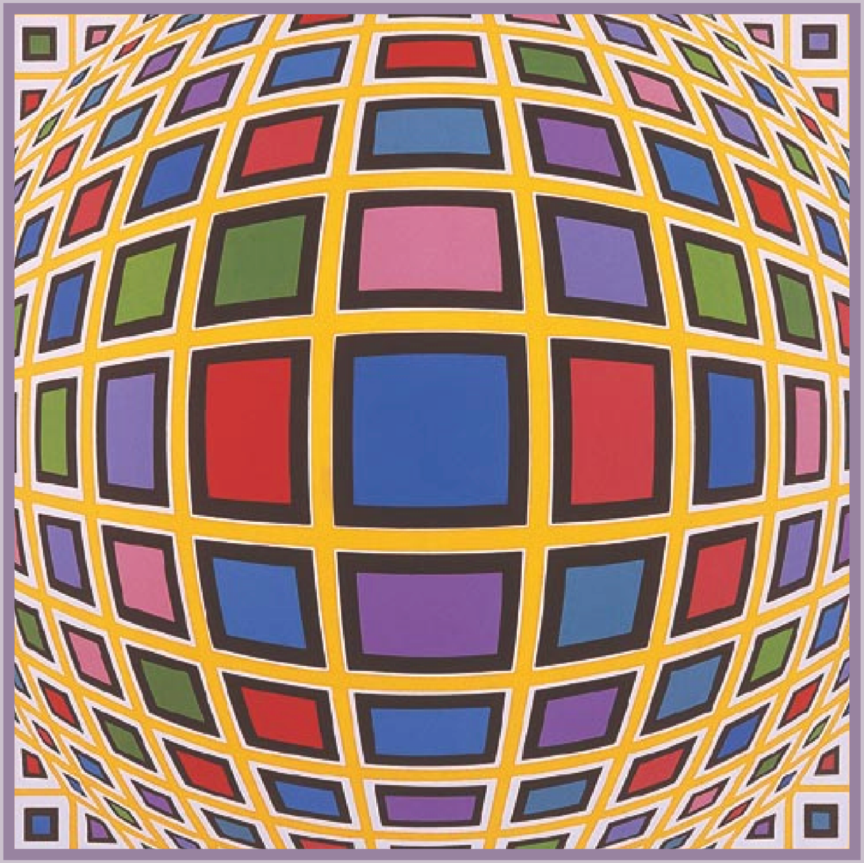
ALGORITHMS of Informatics



volume 2.

ELTE Faculty of Informatics

ALGORITHMS OF INFORMATICS

Volume 2.

COMPUTER NETWORKS AND OPTIMIZATION

ELTE EÖTVÖS KIADÓ
Budapest, 2006

Editor: Antal Iványi

Authors: Burkhard Englert, Dariusz Kowalski, Grzegorz Malewicz, Alexander Shvartsman (Chapter 7), Ferenc Szidarovszky (8), Csanád Imreh (9), Antal Iványi and Claudia Leopold (10), Tibor Gyires and László Lakatos (11), Systolic Systems (12)

Validators: István Majzik (Chapter 7), János Mayer (8), Béla Vizvári (9), Dezső Sima (10), János Sztrik (11), Dezső Sima (12)

ISBN: 963 463 664 0

Published by ELTE EÖTVÖS KIADÓ
Budapest, Szerb utca 21–23.
Hungary
Telephone/facsimile: 411-6740
Internet: http://www.elte.hu/szervezet/eotvos_kiado.html
E-mail: eotvoskiado@ludens.elte.hu

Responsible publisher: András Pándi
Cover design: Antal Iványi

Printed and bound by ???

Introduction

7. Distributed Systems

8. Game Theory

In many situations in engineering and economy there are cases when the conflicting interests of several decision makers have to be taken into account simultaneously, and the outcome of the situation depends on the actions of these decision makers. One of the most popular methodology and modeling is based on game theory.

Let N denote the number of decision makers (who will be called ***players***), and for each $k = 1, 2, \ldots, N$ let S_k be the set of all feasible actions of player $\mathcal{P}_k$. The elements $s_k \in S_k$ are called ***strategies*** of player $\mathcal{P}_k$, S_k is the ***strategy set*** of this player. In any realization of the ***game*** each player selects a strategy, then the vector $\mathbf{s} = (s_1, s_2, \ldots, s_N)$ ($s_k \in S_k$, $k = 1, 2, \ldots, N$) is called a ***simultaneous strategy vector*** of the players. For each $\mathbf{s} \in S = S_1 \times S_2 \times \cdots \times S_N$ each player has an outcome which is assumed to be a real value. This value can be imagined as the utility function value of the particular outcome, in which this function represents how player $\mathcal{P}_k$ evaluates the outcomes of the game. If $f_k(s_1, \ldots, s_N)$ denotes this value, then $f_k : S \to \mathbb{R}$ is called the ***payoff function*** of player $\mathcal{P}_k$. The value $f_k(\mathbf{s})$ is called the ***payoff*** of player $\mathcal{P}_k$ and $(f_1(\mathbf{s}), \ldots, f_N(\mathbf{s}))$ is called the ***payoff vector***. The number N of players, the sets S_k of strategies and the payoff functions f_k ($k = 1, 2, \ldots, N$) completely determine and define the N-person game. We will also use the notation $G = \{N; S_1, S_2, \ldots, S_N; f_1, f_2, \ldots, f_N\}$ for this game.

The solution of game G is the ***Nash-equilibrium***, which is a simultaneous strategy vector $\mathbf{s}^\star = (s_1^\star, \ldots, s_N^\star)$ such that for all k,

1. $s_k^\star \in S_k$;

2. for all $s_k \in S_k$,

$$f_k(s_1^\star, s_2^\star, \ldots, s_{k-1}^\star, s_k, s_{k+1}^\star, \ldots, s_N^\star) \le f_k(s_1^\star, s_2^\star, \ldots, s_{k-1}^\star, s_k^\star, s_{k+1}^\star, \ldots, s_N^\star) . \tag{8.1}$$

Condition 1 means that the k-th component of the equilibrium is a feasible strategy of player $\mathcal{P}_k$, and condition 2 shows that none of the players can increase its payoff by unilaterally changing its strategy. In other words, it is the interest of all players to keep the equilibrium since if any player departs from the equilibrium, its payoff does not increase.

		Player $\mathcal{P}_2$	
		N	C
Player $\mathcal{P}_1$	N	$(-2,-2)$	$(-10,-1)$
	C	$(-1,-10)$	$(-5,-5)$

Figure 8.1. Prisoner's dilemma.

8.1. Finite Games

Game G is called finite if the number of players is finite and all strategy sets S_k contain finitely many strategies. The most famous two-person finite game is the ***prisoner's dilemma,*** which is the following.

