$$n \geq r_m + 1\,. \tag{30.8}$$

Proof If ■

This lemma implies the exact answer for the case $m = 1$.

Corollary 30.10 *(Iványi and Phong, 2004) If $\mathbf{r} = \langle r_1 \rangle$, then exactly the sequence $\mathbf{s} = \langle r_1^{2r_1+1} \rangle$ is a solution for $\mathbf{r}$.*

Proof Lemma **??** implies that only this solution is acceptable. One can check that it satisfies the required inequality and equality. ■

Now we present a useful method of the investigation of the uniqueness. Let $\mathbf{r} = \langle a, a + d \rangle$. Then according to the Reid-equality we get

$$2ax + 2(a + d)y = n(n - 1)$$

implying

$$y = \frac{n(n - 2a - 1)}{2d}\,. \tag{30.9}$$

But here only the values $n = 2a + 1 + i$ $(i \in [1, 2d - 1])$ are permitted where

$$i \geq d + 1 - a\,. \tag{30.10}$$

By substitution $a = (q - 1)d$ from (30.9) we get

$$y = \frac{(2qd - 2d + 2r + 1 + i)i}{2d}\,. \tag{30.11}$$

Here y must be an integer, so transform this formula into

$$y = i(q - d) + \frac{i(2r + 1 + i)}{2d}\,. \tag{30.12}$$

Theorem 30.11 *If $0 \le a < b$, then there exist positive integers x and y satisfying*

$$ax \ge \frac{x(x-1)}{2}$$

and

$$ax + by = \frac{(x+y)(x+y-1)}{2} .$$

In the following cases there is only one solution:

- *$a = 0$;*
- *$d = 1$;*
- *$d = 2$.*

In the following case there are at least two solutions:

- *d is odd and $3 \le d \le a$.*

Proof a) Existence of a solution. Let $d = b - a$ and $i = 2d - 2r - 1$. Then $n = 2(b - r)$, $y = q(2d - 2r - 1)$, $x = q(2r + 1)$ satisfy all requirements.

b) Uniqueness. If $a = 0$, then $d = b$, $q = 1$ and y is integer only if $i = 2b - 1$. So we get the unique $\langle 0^1, b^{2b-1} \rangle$ solution.

If $d = 1$, then only $i = 1$ is permitted, implying the unique solution $\langle a^b, b^b \rangle$.

If $d = 2$ or d is odd, then we also can analyse formula (30.12). ■

This theorem left open the case when the difference d is odd and the investigated set is sparse and also the case when the difference is an even number greater then 2.

30.3.2. 2-unique score sets

Now we present a new form of Reid-problem for 2-tournaments.

For a fixed sequence $\mathbf{q}[m] = \langle q_1, \ldots, q_m \rangle$ with $q_1 < \cdots < q_m$ of positive integers, we shall denote by $\mathcal{G}(\mathbf{q}[m])$ the set $\mathcal{G}$ of sequences $\mathbf{g} = \langle g_1, \ldots, g_m \rangle$ such that

$$\sum_{i=1}^{k} q_i g_i \ \ge \ \left(\sum_{i=1}^{k} g_i \right)^2 , \qquad k \in [1 : m-1]$$

and

$$\sum_{i=1}^{m} q_i g_i \ = \ \left(\sum_{i=1}^{m} g_i \right)^2 .$$

Here we also say that $\mathbf{g}$ is a solution for $\mathbf{q}$.

We wish to give necessary and sufficient conditions for $\mathbf{q}[m]$ to have a solution that is a nonempty $\mathcal{G}(\mathbf{q}[m])$.)

Theorem 30.12 *For the sequence $\mathbf{q}[1] = \langle q_1 \rangle$, we have $\mathcal{G}(\mathbf{q}[1]) = \langle q_1 \rangle$.*

Proof If $\mathbf{q}[1] = \langle q_1 \rangle$, then it is obvious that the solution of $q_1 g_1 = g_1^2$ is given in the form $g_1 = q_1$. Hence we have $\mathcal{G}(\mathbf{q}[1]) = \langle q_1 \rangle$. ■

Theorem 30.13 *Let $\mathbf{q}[2] = \langle q_1, q_2\rangle$ be a sequence of positive integers with $d = q_2 - q_1 > 0$. Then $\mathcal{G}(\mathbf{q}[2]) \neq \emptyset$ if and only if either $d \not| (q_1, q_2)$ or $d | (q_1, q_2)$ and there is a prime p such that $p^2 | d$.*

Proof According to the definition of $\mathcal{G}(\mathbf{q}[m])$, we need only find positive integers g_1, g_2 such that $q_1 \geq g_1$ and $q_1 g_1 + q_2 g_2 = (g_1 + g_2)^2$.

Let q, r be integers for which $q_2 = qd + r$, where $0 \leq r < d$. If $d \not|(q_1, q_2)$, then $r \neq 0$ and let $g_1 = rq$ and $g_2 = q_2 - r(q+1)$. Hence we have

$$g_1 = rq = r\frac{q_2 - r}{q_2 - q_1} = r + r\frac{q_1 - r}{q_2 - q_1}$$

$$< r + (R_1 - r) = R_1\,,$$

$$g_2 = R_2 - r(q+1) =$$

$$q_2 - (q_2 - r)\frac{r}{q_2 - q_1} - r$$

$$> q_2 - (q_2 - r) - r = 0$$

and

$$q_1 g_1 + q_2 g_2 = q_1 r q + q_2^2 - q_2 r(q+1)$$
$$= q_2^2 + r(q_1 q - q_2 q + q_2) - 2q_2 r =$$
$$= (q_2 - r)^2 = (g_1 + g_2)^2.$$

Now assume that $d | (q_1, q_2)$ and there is a prime p such that $p^2 | d$. In this case $r = 0$ and we choose g_1, g_2 as follows:

$$g_1 := \frac{q_2}{p} - \frac{d}{p^2} \quad \text{and} \quad g_2 := g_1(p-1)\,.$$

It is obvious that

$$g_1 > 0, \;\; g_2 > 0\;, \;\; g_1 \leq R_1$$

and

$$q_1 g_1 + q_2 g_2 = g_1(q_1 + (p-1)q_2)$$
$$= g_1(pq_2 - d) =$$
$$= g_1 p^2 (\frac{q_2}{p} - \frac{d}{p^2}) = (g_1 p)^2 = (g_1 + g_2)^2.$$

Finally, assume that $d = 1$ or $d | (q_1, q_2)$ and d is the product of distinct primes. If there are positive integers g_1, g_2 such that $q_1 \geq g_1$ and $q_1 g_1 + R_2 g_2 = (g_1 + g_2)^2$, then we have $d | g_1 + g_2$ and

$$\frac{1}{d}(g_1 + g_2)^2 - \frac{q_1}{d}(g_1 + g_2) = g_2 > 0\,,$$

$$\frac{1}{d}(g_1 + g_2)^2 - \frac{R_2}{d}(g_1 + g_2) = -g_1 < 0\,,$$

consequently

$$\frac{q_2}{d} = \frac{q_1}{d} + 1 > \frac{g_1 + g_2}{d} > \frac{q_1}{d} \, .$$

This is impossible. ■

Theorem 30.14 Iványi, Phong, 2004 *Let* $\mathbf{q}[2] =< q_1, \ q_2 >$ *be the sequence of positive integers with conditions* $q_1 < R_2$, $(q_1, q_2) = 1$, $2q_1 > q_2$ *and* $d := q_2 - R_1$ *has* s *distinct prime factors. Then*

$$|\mathcal{G}(\mathbf{q}[2])| = 2^s - 1 \, .$$

Proof Since $d = q_2 - q_1 < q_1$ and $(q_1, q_2) = 1$, the congruence $x^2 \equiv q_2 x \pmod{d}$ has $2^s - 1$ solutions in positive integers less than d. For each solution x we set $g_1 = \frac{x(q_2 - x)}{d}$ and $g_2 = (d - x)\frac{q_2 - x}{d}$. One can check that $g_1, \ g_2$ satisfy conditions $q_1 \geq g_1$ and $q_1 g_1 + q_2 g_2 = (g_1 + g_2)^2$. ■

Exercises

30.3-1 How many ?

30.3-2 Design an algorithm

30.4. Kings and serfs in tournaments

Sociologists are often interested in determining the most *dominant* actors in a social network. Moreover, dominance in animal societies is an important theme in ecology and population biology. Social networks are generally modelled as digraphs with vertices representing actors and arcs representing dominance relations among actors. The concept of "king" is very closely related to dominance in digraphs. Kings and serfs were initially introduced to study dominance in round-robin competitions. These concepts were latter extended to more general families of digraphs such as multipartite tournaments, quasi-transitive digraphs, semicomplete multipartite digraphs and oriented graphs. In this section our focus will be on algorithmic aspects of kings and serfs in tournaments and their applications in majority preference voting.

A king in a tournament dominates every other vertex either directly or through another vertex. To make the idea more formal we define a *path* of length k from a vertex u to a vertex v in a tournament (or any digraph) as a sequence of arcs $e_1, e_2, \ldots, e_k$ where u is the initial vertex of e_1, v is the terminal vertex of e_k and the terminal vertex of e_i is the same as the initial vertex of e_{i+1}, for all $1 \leq i \leq k-1$. If there is a path of length 1 or 2 from a vertex u to a vertex v, then v is said to be ***reachable*** from u within two steps. Analogously, if there is a path of length $1, 2, \ldots$ or r from u to v then v is said to be reachable from u within r steps. Let T be an n-tournament. A vertex u in T is called an ***r-king,*** where $1 \leq r \leq n-1$, if every other vertex v in the tournament is reachable within r steps from u. A vertex u is called an ***r-serf*** if u is reachable within r if u is reachable within r steps from every

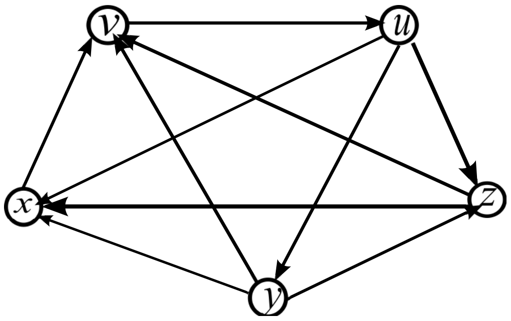


Figure 30.10 A tournament with three kings $\{u, v, y\}$ and three serfs $\{u, v, x\}$. Note that z is neither a king nor a serf and $\{u.v\}$ are both kings and serfs.

other vertex v in T. In particular, a 2-king is simply called a king and a 2-serf is called a serf.

S. B. Maurer introduced the dual terms of king and serf in a delightful exposition of a tournament model for dominance in flocks of chicken. In his influential series of papers on dominance in animal societies, H. G. Landau proved that every tournament has a king (although he did not use the word king). In fact, he showed the following.