8.1. Example. The players are two prisoners who committed a serious crime, but the prosecutor has only insufficient evidence to prosecute them. The prisoners are held in separate cells and cannot communicate, and the prosecutor wants them to cooperate with the authorities in order to get the needed additional information. So $N = 2$, and the strategy sets for both players have two elements: cooperating (C), or not cooperating (N). It is told to both prisoners privately that if he is the only one to confess, then he will get only a light sentence of 1 year, while the other will go to prison for a period of 10 years. If both confess, then their reward will be a 5 year prison sentence each, and if none of them confesses, then they will be convicted to a less severe crime with sentence of 2 years each. The objective of both players are to minimize the time spent in prison, or equivalently to maximize its negative. Figure 8.1 shows the payoff values, where the rows correspond to the strategies of player $\mathcal{P}_1$, the columns show the strategies of player $\mathcal{P}_2$, and for each strategy pair the first number is the payoff of player $\mathcal{P}_1$, and the second number is the payoff of player $\mathcal{P}_2$. Comparing the payoff values, it is clear that only (C, C) can be equilibrium, since

$$\begin{aligned} f_2(\text{N,N}) = -2 &\quad<\quad f_2(\text{N,C}) = -1,\\ f_1(\text{N,C}) = -10 &\quad<\quad f_1(\text{C,C}) = -5,\\ f_2(\text{C,N}) = -10 &\quad<\quad f_2(C, C) = -5.\end{aligned}$$

The strategy pair (C, C) is really an equilibrium, since

$$\begin{aligned} f_1(\text{C,C}) = -5 > f_1(\text{N,C}) = -10,\\ f_2(\text{C,C}) = -5 > f_2(\text{C,N}) = -10.\end{aligned}$$

In this case we have a unique equilibrium.

The existence of an equilibrium is not guaranteed in general, and if equilibrium exists, it might not be unique.

8.2. Example. Modify the payoff values of Figure 8.1 as shown in Figure 8.2. It is easy to see that no equilibrium exists:

$$\begin{aligned} f_1(\text{N,N}) = 1 &\quad<\quad f_1(\text{C,N}) = 2,\\ f_2(\text{C,N}) = 4 &\quad<\quad f_2(\text{C,C}) = 5,\\ f_1(\text{C,C}) = 0 &\quad<\quad f_1(\text{N,C}) = 2,\\ f_2(\text{N,C}) = 1 &\quad<\quad f_2(\text{N,N}) = 2.\end{aligned}$$

		Player $\mathcal{P}_2$	
		N	C
Player $\mathcal{P}_1$	N	$(1, 2)$	$(2, 1)$
	C	$(2, 4)$	$(0, 5)$

Figure 8.2. Game with no equilibrium.

If all payoff values are identical, then we have multiple equilibria: any strategy pair is an equilibrium.

8.1.1. Enumeration

Let N denote the number of players, and for the sake of notational convenience let $s_k^{(1)}, \ldots, s_k^{(n_k)}$ denote the feasible strategies of player $\mathcal{P}_k$. That is, $S_k = \{s_k^{(1)}, \ldots, s_k^{(n_k)}\}$. A strategy vector $\mathbf{s}^\star = (s_1^{(i_1)}, \ldots, s_N^{(i_N)})$ is an equilibrium if and only if for all $k = 1, 2, \ldots, N$ and $j \in \{1, 2, \ldots, n_k\} \setminus i_k$,

$$f_k(s_1^{(i_1)}, \ldots, s_{k-1}^{(i_{k-1})}, s_k^{(j)}, s_{k+1}^{(i_{k+1})}, \ldots, s_N^{(i_N)}) \le f_k(s_1^{(i_1)}, \ldots, s_{k-1}^{(i_{k-1})}, s_k^{(i_k)}, s_{k+1}^{(i_{k+1})}, \ldots, s_N^{(i_N)}). \qquad (8.2)$$

Notice that in the case of finite games inequality (8.1) reduces to (8.2).

In applying the enumeration method, inequality (8.2) is checked for all possible strategy N-tuples $\mathbf{s}^\star = (s_1^{(i_1)}, \ldots, s_N^{(i_N)})$ to see if (8.2) holds for all k and j. If it does, then $\mathbf{s}^\star$ is an equilibrium, otherwise not. If during the process of checking for a particular $\mathbf{s}^\star$ we find a k and j such that (8.2) is violated, then $\mathbf{s}^\star$ is not an equilibrium and we can omit checking further values of k and j. This algorithm is very simple, it consists of $N + 2$ imbedded loops with variables $i_1, i_2, \ldots, i_N, k$ and j.

The maximum number of comparisons needed equals

$$\left(\prod_{k=1}^{N} n_k\right)\left(\sum_{k=1}^{N}(n_k - 1)\right),$$

however in practical cases it might be much lower, since if (8.2) is violated with some j, then the comparison must stop for the same strategy vector.

The algorithm can formally be given as follows:

```
 1  for $i_1 \leftarrow 1$ to $n_1$
 2      do for $i_2 \leftarrow 1$ to $n_2$
 3              $\ddots$
 4              do for $i_N \leftarrow 1$ to $n_N$
 5                  do $key \leftarrow 0$
 6                  for $k \leftarrow 1$ to $N$
 7                      do for $j \leftarrow 1$ to $n_k$
 8                          do if (8.2) fails
 9                              then $key \leftarrow 1$ and go to 10
10                  if  $key = 0$
11                      then $(s_1^{(i_1)}, \ldots, s_N^{(i_N)})$ is equilibrium
12                      and give message accordingly
```

Consider next the two-person case, N=2, and introduce the $n_1 \times n_2$ real matrixes $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ with (i, j) elements $f_1(i, j)$ and $f_2(i, j)$ respectively. Matrixes $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ are called the ***payoff matrixes*** of the two players. A strategy vector $\left(s_1^{(i_1)}, s_2^{(i_2)}\right)$ is an equilibrium if and only if the (i_1, i_2) element in matrix $\mathbf{A}^{(1)}$ is the largest in its column, and in matrix $\mathbf{A}^{(2)}$ it is the largest in its row. In the case when $f_2 = -f_1$, the game is called ***zero-sum***, and $\mathbf{A}^{(2)} = -\mathbf{A}^{(1)}$, so the game can be completely described by the payoff matrix $\mathbf{A}^{(1)}$ of the first player. In this special case a strategy vector $(s_1^{(i_1)}, s_2^{(i_2)})$ is an equilibrium if and only if the element (i_1, i_2) is the largest in its column and smallest in its row. In the zero-sum cases the equilibria are also called the ***saddle points*** of the games. Clearly, the enumeration method to find equilibria becomes more simple since we have to deal with a single matrix only.

The simplified algorithm is as follows:

```
 1  for $i_1 \leftarrow 1$ to $n_1$
 2      do for $i_2 \leftarrow 1$ to $n_2$
 3          do $key \leftarrow 0$
 4          for $j \leftarrow 1$ to $n_1$
 5              do if $a_{ji_2}^{(1)} > a_{i_1 i_2}^{(1)}$
 6                  then $key \leftarrow 1$
 7                  and go to 12
 8          for $j \leftarrow 1$ to $n_2$
 9              do if $a_{i_1 j}^{(2)} > a_{i_1 i_2}^{(2)}$
10                  then $key \leftarrow 1$
11                  and go to 12
12          if $key = 0$
13              then give message that $(s_{i_1}^{(1)}, s_{i_2}^{(2)})$ is equilibrium
```

8.1.2. Games Represented by Finite Trees

Many finite games have the common feature that they can be represented by a finite directed tree with the following properties:

1. there is a unique root of the tree (which is not the endpoint of any arc), and the game starts at this node;
2. to each node of the tree a player is assigned and if the game reaches this node at any time, then this player will decide on the continuation of the game by selecting an arc originating from this node. Then the game moves to the endpoint of the chosen arc;
3. to each terminal node (in which no arc originates) an N-dimensional real vector is assigned which gives the payoff values for the players if the game terminates at this node;
4. each player knows the tree, the nodes he is assigned to, and all payoff values at the terminal nodes.

For example, the chess-game satisfies the above properties in which $N = 2$, the nodes of the tree are all possible configurations on the chessboard twice: once with the white player and once with the black player assigned to it. The arcs represent all possible moves of the assigned player from the originating configurations. The endpoints are those configurations in which the game terminates. The payoff values are from the set $\{1, 0, -1\}$ where 1 means win, -1 represents loss, and 0 shows that the game ends with a tie.

Theorem 8.1 *All games represented by finite trees have at least one equilibrium.*

Proof. We present the proof of this result here, since it suggests a practical algorithm to find equilibria. The proof goes by induction with respect to the number of nodes of the game tree. If the game has only one node, then clearly it is the only equilibrium.

Assume next that the theorem holds for any tree with less than n nodes ($n \geq 2$), and consider a game T_0 with n nodes. Let R be the root of the tree and let $r_1, r_2, \ldots, r_m$ ($m < n$) be the nodes connected to R by an arc. If $T_1, T_2, \ldots, T_m$ denote the disjoint subtrees of T_0 with roots $r_1, r_2, \ldots, r_m$, then each subtree has less than n nodes, so each of them has an equilibrium. Assume that player $\mathcal{P}_k$ is assigned to R. Let $e_1, e_2, \ldots, e_m$ be the equilibrium payoffs of player $\mathcal{P}_k$ on the subtrees $T_1, T_2, \ldots, T_m$ and let $e_j = \max\{e_1, e_2, \ldots, e_m\}$. Then player $\mathcal{P}_k$ will move to node r_j from the root, and then the equilibrium continues with the equilibrium obtained on the subtree T_j. We note that not all equilibria can be obtained by this method, however the payoff vectors of all equilibria, which can obtained by this method, are identical. ■

We note that not all equilibria can be obtained by this method, however the payoff vectors of all equilibria, which can be obtained by this method, are identical.

The proof of the theorem suggests a dynamic programming-type algorithm which is called ***backward induction***. It can be extended to the more general case when the tree has chance nodes from which the continuations of the game are random according to given discrete distributions.

The solution algorithm can be formally presented as follows. Assume that the nodes are numbered so each arc connects nodes i and j only for $i < j$. The root has to get the smallest number 1, and the largest number n is given to one of the terminal nodes. For each node i

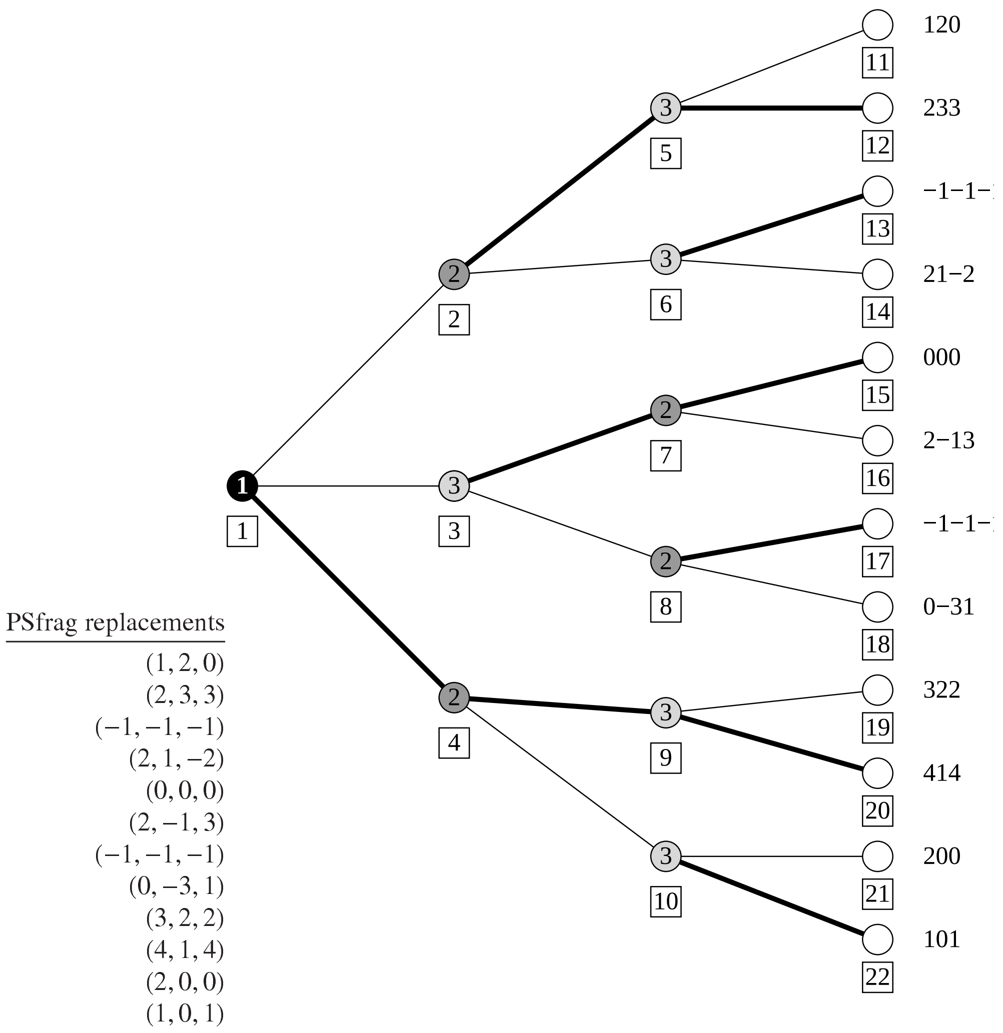


Figure 8.3. Finite tree of Example 10.3.

let $J(i)$ denote the set of all nodes j such that there is an arc from i to j. For each terminal node i, $J(i)$ is empty, and let $\mathbf{p}^{(i)} = (p_1^{(i)}, \ldots, p_N^{(i)})$ denote the payoff vector associated to this node. And finally we will denote player assigned to node i by K_i for all i. The algorithm starts at the last node n and moves backward in the order $n, n-1, n-2, \ldots, 2$ and 1. Node n is an endpoint, so vector $\mathbf{p}^{(n)}$ has been already assigned. If in the process the next node i is an endpoint, then $\mathbf{p}^{(i)}$ is already given, otherwise we find the largest among the values $p_{K_i}^{(j)}$, $j \in J(i)$. Assume that the maximal value occurs at node j_i, then we assign $\mathbf{p}^{(i)} = \mathbf{p}^{(j_i)}$ to node i, and move to node $i-1$. After all payoff vectors $\mathbf{p}^{(n)}$, $\mathbf{p}^{(n-1)}$, $\ldots, \mathbf{p}^{(2)}$ and $\mathbf{p}^{(1)}$ are determined, then vector $\mathbf{p}^{(1)}$ gives the equilibrium payoffs and the equilibrium path is obtained by nodes:

$$1 \to i_1 = j_1 \to i_2 = j_{i_1} \to i_3 = j_{i_2} \to \ldots,$$

until an endpoint is reached, when the equilibrium path terminates.