Theorem 30.15 (Landau, 1953) *Every vertex of maximum score in a tournament is a king.*

The proof is quite intuitive. Suppose to the contrary that u is a vertex with maximum score in a tournament T and u is not a king. Then there exists another vertex v in T such that v is not reachable from u within 2 steps. But this means that u and all out-neighbours of u are reachable from v in 1 step and so $s(v) > s(u)$, a contradiction. Another classical result by J. W. Moon states that

Theorem 30.16 (Moon, 1968) *A tournament without transmitters (vertices with in-degree 0) contains at least three kings.*

It is natural to ask if the bound on the number of kings given in Theorem 30.16 is tight. The answer is yes, as demonstrated by the following example.

Example 30.3 Let T be a tournament with vertex set $\{v_1, v_2, \ldots, v_5\}$. Let us denote by (u, v), an arc directed from u to v. Suppose that the arc set of T consists of the arcs (v_3, v_5), (v_4, v_3), all arcs of the form (v_{j-1}, v_j), with $1 < j \le 5$ and all arcs of the form $(v_{j+2}, v_j), (v_{j+3}, v_j), \ldots, (v_n, v_j)$ with $j = 1, 2, 4$. Then it can be easily verified (Exercise 30.4-2) that T has no transmitters and v_2, v_3 and v_4 are the only kings in T.

K. B. Reid proved the existence of a tournament with an arbitrary number of vertices and an arbitrary number of kings, with few exceptions.

Theorem 30.17 (Reid, 1982) *For all integers $n \ge k \ge 1$ there exists a tournament on n vertices with exactly k kings except when $k = 2$ or when $n = k = 4$ (in which case no such n-tournament exists).*

Hence no tournament has exactly two kings. The above theorems can be stated just as well in terms of serfs. To see this, note that the *converse* T' of a tournament T, obtained by reversing the arcs of T, is also a tournament and that the kings and serfs of T and T' are interchanged.

The ***king set*** of a tournament consists of all kings in the tournament. We can define the ***serf set*** analogously. The problem of determining the king set of a tournament is very important both for theoretical and practical considerations. In voting theory literature, political scientists often refer to the uncovered set in majority preference voting. This uncovered set is actually the king set for the tournament whose vertices consist of the candidates to be elected and arcs represent the outcomes of the two-way race between candidates. Here we present a simple polynomial time algorithm for determining the king set of a tournament. Given an n-tournament T, let us define an $n \times n$ matrix D_T^+ as

$$(D_T^+)_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an arc of } T\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{30.13}$$

We call D_T^+, the *out-degree matrix* of T. When there is no danger of ambiguity we will drop the subscript T and simply denote the out-degree matrix by D^+. King-Set takes a tournament $T(V, A)$ as input, calculates the out-degree matrix D^+ of T and uses it to generate the king set K of T. Let O be the $n \times n$ zero matrix and let I be the $n \times n$ identity matrix.

```
King-Set(V, A)
1  $D^+ =$
2  $K = \emptyset$
3  for $i = 1$ to $n$
4      for $j = 1$ to $n$
5          if $(v_i, v_j) \in A$
6              $(D^+)_{ij} = 1$
7  $M = I + D^+ + (D^+)^2$
8  $K = \{v_i \in V | \forall v_j \in V, (M)_{ij} \neq 0\}$
9      $N_n$ dominates $D^(1)$
9  return $K$
```

Algorithm description The algorithm works on the same principle as the algorithm for finding the number of paths, from one vertex to another, in a digraph (Exercise 30.4-1 asks you to derive this algorithm). The (i, j) entry of the matrix $(D^+)^2$ is equal to the number of paths of length two from vertex v_i to vertex v_j (check this!). Therefore, the (i, j) entry of matrix $D^+ + (D^+)^2$ counts the number of paths of length one or two from v_i to v_j; and if vertex v_i is a king, all entries in the i^{th} row of $I + D^+ + (D^+)^2$ must be non-zero.

The computational complexity of Algorithm King-Set depends on the way $(D_T^+)^2$ is computed. If naive matrix multiplication is used, the algorithm runs in $\Theta(n^3)$ time. However, using the fast matrix multiplication by Coppersmith and Winograd, the running time can be reduced to $O(n^{2.38})$. The Reader should note that by using the duality of kings and serfs, King-Set can be adapted for finding the serf set of a tournament.

King sets in majority preference voting Kings frequently arise in political science literature. A ***majority preference voting*** procedure asks each voter to rank candidates in order of preference. The results can be modeled by a tournament where vertices represent the candidates and arcs point toward the loser of each two way race, where candidate u defeats candidate v if some majority of voters prefer u to v. Political scientists are often interested in determining uncovered vertices in the resulting tournament. A vertex u is said to *cover* another vertex v if u defeats v and also defeats every vertex that v defeats.

The covering relation is clearly transitive and has maximal elements, called ***uncovered vertices.*** An uncovered vertex u has the strategically important property that u defeats any other vertex v in no more than two steps, i.e., either

1. u defeats v or
2. there is some third alternative w such that u defeats w and w defeats v.

Thus an uncovered vertex is actually a king. In fact the uncovered set, consisting of all uncovered vertices, is precisely the set of all kings (see Exercise 30.4-8).

The idea behind finding kings in a tournament can be easily extended to finding r-kings for any positive integer r.

```
rKing-Set(V, A, r)
1  $D^+ = 0$
2  $K = \emptyset$
3  for $i = 1$ to $n$
4      for $j = 1$ to $n$
5          if $(v_i, v_j) \in A$
6              $(D^+)_{ij} = 1$
7  $M = I + D^+ + \ldots + (D^+)^r$
8  $K = \{v_i \in V | \forall v_j \in V, (M)_{ij} \neq 0\}$
9  return $K$
```

The above algorithm runs in $O(rn^3)$ if the matrix multiplications are performed naively, and in $O(rn^{2.38})$ time if fast matrix multiplication is incorporated.

As we have seen, kings dominate in tournaments. However, there exists a stronger notion of dominance in tournaments in the form of strong kings. Let us write $u \to v$ to denote that u defeats v in a tournament T, or in other words (u, v) is an arc of T. If U_1 and U_2 are disjoint subsets of vertices of T then we write $U_1 \to U_2$ to denote that all vertices in U_1 defeat all vertices in U_2. We define $B_T(u, v) = \{w \in V - \{u, v\} : u \to w \text{ and } w \to v\}$, where V denotes the vertex set of T. Let $b_T(u, v) = |B_T(u, v)|$. When no ambiguity arises, we drop the subscript T from the notation.

A vertex u in a tournament T is said to be a ***strong king*** if $u \to v$ or $b(u, v) > b(v, u)$ for every other vertex v of T.

Note that $b_T(u,v)$ is the number of paths of length two through which v is reachable from u. Therefore, $b_T(v_i,v_j) = ((D_T^+)^2)_{ij}$, where D_T^+ is the out-degree matrix of T.

Obviously, it is not true that every king is a strong king. For example, Figure 30.11 demonstrates a tournament with three kings, namely x, y and z. However, only x and y are strong kings as $b(z,x) < b(x,z)$. Figure 30.11 also shows that when searching for the most dominant vertex in real life applications, a king may not be the best choice (vertex z is a king, but it defeats only one vertex and is defeated by all other vertices). Therefore, choosing a strong king is a better option. This intuition is further confirmed by the fact that, in the probabilistic sense it can be shown that in almost all tournaments every vertex is a king.

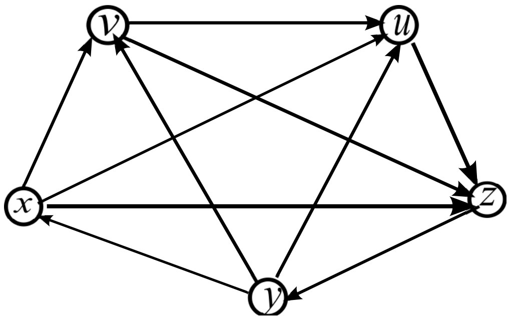


Figure 30.11 A tournament with three kings and two strong kings

We have already shown that every tournament has a king. We now prove that every tournament has a strong king.

Theorem 30.18 *(???, ????) Every vertex with maximum score in a tournament is a strong king.*

Proof Suppose u is a vertex with maximum score in a tournament T that is not a strong king. Then there is a vertex v in T such that $v \to u$ and $b(u,v) \le b(v,u)$. Let V be the vertex set of T. Define

$$W = \{w \in V - \{u,v\} : u \to w \text{ and } v \to w\}.$$

Then $s(u) = b(u,v) + |W|$ and $s(v) = b(v,u) + |W| + 1$. This implies that $s(u) < s(v)$, a contradiction. ■

The problem of finding strong kings is no harder than finding kings in tournaments. Like King-Set, we present a polynomial time algorithm for finding all strong kings in a tournament using the out-degree matrix D^+.

Strong-Kings(V, A)

```
1  $D^+ = 0$
2  $K = \emptyset$
3  for $i = 1$ to $n$
4      for $j = 1$ to $n$
5          if $(v_i, v_j) \in A$
6              $D^+_{ij} = 1$
7  $M = D^+ + (D^+)^2$
8  $K = \{v_i \in V \mid \forall j (1 \le j \le n \text{ and } j \ne i), M_{ij} > M_{ji}\}$
9  return $K$
```

Strong-Kings has the same order of running time King-Set.

So far we have been focusing on finding certain type of dominant vertices (like kings and strong kings) in a tournament. Another very important problem is to construct tournaments with a certain number of dominant vertices. Maurer posed the problem of determining all 4-tuples (n, k, s, b) for which there exists a tournament on n vertices with exactly k kings and s serfs such that b of the kings are also serfs. Such a tournament is called an (n, k, s, b)-tournament. For example the tournament given in Figure **??** is a $(5, 3, 3, 2)$-tournament. Reid gave the following characterization of such 4-tuples.