At each node the number of comparisons equals the number of arcs starting at that node minus 1. Therefore the total number of comparisons in the algorithm is the total number of arcs minus the number of nodes.

This algorithm can be formally given as follows:

```
1  for $i \leftarrow n$ to 1
2      do $p_{K_i}^{(j_i)} \leftarrow \max\{p_{K_i}^{(l)}, l \in J(i)\}$
3      $\mathbf{p}^{(i)} \leftarrow \mathbf{p}^{(j_i)}$
4  print sequence 1, $i_1(= j_1)$, $i_2(= j_{i_1})$, $i_3(= j_{i_2})$, ...
   until an endpoint is reached
```

8.3. Example. Figure 8.3 shows a finite tree. In the circle at each nonterminal node we indicate the player assigned to that node. The payoff vectors are also shown at all terminal nodes. We have three players, so the payoff vectors have three elements.

First we number the nodes such that the beginning of each arc has a smaller number than its endpoint. We indicated these numbers in a box under each node. All nodes i for $i \geq 11$ are terminal nodes, as we start the backward induction with node 10. Since player $\mathcal{P}_3$ is assigned to this node we have to compare the third components of the payoff vectors $(2, 0, 0)$ and $(1, 0, 1)$ associated to the endpoints of the two arcs originating from node 10. Since $1 > 0$, player $\mathcal{P}_3$ will select the arc to node 22 as his best choice. Hence $j_{10} = 22$, and $\mathbf{p}^{(10)} = \mathbf{p}^{(22)} = (1, 0, 1)$. Then we check node 9. By comparing the third components of vectors $\mathbf{p}^{(19)}$ and $\mathbf{p}^{(20)}$ it is clear that player $\mathcal{P}_3$ will select node 20, so $j_9 = 20$, and $\mathbf{p}^{(9)} = \mathbf{p}^{(20)} = (4, 1, 4)$. In the graph we also indicated the choices of the players by thicker arcs. Continuing the procedure in the same way for nodes $8, 7, \ldots, 1$ we finally obtain the payoff vector $\mathbf{p}^{(1)} = (4, 1, 4)$ and equilibrium path $1 \to 4 \to 9 \to 20$.

Exercises

8.1-1 An entrepreneur (E) enters to a market, which is controlled by a chain store (C). Their competition is a two-person game. The strategies of the chain store are soft (S), when it allows the competitor to operate or tough (T), when it tries to drive out the competitor. The strategies of the entrepreneur are staying in (I) or leaving (L) the market. The payoff tables of the two player are assumed to be

	I	L
S	2	5
T	0	5

payoffs of C

	I	L
S	2	1
T	0	1

payoffs of E

Find the equilibrium.

8.1-2 A salesman sells an equipment to a buyer, which has 3 parts, under the following conditions. If all parts are good, then the customer pays $\$\alpha$ to the salesman, otherwise the salesman has to pay $\$\beta$ to the customer. Before selling the equipment, the salesman is able to check any one or more of the parts, but checking any one costs him $\$\gamma$. Consider a two-

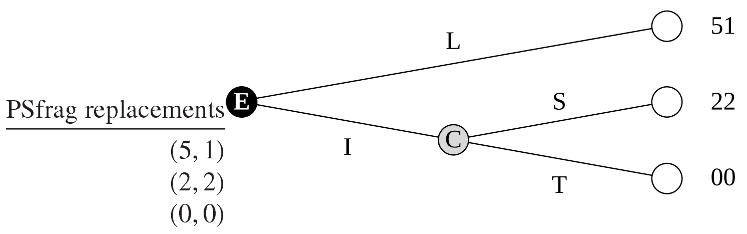


Figure 8.4. Tree for Exercise 8.1-5.

person game in which player $\mathcal{P}_1$ is the salesman with strategies $0, 1, 2, 3$ (how many parts he checks before selling the equipment), and player $\mathcal{P}_2$ is the equipment with strategies $0, 1, 2, 3$ (how many parts are defective). Show that the payoff matrix of player $\mathcal{P}_1$ is given as below when we assume that the different parts can be defective with equal probability.

		player $\mathcal{P}_2$			
		0	1	2	3
player $\mathcal{P}_1$	0	α	$-\beta$	$-\beta$	$-\beta$
	1	$\alpha-\gamma$	$-\frac{2}{3}\beta-\gamma$	$-\frac{1}{3}\beta-\gamma$	$-\gamma$
	2	$\alpha-2\gamma$	$-\frac{1}{3}\beta-\frac{5}{3}\gamma$	$-\frac{4}{3}\gamma$	$-\gamma$
	3	$\alpha-3\gamma$	-2γ	$-\frac{4}{3}\gamma$	$-\gamma$

8.1-3 Assume that in the previous problem the payoff of the second player is the negative of the payoff of the salesman. Give a complete description of the number of equilibria as a function of the parameter values α, β, γ. Determine the equilibria in all cases.
8.1-4 Assume that the payoff function of the equipment is its value (V if all parts are good, and zero otherwise) in the previous exercise. Is there an equilibrium point?
8.1-5 Exercise 8.1-1. can be represented by the tree shown in Figure 8.4.
Find the equilibrium with backward induction.
8.1-6 Show that in the one-player case backward induction reduces to the classical dynamic programming method.
8.1-7 Assume that in the tree of a game some nodes are so called “chance nodes" from which the game continuous with given probabilities assigned to the possible next nodes. Show the existence of the equilibrium for this more general case.
8.1-8 Consider the tree given in Figure 8.3, and double the payoff values of player $\mathcal{P}_1$, change the sign of the payoff values of player $\mathcal{P}_2$, and do not change those for Player $\mathcal{P}_3$. Find the equilibrium of this new game.

8.2. Continuous Games

If the strategy sets S_k are connected subsets of finite dimensional Euclidean Spaces and the payoff functions are continuous, then the game is considered continuous.

8.2.1. Fixed-Point Methods Based on Best Responses

It is very intuitive and usefull from algorithmic point of view to reformulate the equilibrium concept as follows. For all players $\mathcal{P}_k$ and $\mathbf{s} = (s_1, s_2, \ldots, s_N) \in S = S_1 \times S_2 \times \cdots \times S_N$ define the mapping:

$$B_k(\mathbf{s}) = \{s_k \in S_k | f_k(s_1, s_2, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots, s_N) \\ = \max_{t_k \in S_k} f_k(s_1, s_2, \ldots, s_{k-1}, t_k, s_{k+1}, \ldots, s_N)\}, \tag{8.3}$$

which is the set of the best choices of player $\mathcal{P}_k$ with given strategies $s_1, s_2, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$ of the other players. Note that $B_k(\mathbf{s})$ does not depend on s_k, it depends only on all other strategies $\mathbf{s}_l$, $k \neq l$. There is no guarantee that maximum exists for all $\mathbf{s} \in S_1 \times S_2 \times \cdots \times S_N$. Let $\sum \subseteq S$ be the subset of S such that $B_k(\mathbf{s})$ exists for all k and $\mathbf{s} \in \sum$. A simultaneous strategy vector $\mathbf{s}^\star = (s_1^\star, s_2^\star, \ldots, s_N^\star)$ is an equilibrium if and only if $\mathbf{s}^\star \in \sum$, and $s_k^\star \in B_k(\mathbf{s}^\star)$ for all k. By introducing the ***best reply mapping,*** $\mathbf{B}_k(\mathbf{s}) = (B_1(\mathbf{s}), \ldots, B_N(\mathbf{s}))$ we can further simplify the above reformulation:

Theorem 8.2 *Vector* $\mathbf{s}^\star$ *is equilibrium if and only if* $\mathbf{s}^\star \in \sum$ *and* $\mathbf{s}^\star \in \mathbf{B}(\mathbf{s}^\star)$.

Hence we have shown that the equilibrium-problem of N-person games is equivalent to find fixed points of certain point-to-set mappings.

The most frequently used existence theorems of equilibria are based on fixed point theorems such as the theorems of Brouwer, Kakutani, Banach, Tarski etc. Any algorithm for finding fixed points can be successfully applied for computing equilibria.

The most popular existence result is a straightforward application of the Kakutani-fixed point theorem.

Theorem 8.3 *Assume that in an N-person game*

1. *the strategy sets S_k are nonempty, closed, bounded, convex subsets of finite dimensional Euclidean spaces;*
   for all k,
2. *the payoff function f_k are continuous on S;*
3. *f_k is concave in s_k with all fixed $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$.*

Then there is at least one equilibrium.

8.4. Example. Consider a 2-person game, $N = 2$, with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions $f_1(s_1, s_2) = s_1 s_2 - 2s_1^2 + 5$, and $f_2(s_1, s_2) = s_1 s_2 - 2s_2^2 + s_2 + 3$. We will first find the best responses of both players. Both payoff functions are concave parabolas in their variables with vertices:

$$s_1 = \frac{s_2}{4} \quad \text{and} \quad s_2 = \frac{s_1 + 1}{4}.$$

For all $s_2 \in [0, 1]$ and $s_1 \in [0, 1]$ these values are clearly feasible strategies, so

$$B_1(\mathbf{s}) = \frac{s_2}{4} \quad \text{and} \quad B_2(\mathbf{s}) = \frac{s_1 + 1}{4}.$$

So $(s_1^\star, s_2^\star)$ is equilibrium if and only if it satisfies equations:

$$s_1^\star = \frac{s_2^\star}{4} \quad \text{and} \quad s_2^\star = \frac{s_1^\star + 1}{4}.$$

It is easy to see that the unique solution is:

$$s_1^\star = \frac{1}{15} \quad \text{and} \quad s_2^\star = \frac{4}{15},$$

which is therefore the unique equilibrium of the game.

8.5. Example. Consider a portion of a sea-channel, assume it is the unit interval $[0, 1]$. Player $\mathcal{P}_2$ is a submarine hiding in location $s_2 \in [0, 1]$, player $\mathcal{P}_1$ is an airplane dropping a bomb at certain location $s_1 \in [0, 1]$ resulting in a damage $\alpha e^{-\beta(s_1-s_2)^2}$ to the submarine. Hence a special two-person game is defined in which $S_1 = S_2 = [0, 1]$, $f_1(s_1, s_2) = \alpha e^{-\beta(s_1-s_2)^2}$ and $f_2(s_1, s_2) = -f_1(s_1, s_2)$. With fixed s_2, $f_1(s_1, s_2)$ is maximal if $s_1 = s_2$, therefore the best response of player $\mathcal{P}_1$ is $B_1(\mathbf{s}) = s_2$. Player $\mathcal{P}_2$ wants to minimize f_1 which occurs if $|s_1 - s_2|$ is as large as possible, which implies that

$$B_2(\mathbf{s}) = \begin{cases} 1, & \text{if } s_1 < 1/2, \\ 0, & \text{if } s_1 > 1/2, \\ \{0, 1\}, & \text{if } s_1 = 1/2. \end{cases}$$

Clearly, there is no $(s_1, s_2) \in [0, 1] \times [0, 1]$ such that $s_1 = B_1(\mathbf{s})$ and $s_2 \in B_2(\mathbf{s})$, consequently no equilibrium exists.

8.2.2. Applying Fan's Inequality

Define the ***aggregation function*** $H : S \times S \to \mathbb{R}$ as:

$$H_{\mathbf{r}}(\mathbf{s}, \mathbf{z}) = \sum_{k=1}^{N} r_k f_k(s_1, \ldots, s_{k-1}, z_k, s_{k+1}, \ldots, s_N) \tag{8.4}$$

for all $\mathbf{s} = (s_1, \ldots, s_N)$ and $\mathbf{z} = (z_1, \ldots, z_N)$ from S and some $\mathbf{r} = (r_1, r_2, \ldots, r_N) > \mathbf{0}$.

Theorem 8.4 *Vector $\mathbf{s}^\star \in S$ is an equilibrium if and only if*

$$H_{\mathbf{r}}(\mathbf{s}^\star, \mathbf{z}) \leq H_{\mathbf{r}}(\mathbf{s}^\star, \mathbf{s}^\star) \tag{8.5}$$

for all $\mathbf{z} \in S$.

Proof. Assume first that $\mathbf{s}^\star$ is an equilibrium, then inequality (8.1) holds for all k and $s_k \in S_k$. Adding the r_k-multiples of these relations for $k = 1, 2, \ldots, N$ we immediately have (8.5).

Assume next that (8.5) holds for all $\mathbf{z} \in S$. Select any k and $s_k \in S_k$, define $\mathbf{z} = (s_1^\star, \ldots, s_{k-1}^\star, s_k, s_{k+1}^\star, \ldots, s_N^\star)$, and apply inequality (8.5). All but the k-th terms cancel and the remaining term shows that inequality (8.1) holds. Hence $\mathbf{s}^\star$ is an equilibrium. ■

Introduce function $\phi(\mathbf{s}, \mathbf{z}) = H_{\mathbf{r}}(\mathbf{s}, \mathbf{z}) - H_{\mathbf{r}}(\mathbf{s}, \mathbf{s})$, then clearly $\mathbf{s}^\star \in S$ is an equilibrium if and only if

$$\phi(\mathbf{s}^\star, \mathbf{z}) \leq 0 \quad \text{for all} \quad \mathbf{z} \in S. \tag{8.6}$$

Relation (8.6) is known as ***Fan's inequality*** . It can be rewritten as a variational inequality (see section 8.2.9 later), or as a fixed point problem. We show here the second approach. For all $\mathbf{s} \in S$ define

$$\Phi(\mathbf{s}) = \{\mathbf{z} | \mathbf{z} \in S,\ \phi(\mathbf{s}, \mathbf{z}) = \max_{\mathbf{t} \in S} \phi(\mathbf{s}, \mathbf{t})\}. \tag{8.7}$$

Since $\phi(\mathbf{s}, \mathbf{s}) = 0$ for all $\mathbf{s} \in S$, relation (8.6) holds if and only if $\mathbf{s}^\star \in \Phi(\mathbf{s}^\star)$, that is $\mathbf{s}^\star$ is a fixed-point of mapping $\Phi : S \to 2^S$. Therefore any method to find fixed point is applicable for computing equilibria.