Theorem 30.19 *Suppose that $n \ge k \ge s \ge b \ge 0$ and $n > 0$. There exists an (n, k, s, b)-tournament if and only if the following conditions hold.*

1. *$n \ge k + s - b$,*
2. *$s \ne 2$ and $k \ne 2$,*
3. *either $n = k = s = b \ne 4$ or $n > k$ and $s > b$,*
4. *(n, k, s, b) is none of $(n, 4, 3, 2)$, $(5, 4, 1, 0)$, or $(7, 6, 3, 2)$.*

However, the corresponding problem for strong kings has been considered only recently. For $1 \le k \le n$, a tournament on n vertices is called an (n, k)-tournament if it has exactly k strong kings. The construction of (n, k)- tournaments follows from the results proved by Chen, Chang, Cheng and Wang in 2004. The results imply the existence of (n, k)-tournaments for all $1 \le k \le n$ satisfying

$$k \ne n - 1, \text{ when } n \text{ is odd} \tag{30.14}$$

$$k \ne n, \qquad \text{when } n \text{ is even}\,. \tag{30.15}$$

Algorithm nk-Tournament takes positive integers n and k as input satisfying the constraints (26.2) and (26.3) and outputs an (n, k)-tournament and the set K of its strong kings. Also for any vertex u of a tournament T, we adopt the notation of Chen et al. in letting $O(u)$ (respectively, $I(u)$) denote the set of vertices reachable from u in one step (respectively, set of vertices from which u is reachable in one step). Note that $O(u)$ and $I(u)$ are often referred to as the first out-neighbourhood and first in-neighbourhood of u respectively.

nk–Tournament(n, k)

```
$K = \emptyset$
$T$ = null digraph on $n$ verices
if $k$ is odd
   $T = T_k$
   $K = \{v_1, \ldots, v_k\}$
if $n \neq k$
   for $i = k + 1$ to $n$
        $V = V \cup \{v_i\}$
        $A = A \cup \{(u, v_i) : u \in V - \{v_i\}\}$
if $k$ is even
   $T = T_{k-1}$
   $V = V \cup \{x, y, z\}$
   $K = \{v_1, \ldots, v_{k-3}, x\}$
   choose $u \in V$ arbitrarily
   $A = A \cup \{(v, x) : v \in O(u)\}$
   $A = A \cup \{(x, v) : v \in \{u, y\} \cup I(u)\}$
   $A = A \cup \{(v, y) : v \in \{u\} \cup I(u) \cup O(u)\}$
   $A = A \cup \{(v, z) : v \in \{u\} \cup I(u)\}$
   $A = A \cup \{(z, v) : v \in O(u)\}$
   if $n \neq k + 2$
      for $i = k + 1$ to $n$
           $V = V \cup \{v_i\}$
           $A = A \cup \{(u, v_i) : u \in V - \{v_i\}\}$
return $T, K$
```

Algorithm description The algorithm consists of performing two separate inductions to generate an (n, k)-tournament, one for odd k and one for even k.. If k is odd then we start by letting $T = T_k$, the regular tournament on k vertices (which always exists for odd k), and inductively add $n - k$ vertices to T that are defeated by all the vertices of T_k. Thus the resulting tournament has n vertices and k kings (the vertices of T_k). The construction for even k is a bit more involved. We start with $T = T_{k-1}$. Note that every vertex of T_{k-1} has score $m = \binom{n-4}{2}$. We then add three vertices x, y and z and several arcs to T_{k-1} such that for a fixed existing vertex u of T_{k-1}.

- $O(u) \to \{x\} \to \{u, y\} \cup I(u)$,
- $\{u\} \cup I(u) \cup O(u) \to \{y\} \to \{x, z\}$,
- $\{u\} \cup I(u) \to \{z\} \to O(u)$.

The resulting tournament T (illustrated in Figure 30.12) has $k + 2$ vertices with scores $s(x) = |I(x)| + 2 = m + 2$, $s(y) = 2$, $s(z) = |O(x)| = m$ and $s(v) = m + 2$,, for all vertices v of T_{k-1}. Now by Theorem 30.18 all vertices v of T_{k-1} and the new

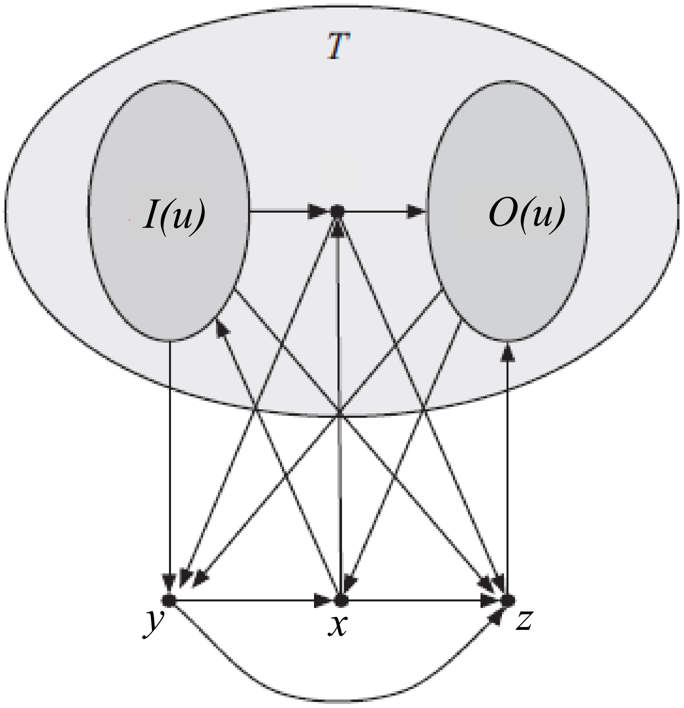


Figure 30.12 Construction of an (n, k)-tournament with even k.

vertex x are strong kings of T, while y and z are not (Exercise 30.4-9). Thus T is a $(k+2, k)$-tournament that can now be extended to an (n, k)-tournament by adding $n-k-2$ more vertices that are defeated by all the existing vertices of T (just like in the case of odd k).

nk-Tournament runs in quadratic time as it takes $O(n^2)$ operations to construct a regular tournament and the remaining steps in the algorithm are completed in linear time.

Exercises

30.4-1 The out-degree matrix D^+ of an n-vertex oriented graph is an $n \times n$ matrix whose (i, j) entry is given by d_{ij} = number of arcs directed from v_i to v_j. Describe an algorithm based on the out-degree matrix for finding the number of paths of length $k < n$ between any two vertices of the graph.

30.4-2 Draw the tournament discussed in Example 30.3 and show that it has no transmitters and exactly three kings.

30.4-3 Using the 5-tournament in Example 30.3 give the construction of an n-tournament with no transmitters and exactly three kings.

30.4-4 For every odd number $n \geq 3$, give an example of an n-tournament, in which all vertices are serfs.

30.4-5 Prove that any tournament on 4 vertices contains a vertex which is not a king.

30.4-6 A *bipartite tournament* is an orientation of a complete bipartite graph. A vertex v of a bipartite tournament is called a 4-king[2] (or simply a king) if there is a directed path of length 4 from v to every other vertex of the tournament. Derive an algorithm to obtain all 4-kings in a bipartite tournament and compare its complexity with the complexity of r-Kings for finding r-kings in ordinary tournaments.

30.4-7 As the name suggests a multipartite tournament is an orientation of a complete multipartite graph. Extend the algorithm obtained in Exercise 30.4-6 to find all 4-kings in multipartite tournaments. Again compare the performance of your algorithms with r-Kings.

30.4-8 Prove that the uncovered set arising in majority preference voting is exactly the king set of the majority preference tournament.

30.4-9 Show that when k is even, the output of nk-Tournament has exactly k kings.

30.5. Weak kings in oriented graphs

In the previous section we studied dominance in tournaments and used the terms kings and strong kings to describe the dominant vertices in a tournament. However, in most practical applications the underlying digraph is not a tournament. Rather we are interested in determining dominant vertices in an oriented graph. For instance, in a social network, an arc (u, v) denotes that actor u has some relation with actor v.. Since most social relations (such as hierarchy relations) are irreflexive and asymmetric, a majority of social networks can be modelled as oriented graphs. Therefore, we would like to generalize the concept of dominance from tournaments to oriented graphs. In Section **??**, we have already defined kings and r-kings in the context of general digraphs. The same definitions are applicable to oriented graphs.

As stated in the beginning of the chapter, oriented graphs can be considered as round-robin competitions in which ties are allowed. Thus the the classical notion of king, that is a vertex that defeats every other vertex either directly or through another vertex, is too strong for oriented graphs. To overcome this difficulty, the study of the so-called "*weak kings*" was initiated in 2008 by S. Pirzada and N. A. Shah. Here we follow their notation. For any two vertices u and v in an oriented graph D,, one of the following possibilities exist.

1. An arc directed from u to v, denoted by $u(1-0)v$ (i.e., u defeats v).
2. An arc directed from v to u , denoted by $u(0-1)v$ (i.e., v defeats u).
3. There is no arc from u to v or from v to u , and is denoted by $u(0-0)v$ (i.e., there is a tie).

A ***triple*** in an oriented graph is an induced oriented subgraph with three vertices.

[2]Several bipartite and multipartite tournaments have no 2-king or 3-king. However, a multipartite tournament with at least one vertex of in-degree zero contains a 4-king. Therefore it is logical to look for 4-kings in a multipartite tournament.

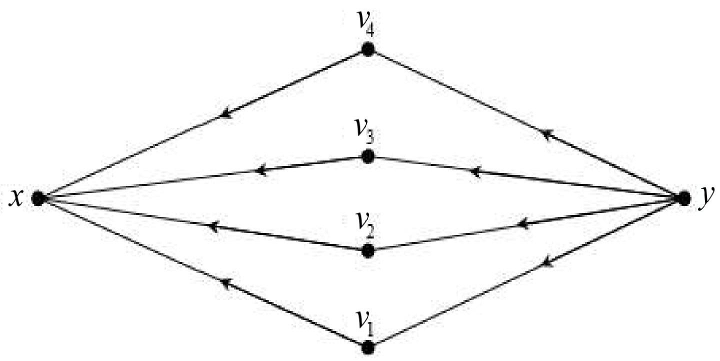


Figure 30.13 Six vertices and six weak kings.

For any three vertices u, v and w, the triples of the form $u(1-0)v(1-0)w(1-0)u$, $u(1-0)v(1-0)w(0-0)u$, $u(0-0)v(1-0)w(1-0)u$ or $u(1-0)v(0-0)w(1-0)u$ are said to be ***intransitive,*** while the triples of the form $u(1-0)v(1-0)w(0-1)u$, $u(0-1)v(1-0)w(1-0)u$, $u(1-0)v(0-1)w(1-0)u$, $u(1-0)v(0-1)w(0-0)u$, $u(0-1)v(0-0)w(1-0)u$, $u(0-0)v(1-0)w(0-1)u$, $u(1-0)v(0-0)w(0-1)u$, $u(0-0)v(0-1)w(1-0)u$, $u(0-1)v(1-0)w(0-0)u$, $u(1-0)v(0-0)w(0-0)u$, $u(0-1)v(0-0)w(0-0)u$, $u(0-0)v(1-0)w(0-0)u$, $u(0-0)v(0-1)w(0-0)u$, $u(0-0)v(0-0)w(1-0)u$ or $u(0-0)v(0-0)w(0-1)u$ are said to be ***transitive.*** An oriented graph is said to be ***transitive*** if all its triples are transitive. The converse $\overline{D}$ of an oriented graph D is obtained by reversing each arc of D.