The computation cost depends on the type and size of the fixed point problem and also on the selected method.

8.6. Example. Consider again the problem of Example 8.4.. In this case

$$f_1(z_1, s_2) = z_1 s_2 - 2z_1^2 + 5,$$

$$f_2(s_1, z_2) = s_1 z_2 - 2z_2^2 + z_2 + 3,$$

so the aggregate function has the form with $r_1 = r_2 = 1$:

$$H_{\mathbf{r}}(\mathbf{s}, \mathbf{z}) = z_1 s_2 - 2z_1^2 + s_1 z_2 - 2z_2^2 + z_2 + 8.$$

Therefore

$$H_{\mathbf{r}}(\mathbf{s}, \mathbf{s}) = 2s_1 s_2 - 2s_1^2 - 2s_2^2 + s_2 + 8,$$

and

$$\phi(\mathbf{s}, \mathbf{z}) = z_1 s_2 - 2z_1^2 + s_1 z_2 - 2z_2^2 + z_2 - 2s_1 s_2 + 2s_1^2 + 2s_2^2 - s_2.$$

Notice that this function is strictly concave in z_1 and z_2, and is separable in these variables. At the stationary points:

$$\frac{\partial \phi}{\partial z_1} = s_2 - 4z_1 = 0$$

$$\frac{\partial \phi}{\partial z_2} = s_1 - 4z_2 + 1 = 0$$

implying that at the optimum

$$z_1 = \frac{s_2}{4} \quad \text{and} \quad z_2 = \frac{s_1 + 1}{4} ,$$

since both right hand sides are feasible. At the fixed point:

$$s_1 = \frac{s_2}{4} \quad \text{and} \quad s_2 = \frac{s_1 + 1}{4},$$

giving the unique solution:

$$s_1 = \frac{1}{15} \quad \text{and} \quad s_2 = \frac{4}{15}.$$

8.2.3. Solving the Kuhn–Tucker Conditions

Assume that for all k,

$$S_k = \{s_k | \mathbf{g}_k(s_k) \geq \mathbf{0}\},$$

where $\mathbf{g}_k : \mathbb{R}^{n_k} \to \mathbb{R}^{m_k}$ is a vector variable vector valued function which is continuously differentiable in an open set O_k containing S_k. Assume furthermore that for all k, the payoff function f_k is continuously differentiable in s_k on O_k with any fixed $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$.

If $\mathbf{s}^\star = (s_1^\star, \ldots, s_N^\star)$ is an equilibrium, then for all k, $s_k^\star$ is the optimal solution of problem:

$$\begin{array}{ll} \text{maximize} & f_k(s_1^\star, \ldots, s_{k-1}^\star, s_k, s_{k+1}^\star, \ldots, s_N^\star) \\ \text{sugject to} & \mathbf{g}_k(s_k) \geq \mathbf{0}. \end{array} \tag{8.8}$$

By assuming that at s_k the Kuhn–Tucker regularity condition is satisfied, the solution has to satisfy the Kuhn–Tucker necessary condition:

$$\begin{array}{rcl} \mathbf{u}_k & \geq & \mathbf{0} \\ \mathbf{g}_k(s_k) & \geq & \mathbf{0} \\ \nabla_k f_k(\mathbf{s}) + \mathbf{u}_k^T \nabla_k \mathbf{g}_k(s_k) & = & \mathbf{0}^T \\ \mathbf{u}_k^T \mathbf{g}_k(s_k) & = & 0, \end{array} \tag{8.9}$$

where $\mathbf{u}_k$ is an m_k-element column vector, $\mathbf{u}_k^T$ is its transpose, $\nabla_k f_k$ is the gradient of f_k (as a row vector) with respect to s_k and $\nabla_k \mathbf{g}_k$ is the Jacobian of function g_k.

Theorem 8.5 *If $\mathbf{s}^\star$ is an equilibrium, then there are vectors $\mathbf{u}_k^\star$ such that relations (8.9) are satisfied.*

Relations (8.9) for $k = 1, 2, \ldots, N$ give a (usually large) system of equations and inequalities for the unknowns s_k and $\mathbf{u}_k$ ($k = 1, 2, \ldots, N$). Any equilibrium (if exists) has to be among the solutions. If in addition for all k, all components of $\mathbf{g}_k$ are concave, and f_k is concave in s_k, then the Kuhn–Tucker conditions are also sufficient, and therefore all solutions of (8.9) are equilibria.

The computation cost in solving system (8.9) depends on its type and the chosen method.

8.7. Example. Consider again the two-person game of the previous example. Clearly,

$$S_1 = \{s_1 | s_1 \geq 0,\ 1 - s_1 \geq 0\},$$

$$S_2 = \{s_2 | s_2 \geq 0,\ 1 - s_2 \geq 0\},$$

so we have

$$\mathbf{g}_1(s_1) = \begin{pmatrix} s_1 \\ 1 - s_1 \end{pmatrix} \text{ and } \mathbf{g}_2(s_2) = \begin{pmatrix} s_2 \\ 1 - s_2 \end{pmatrix}.$$

Simple differentiation shows that

$$\nabla_1 \mathbf{g}_1(s_1) = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \quad \nabla_2 \mathbf{g}_2(s_2) = \begin{pmatrix} 1 \\ -1 \end{pmatrix},$$

$$\nabla_1 f_1(s_1, s_2) = s_2 - 4s_1, \quad \nabla_2 f_2(s_1, s_2) = s_1 - 4s_2 + 1,$$

therefore the Kuhn–Tucker conditions can be written as follows:

$$\begin{array}{rcl} u_1^{(1)}, u_2^{(1)} & \geq & 0 \\ s_1 & \geq & 0 \\ s_1 & \leq & 1 \\ s_2 - 4s_1 + u_1^{(1)} - u_2^{(1)} & = & 0 \\ u_1^{(1)} s_1 + u_2^{(1)}(1 - s_1) & = & 0 \\ u_1^{(2)}, u_2^{(2)} & \geq & 0 \\ s_2 & \geq & 0 \\ s_2 & \leq & 1 \\ s_1 - 4s_2 + 1 + u_1^{(2)} - u_2^{(2)} & = & 0 \\ u_1^{(2)} s_2 + u_2^{(2)}(1 - s_2) & = & 0. \end{array}$$

Notice that f_1 is concave in s_1, f_2 is concave in s_2, and all constraints are linear, therefore all solutions of this equality-inequality system are really equilibria. By systematically examining the combination of cases

$$s_1 = 0, \quad 0 < s_1 < 1, \quad s_1 = 1,$$

and

$$s_2 = 0, \quad 0 < s_2 < 1, \quad s_2 = 1,$$

it is easy to see that there is a unique solution

$$u_1^{(1)} = u_1^{(2)} = u_2^{(1)} = u_2^{(2)} = 0, \quad s_1 = \frac{1}{15}, \quad s_2 = \frac{4}{15}.$$

By introducing slack and surplus variables the Kuhn–Tucker conditions can be rewritten as a system of equations with some nonnegative variables. The nonnegativity conditions can be formally eliminated by considering them as squares of some new variables, so the result becomes a system of (usually) nonlinear equations without additional constraints. There is a large set of numerical methods for solving such systems.