Let u and v be vertices in an oriented graph D such that either $u(1-0)v$ or $u(0-0)v$ or $u(1-0)w(1-0)v$ or $u(1-0)w(0-0)v$ or $u(0-0)w(1-0)v$ for some vertex w in D. Then v is said to be weakly reachable within two steps from u. If either $u(1-0)v$, or $u(1-0)w(1-0)v$ for some w in D, then v is reachable within two steps from u.

A vertex u in an oriented graph D is called a ***weak king*** if every other vertex v in D is weakly reachable within two steps from u. A vertex u is called a ***king*** if every other vertex v in D is reachable within two steps from u. A vertex u in an oriented graph D is called a ***weak serf*** if u is weakly reachable within two steps from every other vertex in D, and a vertex u in D is called a ***serf*** if u is reachable within two steps from every other vertex v in D.

We note that there exist oriented graphs on n vertices with exactly k kings for all integers $n \geq k \geq 1$, with the exception of $n = k = 4$. Theorem 30.17 guarantees the existence of complete oriented graphs (tournaments) with n vertices and exactly k kings for all integers $n \geq k \geq 1$, with the exceptions $k = 2$ and $n = k = 4$. An oriented graph D with exactly two kings can be constructed as follows. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the vertex set of D, with arcs defined as $v_1(1-0)v_i$, for $i = 2, 4, \ldots, n$; $v_1(0-1)v_3$; $v_2(1-0)v_3$ and $v_2(1-0)v_i$, for $4 \leq i \leq n$; and for all other $i \neq j$, $v_i(0-0)v_j$. The vertices v_1 and v_3 are the only kings in D (Exercise

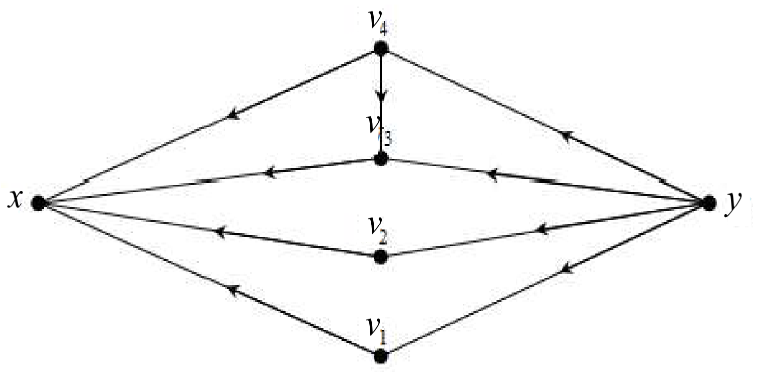


Figure 30.14 Six vertices and five weak kings.

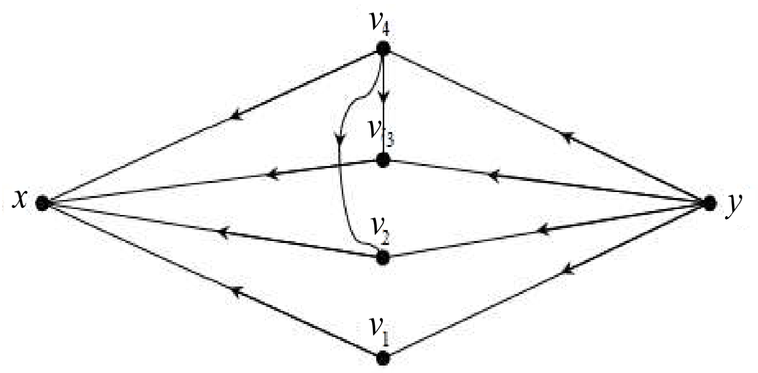


Figure 30.15 Six vertices and four weak kings.

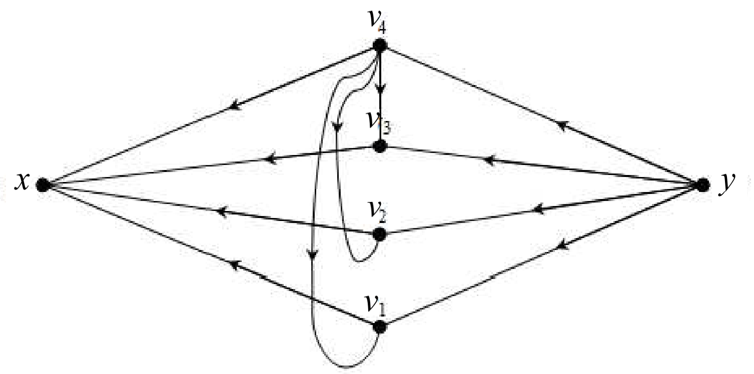


Figure 30.16 Six vertices and three weak kings.

30.5-1).

There do not exist any complete or incomplete oriented graphs with 4 vertices

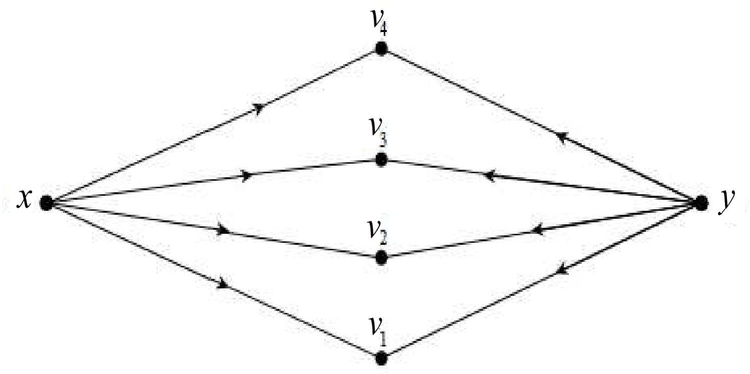


Figure 30.17 Six vertices and two weak kings.

and exactly 4 kings. Suppose to the contrary that this is the case and let D be the incomplete oriented graph with 4 vertices, all of whom are kings. Then D can be extended to a tournament on 4 vertices by inserting all the missing arcs with arbitrary orientation. Clearly such a tournament contains 4 kings, which contradicts Theorem 30.17.

The rest of the section is aimed at investigating weak kings in oriented graphs as they present a more suitable notion of dominance in oriented graphs. The score of a vertex in an oriented graph was defined in Section **??**. Considering Theorem 30.15, it is natural to ask if a vertex of maximum score in an oriented graph is a king. The answer is negative as shown by the following example:

Example 30.4 Consider the oriented graph D shown in Figure 30.18. The scores of vertices v_1, v_2, v_3 and v_4 are respectively 2, 3, 3 and 4. Clearly, v_4 is a vertex of maximum score but is not a king as v_1 is not reachable within two steps from v_4. However, v_4 is a weak king.

Now consider the oriented graph D^* with vertices u_1,, u_2, u_3, u_4 and u_5, and arcs defined by $u_1(1-0)u_2$, $u_2(1-0)u_i$, for $i = 3,\ 4, q5$ and $u_i(0-0)u_j$ for all other $i \neq j$. Clearly, $s(u_1) = 5$, $s(u_2) = 6$, $s(u_3) = 3$, $s(u_4) = 3$, and $s(u_5) = 3$. Evidently, u_1 is a king in D^* whereas the vertex u_2 of maximum score is not a king.

However, we do have the following weaker result.

Theorem 30.20 *If u is a vertex with maximum score in a 2-tournament D, then u is a weak king.*

Proof Let u be a vertex of maximum score in D, and let X, Y and Z be respectively the set of vertices x, y, and z such that $u(1-0)x$, $u(0-0)y$, and $u(0-1)z$. Let $|X| = n_1$, $|Y| = n_2$ and $|Z| = n_3$. Clearly, $s(u) = 2n_1 + n_2$. If $n_3 = 0$, the result is trivial. So assume that $n_3 \neq 0$. We claim that each $z \in Z$ is weakly reachable within two steps from u. If not, let z_0 be a vertex in Z not weakly reachable within two steps from u. Then for each $x \in X$ and each $y \in Y$, $z_0(1-0)x$, and $z_0(1-0)y$ or $z_0(0-0)y$. In case $z_0(1-0)x$ and $z_0(1-0)y$ for each $x \in X$ and each $y \in Y$, then $s(z_0) \geq 2 + 2n_1 + 2n_2 = s(u) + n_2 + 2 > s(u)$. which contradicts the choice of u. If

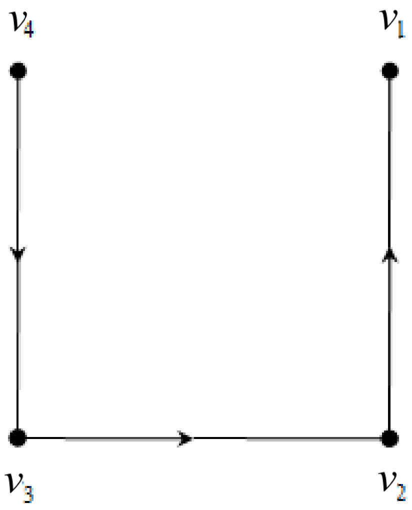


Figure 30.18 Vertex of maximum score is not a king.

$z_0(1-0)x$ and $z_0(0-0)y$ for each $x \in X$ and each $y \in Y$, then $s(z_0) \geq 2+2n_1+n_2 = s(u)+2 > s(u)$, again contradicting the choice of u. This establishes the claim, and hence the proof is complete. ■

We now consider the problem of finding all weak kings in an oriented graph (as kings can be determined by applying Algorithm **??**). Let D^- and D^+ respectively denote the in-degree and out-degree matrix of an oriented graph $D(V, A)$ with n vertices. Also let O and J denote the $n \times n$ zero matrix and all-ones matrix respectively.

```
Weak-Kings($V, A$)
 1  $D^+ = 0$
 2  $D^- = 0$
 3  $K = \emptyset$
 4  for $i = 1$ to $n$ and $j = 1$ to n
 5      for $j = 1$ to $n$
 6          if    $(v_i, v_j) \in A$
 7                $D^+_{ij} = 1$
 8          else if $(v_i, v_j) \in A$
 9                $D^-_{ij} = 1$
10  $M = J - D^-$
11  $M = D^+ + (D^+)^2$
12  $N = M + MD^+ + D^+M$
13  $K = \{v_i \in V \mid \forall v_j \in V, (N)_{ij} \neq 0\}$
14  return $K$
```

Algorithm **??** returns the set of all weak kings of an oriented graph. Exercise 30.5-3 asks you to prove that the algorithm works correctly and to determine its running time.

Indeed, it is also possible to extend Theorem 30.16 to weak kings in oriented graphs as an oriented graph D without transmitters (vertices of in-degree 0) has at least three weak kings. To see this let u be a vertex of maximum score in the oriented graph D. Clearly, by Theorem 30.20, u is a weak king. As D has no transmitters, there is at least one vertex v such that $v(1-0)u$. Let S be the set of these vertices v, and let v_1 be a vertex of maximum score in S. Let X, Y and Z respectively be the set of vertices x, y and z, other than u, with $v_1(1-0)x$, $v_1(0-0)y$ and $v_1(0-1)z$. Assume that $|X| = n_1$, $|Y| = n_2$, and $|Z| = n_3$ so that $s(v_1) = 2n_1 + n_2 + 2$. We note that all vertices of Z are weakly reachable within two steps from v_1. If this is not the case, let z_0 be a vertex which is not weakly reachable within two steps from v_1. Then $z_0(1-0)u$, and (a) $z_0(1-0)x$ and (b) $z_0(1-0)y$ or $z_0(0-0)y$ for each $x \in X$ and each $y \in Y$.

If for each x in X and each y in Y, $z_0(1-0)x$ and $z_0(1-0)y$, then $s(z_0) \geq 2n_1 + 2n_2 + 4 = s(v_1) + n_2 + 2 > s(v_1)$. This contradicts the choice of v_1. If for each x in X and each y in Y, $z_0(1-0)x$ and $z_0(0-0)y$, then $s(z_0) \geq 2n_1 + n_2 + 4 > s(v_1)$, again contradicting the choice of v_1. This establishes the claim, and thus v_1 is also a weak king.

Now let W be set of vertices w with $w(1-0)v_1$ and let w_1 be the vertex of maximum score in W. Then by the same argument as above, every other vertex in D is weakly reachable within two steps from w_1, and so w_1 is a weak king. Since D is asymmetric, and in D we have $w_1(1-0)v_1$ and $v_1(1-0)u$, therefore u, v_1 and w_1 are necessarily distinct vertices. Hence D contains at least three weak kings.

Although, no oriented graph with 4 vertices and exactly 4 kings exists, it is possible to generate an oriented graph on n vertices with exactly k weak kings, for all integers $n \geq k \geq 1$. The following algorithm constructs such an oriented graph.

```
kWeak-Kings($n, k$)
$V = \{x, y, u_1, u_2, \ldots, u_{n-2}\}$
$x(0-0)y$
if $k > 2$
   for $i = 1$ to $n-2$
        $u_i(1-0)x$
        $u_i(0-1)y$

   for $i = n-3$ downto $k-2$
        $u_{n-2}(1-0)u_i$
   for $i = k-3$ downto $1$
        $u_{n-2}(0-0)u_i$
   $K = \{x, y, u_{n-2}\} \cup \{u_i \mid i = 1, \ldots, k-3\}$
   else if    $k = 2$
               for $i = 1$ to $n-2$
                    $x(1-0)u_i$
                    $y(1-0)u_i$
               for $j = 1$ to $n-2$
                    if $i \neq j$
                       $u_i(0-0)u_j$
               $K = \{x, y\}$
         else $x(1-0)u_i$
               $u_1(1-0)y$
               for $i = 2$ to $n-2$
                    $u_1(1-0)u_i$
                    $x(1-0)u_i$
                    $y(1-0)u_i$
               $K = \{u_1\}$
return $V, A, K$
```

Algorithm description When $k = n$, the algorithm defines the arcs of a 2-tournament D with vertex set $V = \{x, y, u_1, u_2, \cdots, u_{n-2}\}$ as

$x(0-0)y$,

$u_i(1-0)x$ and $u_i(0-1)y$ for all $1 \leq i \leq n-2$,

$u_i(0-0)u_j$ for all $i \neq j$ and $1 \leq i \leq n-2$, $1 \leq j \leq n-2$,

Clearly, x is a weak king as $x(0-0)y$ and $x(0-0)y(1-0)u_i$ for all $1 \leq i \leq n-2$.. Also y is a weak king as $y(0-0)x$ and $y(1-0)u_i$ for all $1 \leq i \leq n-2$. Finally, every u_i is a weak king, since $u_i(0-0)u_j$, for all $i \neq j$ and $u_i(1-0)x$ and $u_i(1-0)x(0-0)y$. Thus D contains exactly n weak kings.

If $n = k-1$, the algorithm creates one additional arc $u_{n-2}(1-0)u_{n-3}$ in D. The resulting oriented graph contains exactly $n-1$ weak kings, since now u_{n-2} is not weakly reachable within two steps from u_{n-3} and so u_{n-3} is not a weak king.

If $n = k-2$ then the algorithm creates two additional arcs in D. namely $u_{n-2}(1-0)u_{n-3}$ and $u_{n-2}(1-0)u_{n-4}$. Thus D now contains exactly $n-2$ weak kings, with

u_{n-3} and u_{n-4} not being weak kings.

Continuing in this way, for any $3 \leq k \leq n$, the algorithm creates new arcs $u_{n-2}(1-0)u_i$ in D for all $k-2 \leq i \leq n-3$. The resulting graph D contains exactly k weak kings.

If $k = 2$, then D is constructed so that $x(0-0)y$, $x(1-0)u_i$. $y(1-0)u_i$ and $u_i(0-0)u_j$ for all $1 \leq i \leq n-2$, $1 \leq j \leq n-2$ and $i \neq j$. Thus D contains exactly two weak kings x and y.

Finally, D has exactly one weak king if it is constructed such that $x(0-0)y$, $u_1(1-0)x$, $u_1(1-0)y$ and $u_1(1-0)u_i$, $x(1-0)u_i$ and $y(1-0)u_i$ for all $2 \leq i \leq n-2$.

Due to the nested **for** loops the algorithm runs in $O(n^2)$ time.

Figure 30.13 shows a 6 vertex oriented graph with exactly 6 weak kings, Figure 30.14 shows a 6 vertex oriented graph with exactly 5 weak kings namely x, y, v_1, v_2 and v_4, Figure 30.15 shows a 6 vertex oriented graph with exactly 4 weak kings namely x, y. v_1 and v_4. Figure 30.16 shows a 6 vertex oriented graph with exactly 3 weak kings namely x, y and v_4 and Figure 30.17 shows a 6 vertex oriented graph with exactly 2 weak kings namely x and y.

The directional dual of a weak king is a weak serf, and thus a vertex u is a weak king of an oriented graph D if and only if u is a weak serf of $\bar{D}$, the converse of D. So by duality, there exists an oriented graph on n vertices with exactly s weak serfs for all integers $n \geq s \geq 1$. If $n = k \geq 1$, then every vertex in any such oriented graph is both a weak king and a weak serf. Also if $n > k \geq 1$, the oriented graph described in algorithm kWeakKings contains vertices which are both weak kings and weak serfs, and also contains vertices which are weak kings but not weak serfs and vice versa. These ideas give rise to the following problem. For what 4-tuples (n, k, s, b) does there exist an oriented graph with n vertices, exactly k weak kings, s weak serfs and that exactly b of the weak kings are also serfs? By analogy with (n, k, s, b)-tournaments, such oriented graphs are called (n, k, s, b)-oriented graphs. Without loss of generality, we assume that $k \geq s$. The following results by Pirzada and Shah address this problem.

Theorem 30.21 Pirzada, Shah, 2008 *If $n > k \geq s \geq 0$, then there exists no (n, k, s, s)-oriented graph.*

Theorem 30.22 Pirzada, Shah, 2008 *There exist (n, k, s, b)-oriented graphs, $n \geq k \geq s > b \geq 0$ and $n > 0$, $n \geq k + s - b$.*

Proof Let D_1 be the oriented graph with vertex set $\{x_1, y_1, u_1, u_2, \cdots, u_{k-b-2}\}$ and $x_1(0-0)y_1$, $u_i(1-0)x_1$, $u_i(0-1)y_1$ for all $1 \leq i \leq k-b-2$, and $u_i(0-0)u_j$ for all $i \neq j$.

Take the oriented graph D_2 with vertex set $\{x_2, y_2, v_1, v_2, \ldots, v_{b-2}\}$ and arcs defined as in D_1. Let D_3 be the oriented graph with vertex set $\{z_1, z_2, \ldots, z_{s-b}\}$ and $z_i(0-0)z_j$ for all i, j. Let D be the oriented graph $D_1 \cup D_2 \cup D_3$ (see Figure 30.19) with

$z_i(1-0)y_2$ for $1 \leq i \leq s-b$
$z_i(0-0)x_2$ for $1 \leq i \leq s-b$
$z_i(0-0)v_j$ for $1 \leq i \leq s-b$, $1 \leq j \leq b-2$
$x_1(1-0)z_i$, $y_1(1-0)z_i$ for $1 \leq i \leq s-b$

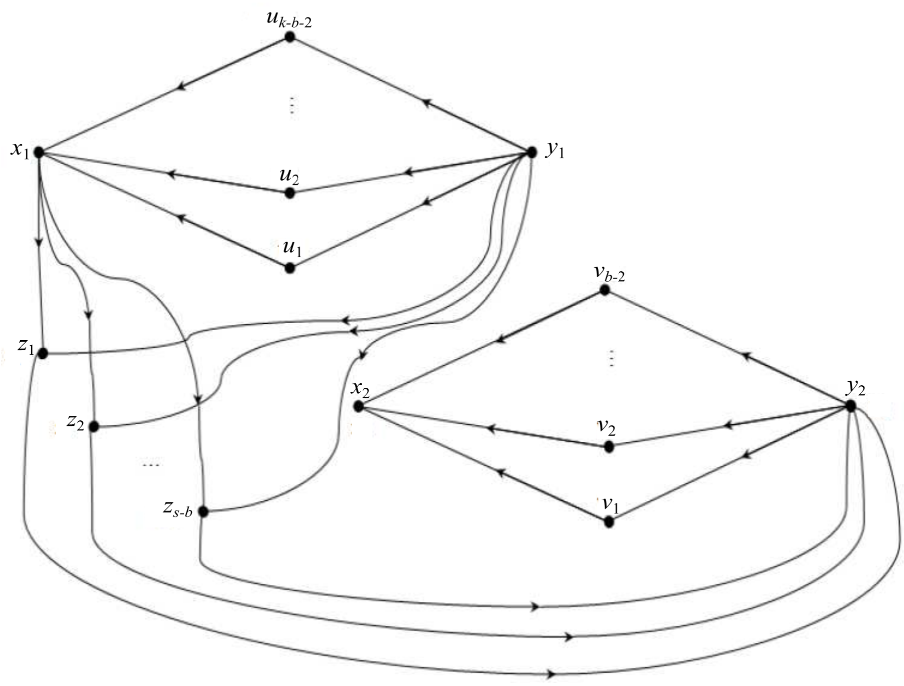


Figure 30.19 Construction of an (n, k, s, b)-oriented graph.