8.2.4. Reduction to Optimization Problems

Assume that all conditions of the previous section hold. Consider the following optimization problem:

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{N} \mathbf{u}_k^T \mathbf{g}_k(s_k) \\ \text{subjective to} & \mathbf{u}_k \geq \mathbf{0} \\ & \mathbf{g}_k(s_k) \geq \mathbf{0} \\ & \nabla_k f_k(\mathbf{s}) + \mathbf{u}_k^T \nabla_k \mathbf{g}_k(s_k) = 0. \end{array} \tag{8.10}$$

The two first constraints imply that the objective function is nonnegative, so is the minimal value of it. Therefore system (8.9) has feasible solution if and only if the optimal value of the objective function of problem (8.10) is zero, and in this case any optimal solution satisfies relations (8.9).

Theorem 8.6 *The N-person game has equilibrium only if the optimal value of the objective function is zero. Then any equilibrium is optimal solution of problem* (8.10)*. If in addition all components of $\mathbf{g}_k$ are concave and f_k is concave in s_k for all k, then any optimal solution of problem* (8.10) *is equilibrium.*

Hence the equilibrium problem of the N-person game has been reduced to finding the optimal solutions of this (usually nonlinear) optimization problem. Any nonlinear programming method can be used to solve the problem.

The computation cost in solving the optimization problem (8.10) depends on its type and the chosen method. For example, if (8.10) is an LP, and solved by the simplex method, then the maximum number of operations is exponential. However in particular cases the procedure terminates with much less operations.

8.8. Example. In the case of the previous problem the optimization problem has the following form:

$$\begin{array}{ll} \text{minimize} & u_1^{(1)}s_1 + u_2^{(1)}(1-s_1) + u_1^{(2)}s_2 + u_2^{(2)}(1-s_2) \\ \text{subject to} & u_1^{(1)}, u_1^{(2)}, u_2^{(1)}, u_2^{(2)} \geq 0 \\ & s_1 \geq 0 \\ & s_1 \leq 1 \\ & s_2 \geq 0 \\ & s_2 \leq 1 \\ & s_2 - 4s_1 + u_1^{(1)} - u_2^{(1)} = 0 \\ & s_1 - 4s_2 + 1 + u_1^{(2)} - u_2^{(2)} = 0. \end{array}$$

Notice that the solution $u_1^{(1)} = u_1^{(2)} = u_2^{(1)} = u_2^{(2)} = 0$, $s_1 = 1/15$ and $s_2 = 4/15$ is feasible with zero objective function value, so it is also optimal. Hence it is a solution of system (8.9) and consequently an equilibrium.

Mixed Extension of Finite Games

We have seen earlier that a finite game does not necessary have equilibrium. Even if it does, in the case of repeating the game many times the players wish to introduce some randomness into their actions in order to make the other players confused and to seek an equilibrium in the stochastic sense. This idea can be modeled by introducing probability distributions as the strategies of the players and the expected payoff values as their new payoff functions.

Keeping the notation of section 8.1. assume that we have N players, the finite strategy set of player $\mathcal{P}_k$ is $S_k = \{s_k^{(1)}, \ldots, s_k^{(n_k)}\}$. In the mixed extension of this finite game each player selects a discrete probability distribution on its strategy set and in each realization of the game an element of S_k is chosen according to the selected distribution. Hence the new strategy set of player $\mathcal{P}_k$ is

$$\overline{S}_k = \{\mathbf{x}_k | \mathbf{x}_k = (x_k^{(1)}, \ldots, x_k^{(n_k)}),\ \sum_{i=1}^{n_k} x_k^{(i)} = 1,\ x_k^{(i)} \geq 0 \text{ for all } i\}, \tag{8.11}$$

which is the set of all n_k-element probability vectors. The new payoff function of this player is the expected value:

$$\overline{f}_k(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \ldots \sum_{i_N=1}^{n_N} f_k(s_1^{(i_1)}, s_2^{(i_2)}, \ldots, s_N^{(i_N)}) x_1^{(i_1)} x_2^{(i_2)} \ldots x_N^{(i_N)}. \tag{8.12}$$

Notice that the original "pure" strategies $s_k^{(i)}$ can be obtained by selecting $\mathbf{x}_k$ as the k-th basis vector. This is a continuous game and as a consequence of Theorem 8.3 it has at least one equilibrium. Hence if a finite game is without an equilibrium, its mixed extension has always at least one equilibrium, which can be obtained by using the methods outlined in the previous sections.

8.9. Example. Consider the two-person case in which $N = 2$, and as in section 8.1 introduce matrices $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ with (i, j) elements $f_1(s_1^{(i)}, s_2^{(j)})$ and $f_2(s_1^{(i)}, s_2^{(j)})$. In this special case

$$\overline{f}_k(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} a_{ij}^{(k)} x_i^{(1)} x_j^{(2)} = \mathbf{x}_1^T \mathbf{A}^{(k)} \mathbf{x}_2 . \tag{8.13}$$

The constraints of $\overline{S}_k$ can be rewritten as:

$$\begin{aligned} x_k^{(i)} &\geq 0 \qquad (i = 1, 2, \ldots, n_k), \\ -1 + \sum_{i=1}^{n_k} x_k^{(i)} &\geq 0, \\ 1 - \sum_{i=1}^{n_k} x_k^{(i)} &\geq 0. \end{aligned}$$

so we may select

$$\mathbf{g}_k(\mathbf{x}_k) = \begin{pmatrix} x_k^{(1)} \\ \vdots \\ x_k^{(n_k)} \\ \sum_{i=1}^{n_k} x_k^{(i)} - 1 \\ -\sum_{i=1}^{n_k} x_k^{(i)} + 1 \end{pmatrix}. \tag{8.14}$$

The optimization problem (8.10) now reduces to the following:

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{2} [\sum_{i=1}^{n_k} u_k^{(i)} x_k^{(i)} + u_k^{(n_k+1)} (\sum_{j=1}^{n_k} x_k^{(j)} - 1) + u_k^{(n_k+2)} (-\sum_{j=1}^{n_k} x_k^{(j)} + 1)] \\ \text{subject to} & u_k^{(i)} \geq 0 \quad (1 \leq i \leq n_k + 2) \\ & x_k^{(i)} \geq 0 \quad (1 \leq i \leq n_k) \\ & \mathbf{1}^T \mathbf{x}_k = 1 \\ & \mathbf{x}_2^T (\mathbf{A}^{(1)})^T + \mathbf{v}_1^T + (u_1^{(n_1+1)} - u_1^{(n_1+2)}) \mathbf{1}_1^T = \mathbf{0}_1^T \\ & \mathbf{x}_1^T (\mathbf{A}^{(2)}) + \mathbf{v}_2^T + (u_2^{(n_2+1)} - u_2^{(n_2+2)}) \mathbf{1}_2^T = \mathbf{0}_2^T, \end{array} \tag{8.15}$$

where $\mathbf{v}_k^T = (u_k^{(1)}, \ldots .u_k^{(n_k)})$, $\mathbf{1}_k^T = (1^{(1)}, \ldots, 1^{(n_k)})$ and $\mathbf{0}_k^T = (0^{(1)}, \ldots, 0^{(n_k)})$, $k = 1, 2$.