$u_r(0-0)z_i$ for $1 \le r \le k-b-2$, $1 \le i \le s-b$
$x_1(1-0)y_2$, $y_1(1-0)y_2$
$v_r(1-0)y_2$ for $1 \le r \le k-b-2$
$x_1(0-0)x_2$, $y_1(0-0)x_2$
$v_r(0-0)v_j$, for $1 \le r \le k-b-2$, $1 \le j \le b-2$.

Clearly D contains exactly k weak kings and the weak king set is $\{x_1, y_1\} \cup \{u_1, u_2, \ldots, u_{k-b-2}\} \cup \{x_2, y_2\} \cup \{v_1, v_2, \ldots, v_{b-2}\}$. D contains exactly s weak serfs with the weak serf set as $\{x_2, y_2\} \cup \{v_1, v_2, \ldots, v_{b-2}\} \cup \{z_1, z_2, \ldots, z_{s-b}\}$. Also from these k weak kings, exactly b are weak serfs. The weak king-serf set is $\{x_2, y_2\} \cup \{v_1, v_2, \ldots, v_{b-2}\}$. ■

Exercise 30.5-5 asks the reader to derive an algorithm for generating an (n, k, s, b)-oriented graph when the hypothesis of Theorem 30.22 is satisfied.

Exercises

30.5-1 Give an algorithm that generates an oriented graph with n vertices and exactly 2 kings. Prove the correctness of your algorithm.
30.5-2 Draw the graph D^* discussed in Example 30.4.
30.5-3 Prove that WEAK-KINGS is correct. Also determine its runtime.
30.5-4 Construct an oriented graph with six vertices and exactly one king.
30.5-5 Derive an algorithm that takes a 4-tuple (n, k, s, b) satisfying the hypothesis

of Theorem 30.22 as input and generates an (n, k, s, b)-oriented graph. Analyze the performance of your algorithm.

Problems

30-1 Optimal reconstruction of score sets

In connection with the reconstruction of graphs the basic questions are the existence and the construction of at least one corresponding graph. These basic questions are often solvable in polynomial time. In given sense optimal reconstruction is usually a deeper problem.

a) Analyse Exercise 30.1-1 and try to find a smaller tournament with score set $\{0, 1, 3, 6, 10\}$.

b) Write a back-track program which constructs the smallest tournament whose score set is $\{0, 1, 3, 6, 10\}$.

c) Write a back-track program which constructs the smallest tournament arbitrary given score set.

d) Estimate the running time of your programmes.

Hint. Read Yoo's proof.

30-2 Losing set

We define the ***losing score*** of a vertex as the in-degree of the vertex. The *loosing score set* of a tournament is the set of in-degrees of its vertices.

a) Give an argument to show that any set of nonnegative integers is the loosing score set of some tournament.

b) Given a set $L = \{r_1, r_2, \ldots, r_n\}$ of nonnegative integers with $r_1 < r_2 - r_1 < r_3 - r_2 < \cdots < r_n - r_{n-1}$, write an algorithm to generate a tournament with loosing score set L.

30-3 Imbalance set

Let

30-4 Unicity

Let

Chapter Notes

Many classical ans several contemporary graph theory textbooks are available to Readers. Such books are e.g. the books of Claude Berge [16] and László Lovász [97]. However, there is a dearth of books focusing on recent advances in the theory of digraphs. The book due to Bang-Jensen and Gutin [11] probably comes closest and the Reader can refer to it for a comprehensive treatment of the theoretical and algorithmic aspects of digraphs.

The books by Harary, Norman and Cartwright [59], and Chartrand, Lesniak and Zhang [25, 26], Gross and Yellen [57] present introductory material on tournaments and score structures. Moon's book on tournaments [107] is also a good resource but is now out of print.

The books A. Schrijver [138] and A. Frank [52] contain reach material on optimization problems connected with directed graphs.

The algorithms discussed in this chapter are not commonly found in literature. In particular the algorithms presented here for constructing tournaments and oriented graphs with special properties are not available in textbooks. Most of these algorithms are based on fairly recent research on tournaments and oriented graphs.

Majority of the researches connected with score sequences and score sets were inspired by the work of H. G. Landau, K. B. Reid and J. W. Moon. For classical and recent results in this area we refer to the excellent surveys by Reid [132, 135, 136]. Landau's pioneering work on kings and score structure appeared in 1953 [90]. Reid stated his famous score set conjecture in [132]. Partial results were proved by M. Hager [58]. Yao's proof of Reid's conjecture appeared in English in 1989 [154]. The comment of Q. Li on Reid's conjecture and Yao's proof was published in 2006 [92]. The construction of a new special tournament with a prescribed score set is due to Pirzada and Naikoo [125]. The score structure for 1-tournaments was introduced by H. G. Landau [90] and extended for k-tournaments by J. W. Moon in 1963. This result of Moon later was reproved by Avery for $k = 2$ and for arbitrary k by Kemnitz and Dolff [82]. Score sets of 2-tournaments were investigated by Pirzada and Naikoo in 2008 [128].

Authors of a lot of papers investigated the score sets of different generalized tournament, among others Pirzada, Naikoo and Chisthi in 2006 (bipartite graphs), Pirzada and Naikoo in 2006 [126] (k-partite graphs), Pirzada and Naikoo in 2006 [127] (kpartite tournaments@k-partite tournaments).

The basic results on kings are due to K. Brooks Reid [133, 134, 135, 136] and Vojislav Petrović [19, 119, 120, 121, 122].

The problem of the unicity of score sequences was posed and studied by Antal Iványi and Bui Minh Phong [79]. Another unicity results connected with tournaments was published e.g. by P. Tetali, J. W. Moon and recently by Chen et al. [27, 28, 108, 148].

The term king in tournaments was first used by Maurer [101]. Strong kings were introduced by Ho and Chang [62] and studied later by Chen et al. [27, 28], while Pirzada and Shah [130] introduced weak kings in oriented graphs. The problems connected with 3-kings and 4-kings were discussed by Tan in [145] and the construction of tournaments with given number of strong kings by Chen et al. in [28].

The difference of the out-degree and of the in-degree of a given vertex is called *the imbalance* of the given vertex. The imbalance set of directed multigraphs were studied by Pirzada, Naikoo, Samee and Iványi in [129], while the imbalance sets of multipartite oriented graphs by Pirzada, Al-Assaf and Kayibi [124].

Problem 1

Problem 2

Problem 3

Problem 4

An interesting new direction is proposed by "L. B. Beasley, D. E. Brown, and. K. B. Brooks in [14]: the problem is the reconstruction of tournaments on the base of the partially given out-degree matrix.

Bibliography

[1] J.-P. Allouche, M. Baake, J. Cassaigne, D. Damanik. Palindrome complexity. *Theoretical Computer Science*, 292:9–31, 2003. 1314

[2] N. Anbarci. Noncooperative foundations of the area monotonic solution. *The Quaterly Journal of Economics*, 108:245–258, 1993. 1350

[3] M.-C. Anisiu, V. Anisiu, Z. Kása. Total palindrome complexity of finite words. *Discrete Mathematics*, 310:109–114, 2010. 1314

[4] M.-C. Anisiu, Z. Blázsik, Z. Kása. Maximal complexity of finite words. *Pure Mathematics and Applications*, 13:39–48, 2002. 1313, 1314

[5] B. Arazi. Position recovery using binary sequences. *Electronic Letters*, 20:61–62, 1984. 1404

[6] P. Arnoux, C. Mauduit, I. Shiokawa, J. Tamura. Complexity of sequences defined by billiard in the cube. *Bulletin de la Société Mathématique de France*, 122(1):1–12, 1994. MR1259106 (94m:11028). 1314

[7] P. Avery. Score sequences of oriented graphs. *Journal of Graph Theory*, 15:251–257, 1991. MR1111988 (92f:05042). 1262, 1276

[8] M. Baake. A note on palindromicity. *Letters in Mathematical Physics*, 49(3):217–227, 1999. 1314

[9] E. Balas, S. G. Ceria, G. Cornuéjols. Cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993. 1260

[10] E. Balas, R. G. Jeroslow. Strengthening cuts for mixed integer programs. *European Journal of Operations Research*, 4:224–234, 1980. 1261

[11] J. Bang-Jensen, G. Gutin. *Digraphs: Theory, Algorithms and Applications* (1st edition). Springer, 2009. MR2472389 (2009k:05001). 1444

[12] C. Barnhart, E. L. Johnson, G. Nemhauser, M. Savelsbergh, P. Vance. Branch-and-Price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998. 1261

[13] E. Beale, R. Small. Mixed integer programming by a branch-and-bound technique. In W. A. Kalenich (Ed.), *Proceedings of IFIP Congress* New York, May 1865, 450–451 pages, 1965. Spartan Books. 1260

[14] L. B. Beasley, D. E. Brown, K. B. Reid. Extending partial tournaments. *Mathematical and Computer Modelling*, 50(1–2):287–291, 2009. MR2542610 (2010i:05141). 1276, 1445

[15] N. Bell, M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 1–11 pages. ACM, 2009. 1377

[16] C. Berge. *Graphs and Hypergraphs* (2nd revised edition). North-Holland, 1976. MR0384579 (52 #5453). 1262, 1444

[17] V. Berthé, L. Vuillon. Palindromes and two-dimensional sturmian sequences. *Journal of Automata, Languages and Combinatorics*, 6(2):121–138, 2001. 1314

[18] J-P. Borel, C. Reutenauer. Palindrome factors of billiard words. *Theoretical Computer Science*, 340:334–348, 2005. 1314

[19] D. Brcanov, V. Petrović. Toppling kings in multipartite tournaments by introducing new kings. *Discrete Mathematics*, 310(19):2250–2554, 2010. 1445

[20] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer. On the palindromic complexity of infinite words. *International Journal of Foundations of Computer Science*, 15(2):293–306, 2004. MR2071459 (2005d:68109). 1314

[21] N. G. Bruijn. A combinatorial problem. *Nederlandse Akademie van Wetenschappen. Proceedings.*, 49:758–764, 1946. 1393, 1394, 1404

[22] F. W. Burton, M. Huntbach, G. McKeown, V. Rayward-Smith. Parallelism in branch and bound algorithms. Technical Reportof Mathematical Algorithms Group-2, Internal Report CSA/3, University of East Anglia, Norwich, 1983. 1260

[23] J. Cassaigne. Complexité et facteurs spéciaux. *Bulletin of Bull. Belgian Mathematical Society Simon Stevin*, 4(1):67–88, 1997. 1314

[24] J. Cassaigne, I. Kaboré, T. Tapsoba. On a new notion of complexity on infinite words. *Acta Univ. Sapientiae, Mathematica*, 2:127–136, 2010. 1314

[25] G. Chartrand, L. Lesniak. *Graphs and Digraphs* (4th edition). Chapman and Hall/CRC Press, 2005. MR2107429 (2005g:05001). 1444

[26] G. Chartrand, L. Lesniak, P. Zhang. *Graphs and Digraphs* (5th edition). Chapman and Hall/CRC Press, 2010. 1444

[27] A. Chen. *The strong kings of tournaments*. PhD thesis, National Taiwan University of Science and Technology, 2007. 1445

[28] A. Chen, J. Chang, Y. Cheng Y. Wang. The existence and uniqueness of strong kings in tournaments. *Discrete Mathematics*, 308(12):2629–2633, 2008. MR2410473 (2009c:05095). 1445

[29] C. Choffrut, J. Karhumäki. Combinatorics of words. In G. Rozemberg (Ed.), *Handbook of Formal Languages*, Vol. I-III. Springer, 1997. MR2071459 (2005d:68109). 1313

[30] F. Chung, R. L. Graham, P. Diaconis. Universal cycles for combinatorial structures. *Discrete Mathematics*, 110(1–3):43–59, 1992. 1394, 1404

[31] Y. M. Chun. The equal loss principle for bargaining problems. *Economics Letters*, 26:103–106, 1988. 1350

[32] J. C. Cock. Toroidal tilings from de Bruijn cyclic sequences. *Discrete Mathematics*, 70(2):209–210, 1988. 1394, 1404

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms* 3rd edition. The MIT Press/McGraw-Hill, 2009. 1263, 1404

[34] L. J. Cummings, D. Weidemann. Embedded De Bruijn sequences. *Congressus Numer.*, 53:155–160, 1986. 1404

[35] D. Damanik, D. Zare. Palindrome complexity bounds for primitive substitution sequences. *Discrete Mathematics*, 222:259–267, 2000. 1314

[36] G. Dantzig, D. R. Fulkerson, S. Johnson. Solution of a large-scale traveling salesman problem to optimality. *Operations Research*, 2:393–410, 1954. 1261

[37] A. de Luca. On the combinatorics of finite words. *Theoretical Computer Science*, 218(1):13–39, 1999. MR1687752 (2000g:68123). 1314

[38] J. Dénes, A. D. Keedwell. Frequency allocation for a mobile radio telephone system. *IEEE Transactions on Communication*, 36:765–767, 1988. 1404

[39] X. Droubay, G. Pirillo. Palindromes and Sturmian words. *Theoretical Computer Science*, 223(1–2):73–85, 1999. MR1704637 (2000g:68122). 1314

[40] D. Eberly. *Game Physics*. Morgan Kaufmann Publishers, 2004. 1387

[41] C. H. Elzinga. Complexity of categorial time series. *Sociological Methods & Research*, 38(3):463–481, 2010. 1314

[42] C. H. Elzinga, S. Rahmann, H. Wung. Algorithms for subsequence combinatorics. *Theoretical Computer Science*, 409:394–404, 2008. 1314

[43] W. Engel (Ed.). *GPU Pro*. A K Peters, 2010. 1390

[44] C. T. Fan, S. Fan, S. Ma, M. Siu. On de Bruijn arrays. *Ars Combinatoria*, 19A:205–213, 1985. 1392, 1394

[45] S. Ferenczi. Complexity of sequences and dynamical systems. *Discrete Mathematics*, 206(1–3):145–154, 1999. MR1665394 (2000f:68093). 1314

[46] S. Ferenczi, Z. Kása. Complexity for finite factors of infinite sequences. *Theoretical Computer Science*, 218:177–195, 1999. MR1687792 (2000d:68121). 1314

[47] F. Fernando (Ed.). *GPU Gems*. Addison-Wesley, 2004. 1390

[48] T. Flye. Solution of problem 58. *Intermediare des Mathematiciens*, 1:107–110, 1894. 1392, 1393, 1404

[49] N. P. Fogg. *Substitutions in dynamics, arithmetics, and combinatorics (Lecture Notes in Mathematics, Vol. 1794)*. Springer, 2002. 1313

[50] F. Forgó, J. Szép, F. Szidarovszky. *Introduction to the Theory of Games: Concepts, Methods and Applications*. Kluwer Academic Publishers, 1999. 1350

[51] J. J. H. Forrest, J. P. H. Hirst, J. Tomlin. Practical solution of large mixed integer programming problems with umpire. *Management Science*, 20:736–773, 1974. 1260

[52] A. Frank. *Connections in Combinatorial Optimization*. Oxford University Press, 2011 (to appear). 1445

[53] N. Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *Parallel and Distributed Processing*, IPDPS 2008, 1–8 pages. IEEE, 2008. 1377

[54] F. Glover. A multiphase-dual algorithm for zero-one integer programming problem. *Operations Research*, 13:879–919, 1965. 1261

[55] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of American Mathematical Society*, 64:275–278, 1958. 1261

[56] I. Good. Normally recurring decimals. *Journal of London Mathematical Society*, 21:167–169, 1946. 1393, 1404

[57] J. Gross, J. Yellen. *Handbook of Graph Theory* (2nd edition). CRC Press, 2006. MR2181153 (2006f:05001). 1276, 1444

[58] M. Hager. On score sets for tournaments. *Discrete Mathematics*, 58(1):25–34, 1986. 1423, 1445

[59] F. Harary, R. Norman, D. Cartwright. *An Introduction to the Theory of Directed Graphs*. John Wiley and Sons, Inc, 1965. 1444

[60] M. J. Harris, W. V. Baxter, T. Scheuerman, A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS'03, 92–101 pages, 2003. Eurographics Association. 1382

[61] J. F. Harsanyi, R. Selten. A generalized nash solution for two-person bargaining with incomplete information. *Management Science*, 12(2):80–106, 1972. 1350

[62] T. Ho, J. Chang. Sorting a sequence of strong kings in a tournament. *Information Processing Letters*, 87(6):317–320, 2003. 1445

[63] M. Horváth, A. Iványi. Growing perfect cubes. *Discrete Mathematics*, 308:4378–4388, 2008. 1404

[64] P. Horváth, D. Illés. Sph-based fluid simulation in distributed environment. In *MIPRO 2009: 32nd International Convention on Information and Communication Technology, Electronics and Microelectronics*, 249–257 pages, 2009. 1389

[65] G. Hurlbert, G. Isaak. On the de Bruijn torus problem. *Combinatorial Theory Series A*, 164(1):50–62, 1993. 1393, 1404

[66] G. Hurlbert, G. Isaak. A meshing technique for de bruijn tori. *Contemporary Mathematics*, 164(1):50–62, 1994. 1393, 1394, 1396, 1404

[67] G. Hurlbert, G. Isaak. New constructions for De Bruijn tori. *Designs, Codes and Cryptography*, 1:47–56, 1995. 1404

[68] G. Hurlbert, G. Isaak. On higher dimensional perfect factors. *Ars Combinatoria*, 45:229–239, 1997. 1404

[69] G. Hurlbert, C. J. Mitchell, K. G. Paterson. On the existence of the Bruijn tori with two by two window property. *Combinatorial Theory Series A*, 76(2):213–230, 1996. 1393, 1404

[70] T. Ibataki. Computational efficiency of approximate branch-and-bound algorithms. *Mathematics of Operations Research*, 1:287–298, 1976. 1260

[71] T. I, S. Inenaga, H. Bannai, M. Takeda. Counting and verifying maximal palindromes. In E.Chavez, S. Lonardo (Eds.), *Proceeding of 17th Internatioonal Symposium on String Processing and Information Retrieval* (Los Cabos, Mexico, October 11–79, 2010), Lecture Notes in Computer Science **6393**, 135–146 pages, 2010. Springer-Verlag. 1314

[72] L. Ilie. Combinatorial complexity measures for strings. In *Recent Advances in Formal Languages*, 149–170 pages. Springer-Verlag, 2006. 1314

[73] G. Isaak. Construction for higher dimensional perfect multifactors. *Aequationes Mathematicae*, 178:153–160, 2002. 1404

[74] A. Iványi. On the d-complexity of words. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computarorica*, 8:69–90, 1987. 1313, 1314, 1405

[75] A. Iványi. Construction of infinite De Bruijn arrays. *Discrete Applied Mathhematics*, 22:289–293, 1988/89. 1405

[76] A. Iványi. Construction of three-dimensional perfect matrices. *Ars Combinatoria*, 29C:33–40, 1990. 1405

[77] A. Iványi. Reconstruction of complete interval tournaments. *Acta Univ. Sapientiae, Informatica*, 1(1):71–88, 2009. 1262, 1263, 1266, 1271, 1273, 1275, 1276

[78] A. Iványi. Reconstruction of complete interval tournaments. II. *Acta Univ. Sapientiae, Mathematica*, 2(1):47–71, 2010. 1275, 1276

[79] A. Iványi, B. M. Phong. On the unicity of multitournaments. In *Fifth Conference on Mathematics and Computer Science* (Debrecen, June 9–12, 2004), 2004. http://compalg.inf.elte.hu/ tony/Publications. 1445

[80] A. Iványi, Z. Tóth. Existence of De Bruijn words. In I. Peák (Ed.), *Second Conference on Automata, Languages and Programming Systems (Salgótarján, 1988)*, 165–172 pages. Karl Marx University of Economics, 1988. 1393, 1405

[81] E. Kalai, M. Smorodinsky. Other solution to Nash' bargaining problem. *Econometrica*, 43:513–518, 1975. 1350

[82] A. Kemnitz, S. Dolff. Score sequences of multitournaments. *Congressus Numerantium*, 127:85–95, 1997. MR1604995 (98j:05072). 1445

[83] Khronos. OpenCL overview. 2010. http://www.khronos.org/opencl/. 1353

[84] D. E. Knuth. *The Art of Computer Programming, Volume 4., Fascicle 2. Generating All Tuples and Permutations.* Addison-Wesley, 2005. 1394

[85] D. E. Knuth. *The Art of Computer Programming, Volume 4A. Combinatorial Algorithms.* Addison-Wesley, 2011. 1276, 1405

[86] Z. Kása. On the d-complexity of strings. *Pure Mathematics and Applications*, 9:119–128, 1998. 1314

[87] Z. Kása. On scattered subword complexity of finite words. *Acta Univ. Sapientiae, Informatica*, 3(1), 2010 (accepted). 1314

[88] Z. Kása. Super-d-complexity of finite words. In *Proceedings of 8th Joint Conference on Mathematics and Computer Science,* (July 14–17, 2010, Komárno, Slovakia). János Selye University, 2011. 1314

[89] A. H. Land, A. Doig. An automatic method of solving Discrete Programming problems. *Econometrica*, 28:497–520, 1960. 1260

[90] H. G. Landau. On dominance relations and the structure of animal societies. III. the condition for a score sequence. *Bulletin of Mathematical Biophysics*, 15:143–148, 1953. MR0054933 (14,1000e). 1262, 1263, 1267, 1275, 1276, 1445

[91] F. Levé, P. Séébold. Proof of a conjecture on word complexity. *Bull. Belg. Math. Soc. Simon Stevin*, 8:277–291, 2001. 1314

[92] Q. Li. Some results and problems in graph theory. *Annals of New York Academy of Sciences*, 576:336–343, 2006. 1445

[93] M. Lothaire. *Applied Combinatorics on Words.* Cambridge University Press, 2005. 1313

[94] M. Lothaire. *Combinatorics on Words.* Cambridge University Press, 1997. 2nd edition. 1313

[95] M. Lothaire. *Algebraic Combinatorics on Words.* Cambridge University Press, 2002. 1313

[96] H. (Ed.). *GPU Gems*. Addison-Wesley, 2008. 1390

[97] L. Lovász. *Combinatorial Problems and Exercises*. Academic Press, 1979. MR0537284 (80m:05001). 1394, 1405, 1444

[98] L. Lovász, A. Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal on Optimization*, 1:166–190, 1991. 1260

[99] M. Magdics, G. Klár. Rule-based geometry synthesis in real-time. In W. Engel (Ed.), *GPU Pro: Advanced Rendering Techniques*, 41–66 pages. A K Peters, 2010. 1357

[100] M. H. Martin. A problem in arrangements. *Bulletin of American Mathematical Society*, 40:859–864, 1934. 1394, 1405

[101] S. B. Maurer. The king chicken theorems. *Mathematical Magazine*, 53:67–80, 1980. MR0567954 (81f:05089). 1445

[102] Microsoft. HLSL. 2010. http://msdn.microsoft.com/en-us/library/bb509561(v=VS.85).aspx. 1353

[103] C. J. Mitchell. Aperiodic and semi-periodic perfect maps. *IEEE Transactions on Information Theory*, 41(1):88–95, 1995. 1405

[104] S. Molnár, F. Szidarovszky. *Konfliktushelyzetek megoldási módszerei*. SZIE, 2010. 1351

[105] J. W. Moon. On the score sequence of an n-partite tournament. *Canadian Journal of Mathematics*, 5:51–58, 1962. aaaaaa. 1262, 1275, 1276

[106] J. W. Moon. An extension of landau's theorem on tournaments. *Pacific Journal of Mathematics*, 13(4):1343–1346, 1963. MR0155763 (27 #5697). 1262, 1267, 1276

[107] J. W. Moon. *Topics on Tournaments*. Holt, Rinehart, and Winston, 1963. MR0256919 (41 #1574). 1262, 1444

[108] J. W. Moon. The number of tournaments with a unique spanning cycle. *Journal of Graph Theory*, 53(3):303–308, 1982. MR0666798 (84g:05078). 1445

[109] J. Nash. The bargaining problem. *Econometrica*, 18:155–162, 1950. 1350

[110] J. Neider, T. Davis, W. Mason. *The Official Guide to Learning OpenGL*. Addison-Wesley, 1994. http://fly.cc.fer.hr/˜unreal/theredbook/appendixg.html. 1357

[111] NVIDIA. Cg homepage. 2010. http://developer.nvidia.com/page/cg_main.html. 1353, 1390

[112] NVIDIA. CUDA zone. 2010. http://www.nvidia.com/object/cuda_home_new.html. 1353, 1390

[113] J. D. Owens, D. Luebke, N. Govindaraju, M. J. Harris, J. Krüger, A. Lefohn, T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. 1390

[114] K. G. Paterson. Perfect maps. *IEEE Transactions on Information Theory*, 40(3):743–753, 1994. 1394, 1405

[115] K. G. Paterson. New classes of perfect maps. i. *Combinatorial Theory Series A*, 73(2):302–334, 1996. 1393, 1394, 1405

[116] K. G. Paterson. New classes of perfect maps. ii. *Combinatorial Theory Series A*, 73(2):335–345, 1996. 1394, 1405

[117] E. M. Petriou, J. Basran. On the position measurement of automated guided vehicle using pseudorandom encoding. *IEEE Transactions on Instrumentation and Measurement*, 38:799–803, 1989. 1405

[118] E. M. Petriou, J. Basran, F. Groen. Automated guided vehicle position recovery. *IEEE Transactions on Instrumentation and Measurement*, 39:254–258, 1990. 1405

[119] V. Petrović. Bipartite kings. *Novi Sad Journal of Mathematics???*, 98(3):237–238, 1995. MR144406 (92h:05063). 1445

[120] V. Petrović. Kings in bipartite tournaments. *Discrete Mathematics*, 173(1–3):117–119, 1997. MR1468848 (98g:05067). 1445

[121] V. Petrović, C. Thomassen. Kings in k-partite tournaments. *Discrete Mathematics*, 98(3):237–238, 1991. MR144406 (92h:05063). 1445

[122] V. Petrović, M. Treml. 3-kings in 3-partite tournaments. *Discrete Mathematics*, 173(2–3):177–186, 1998. MR1468848 (98g:05067). 1445

[123] M. Pharr (Ed.). *GPU Gems 2*. Addison-Wesley, 2005. 1390

[124] S. Pirzada, A. M. Al-Assaf, K. Kayibi, A. On imbalances in oriented multipartite graphs. *Acta Universitatis Sapientiae, Mathematica*, 3(1), 2011 (to appear). 1445

[125] S. Pirzada, T. A. Naikoo. On score sets in tournaments. *Vietnam Journal of Mathematics*, 34(2):157–161, 2006. 1445

[126] S. Pirzada, T. A. Naikoo. Score sets in oriented k-partite graphs. *AKCE International Journal of Graphs and Combinatorics*, 3(2):135–145, 2006. 1445

[127] S. Pirzada, T. A. Naikoo. Score sets in oriented k-partite tournaments. *Journal of Applied Mathematics and Computing*, 22(1–2):237–245, 2006. 1445

[128] S. Pirzada, T. A. Naikoo. Score sets in oriented graphs. *Applicable Analysis and Discrete Mathematics*, 2:107–113, 2008. 1445

[129] S. Pirzada, T. A. Naikoo, U. Samee, A. Iványi. Imbalances in directed multigraphs. *Acta Universitatis Sapientiae, Mathematica*, 2(1):47–71, 2010. 1445

[130] S. Pirzada, N. A. Shah. Kings and serfs in oriented graphs. *Mathematica Slovaca*, 58(3):277–288, 2008. 1445

[131] I. S. Reed R. Stewart. Note on the existence of perfect maps. *IRE Transactions on Information Theory*, 8:10–12, 1962. 1405

[132] K. B. Reid. Score sets for tournaments. *Congressus Numerantium*, 21:607–618, 1978. 1263, 1423, 1445

[133] K. B. Reid. Tournaments with prescribed numbers of kings and serfs. *Congressus Numerantium*, 29:809–826, 1980. MR0608478 (82d:05064). 1445

[134] K. B. Reid. Every vertex a king. *Discrete Mathematics*, 38(1):93–98, 1982. MR0676524 (84d:05091). 1445

[135] K. B. Reid. Tournaments: Scores, kings, generalisations and special topics. *Congressus Numerantium*, 115:171–211, 1996. MR1411241 (97f:05081). 1262, 1445

[136] K. B. Reid. Tournaments. In J. L. Gross, J. Yellen (Eds.), *Handbook of Graph Theory*, 156–184 pages. CRC Press, 2004. 1263, 1445

[137] G. Reinelt (Ed.). *The Traveling Salesman.* Lecture Notes in Computer Science. Springer, 2004. 1261

[138] A. Schrijver (Ed.). *Combinatorial Optimization. Vol A. Paths, Flows, Matchings; Vol. B. Matroids, Trees, Stable Sets. Vol. C. Disjoint Paths, Hypergraphs.* Springer, 2003. MR1956924 (2004b:90004a), MR1956925 (2004b:90004b), MR1956926 (2004b:90004c). 1445

[139] J. O. Shallit. On the maximum number of distinct factors in a binary string. *Graphs and Combinatorics*, 9:197–200, 1993. 1314

[140] H. D. Sherali, W. P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3:411–430, 1990. 1260

[141] J. Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 121–128 pages, 1999. 1381

[142] F. Szidarovszky, M. E. Gershon. *Techniques for Multiobjective Decision Making in Systems Management.* Elsevier Press, 1986. 1350

[143] L. Szirmay-Kalos, L. Szécsi, M., Sbert. *GPU-Based Techniques for Global Illumination Effects.* Morgan and Claypool Publishers, 2008. 1354, 1390

[144] L. Szirmay-Kalos B. Tóth, M. Magdics, D. Légrády, A. Penzov. Gamma photon transport on the GPU for PET. *Lecture Notes on Computer Science*, 5910:433–440, 2010. 1370

[145] B. Tan. On the 3-kings and 4-kings in multipartite tournaments. *Discrete Mathematics*, 306(21):2702–2710, 2006. MR2263727 (2007f:05079). 1445

[146] N. Tatarchuk, P. Sander, J. L. Mitchell. Early-z culling for efficient GPU-based fluid simulation. In W. Engel (Ed.), *ShaderX 5: Advanced Rendering Techniques*, 553–564 pages. Charles River Media, 2006. 1384

[147] A. D. Taylor, A. Pacelli. *Mathematics and Politics.* Springer-Verlag, 2008. 1350

[148] P. Tetali. A characterization of unique tournaments. *Journal of Combinatorial Theory, Ser. B.*, 72(1):157–159, 1998. MR1604650 (98h:05086). 1445

[149] W. Thomson. Cooperative models of bargaining. In R. J. Aumann, S. Hart (Eds.), *Handbook of Game Theory.* Elsevier, 1994. 1350

[150] J. Tomlin. An improved branch-and-bound method for integer programming. *Operations Research*, 31:1070–1075, 1971. 1260

[151] O. G. Troyanskaya, O. Arbell, Y. Koren, G. M. Landau, A. Bolshoy. Sequence complexity profiles of prokaryotic genomic sequences: A fast algorithm for calculating linguistic complexity. *Bioinformatics*, 18:679–688, 2002. 1314

[152] B. Vizvári. *Integer Programming.* Hungarian. Tankönyvkiadó, 1990. 1260

[153] N. Vörös. On the complexity of symbol sequences. In *Conference of Young Programmers and Mathematicians* (ed. A. Iványi), Eötvös Loránd University, Budapest, 43–50 pages, 1984. 1405

[154] T. Yao. On Reid conjecture of score sets for tournaments. *Chinese Science Bulletin*, 10:804–808, 1989. MR1022031 (90g:05094). 1263, 1423, 1445

[155] F. Zhe, F. Qiu, A. Kaufman, S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC'04, 47–59 pages, 2004. IEEE Computer Society. 1390

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Subject Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *exe* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.