Notice this is a quadratic optimization problem. Computation cost depends on the selected method. Observe that the problem is usually nonconvex, so there is the possibility of stopping at a local optimum.

Bimatrix games

Mixed extensions of two-person finite games are called ***bimatrix games*** . They were already examined in Example 8.9.. For notational convenience introduce the simplifying notation:

$$\mathbf{A} = \mathbf{A}^{(1)}, \mathbf{B} = \mathbf{A}^{(2)}, \mathbf{x} = \mathbf{x}_1, \mathbf{y} = \mathbf{x}_2, m = n_1 \quad \text{and} \quad n = n_2.$$

We will show that problem (8.15) can be rewritten as quadratic programming problem with linear constraints.

Consider the objective function first. Let

$$\alpha = u_1^{(m+2)} - u_1^{(m+1)}, \quad \text{and} \quad \beta = u_2^{(n+2)} - u_2^{(n+1)},$$

then the objective function can be rewritten as follows:

$$\mathbf{v}_1^T \mathbf{x} + \mathbf{v}_2^T \mathbf{y} - \alpha(\mathbf{1}_m^T \mathbf{x} - 1) - \beta(\mathbf{1}_n^T \mathbf{y} - 1). \tag{8.16}$$

The last two constraints also simplify:

$$\begin{aligned} \mathbf{y}^T \mathbf{A}^T + \mathbf{v}_1^T - \alpha \mathbf{1}_m^T &= \mathbf{0}_m^T, \\ \mathbf{x}^T \mathbf{B} + \mathbf{v}_2^T - \beta \mathbf{1}_n^T &= \mathbf{0}_n^T, \end{aligned}$$

implying that

$$\mathbf{v}_1^T = \alpha \mathbf{1}_m^T - \mathbf{y}^T \mathbf{A}^T \quad \text{and} \quad \mathbf{v}_2^T = \beta \mathbf{1}_n^T - \mathbf{x}^T \mathbf{B}, \tag{8.17}$$

so we may rewrite the objective function again:

$$(\alpha \mathbf{1}_m^T - \mathbf{y}^T \mathbf{A}^T)\mathbf{x} + (\beta \mathbf{1}_n^T - \mathbf{x}^T \mathbf{B})\mathbf{y} - \alpha(\mathbf{1}_m^T \mathbf{x} - 1) - \beta(\mathbf{1}_n^T \mathbf{y} - 1) = \alpha + \beta - \mathbf{x}^T(\mathbf{A} + \mathbf{B})\mathbf{y},$$

since

$$\mathbf{1}_m^T \mathbf{x} = \mathbf{1}_n^T \mathbf{y} = 1.$$

Hence we have the following quadratic programming problem :

$$\begin{array}{ll} \text{maximize} & \mathbf{x}^T(\mathbf{A} + \mathbf{B})\mathbf{y} - \alpha - \beta \\ \text{subject to} & \mathbf{x} \ge \mathbf{0} \\ & \mathbf{y} \ge \mathbf{0} \\ & \mathbf{1}_m^T \mathbf{x} = 1 \\ & \mathbf{1}_n^T \mathbf{y} = 1 \\ & \mathbf{A}\mathbf{y} \le \alpha \mathbf{1}_m \\ & \mathbf{B}^T \mathbf{x} \le \beta \mathbf{1}_n, \end{array} \tag{8.18}$$

where the last two conditions are obtained from (8.17) and the nonnegativity of vectors $\mathbf{v}_1$, $\mathbf{v}_2$.

Theorem 8.7 *Vectors $\mathbf{x}^\star$ and $\mathbf{y}^\star$ are equilibria of the bimatrix game if and only if with some $\alpha^\star$ and $\beta^\star$, $(\mathbf{x}^\star, \mathbf{y}^\star, \alpha^\star, \beta^\star)$ is optimal solution of problem (8.18). The optimal value of the objective function is zero.*

This is a quadratic programming problem. Computation cost depends on the selected method. Since it is usually nonconvex, the algorithm might terminate at local optimum. We know that at the global optimum the objective function must be zero, which can be used for optimality check.

8.10. Example. Select

$$\mathbf{A} = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$$

and

$$\mathbf{B} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}.$$

Then

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} 3 & -2 \\ -2 & 3 \end{pmatrix},$$

so problem (8.18) has the form:

$$\begin{array}{ll} \text{maximize} & 3x_1y_1 - 2x_1y_2 - 2x_2y_1 + 3x_2y_2 - \alpha - \beta \\ \text{subject to} & x_1, x_2, y_1, y_2 \ge 0 \\ & x_1 + x_2 = 1 \\ & y_1 + y_2 = 1 \\ & 2y_1 - y_2 \le \alpha \\ & -y_1 + y_2 \le \alpha \\ & x_1 - x_2 \le \beta \\ & -x_1 + 2x_2 \le \beta, \end{array}$$

where $\mathbf{x} = (x_1, x_2)^T$ and $\mathbf{y} = (y_1, y_2)^T$. We also know from Theorem 8.7 that the optimal objective function value is zero, therefore any feasible solution with zero objective function value is necessarily optimal. It is easy to see that the solutions

$$\mathbf{x} = \begin{pmatrix}1\\0\end{pmatrix},\ \mathbf{y} = \begin{pmatrix}0\\1\end{pmatrix},\ \alpha = 2, \beta = 1,$$
$$\mathbf{x} = \begin{pmatrix}0\\1\end{pmatrix},\ \mathbf{y} = \begin{pmatrix}1\\0\end{pmatrix},\ \alpha = 1,\ \beta = 2,$$
$$\mathbf{x} = \begin{pmatrix}0.6\\0.4\end{pmatrix},\ \mathbf{y} = \begin{pmatrix}0.4\\0.6\end{pmatrix},\ \alpha = 0.2,\ \beta = 0.2$$

are all optimal, so they provide equilibria.

One might apply relations (8.9) to find equilibria by solving the equality-inequality system instead of solving an optimization problem. In the case of bimatrix games problem (8.9) simplifies as

$$\begin{array}{rcl}
\mathbf{x}^T\mathbf{A}\mathbf{y} & = & \alpha \\
\mathbf{x}^T\mathbf{B}\mathbf{y} & = & \beta \\
\mathbf{A}\mathbf{y} & \le & \alpha\mathbf{1}_m \\
\mathbf{B}^T\mathbf{x} & \le & \beta\mathbf{1}_n \\
\mathbf{x} & \ge & \mathbf{0}_m \\
\mathbf{y} & \ge & \mathbf{0}_n \\
\mathbf{1}_m^T\mathbf{x} = \mathbf{1}_n^T\mathbf{y} & = & 1,
\end{array} \tag{8.19}$$

which can be proved along the lines of the derivation of the quadratic optimization problem.

The computation cost of the solution of system (8.19) depends on the particular method being selected.

8.11. Example. Consider again the bimatrix game of the previous example. Substitute the first and second constraints $\alpha = \mathbf{x}^T\mathbf{A}\mathbf{y}$ and $\beta = \mathbf{x}^T\mathbf{B}\mathbf{y}$ into the third and fourth condition to have

$$\begin{array}{rcl}
2y_1 - y_2 & \le & 2x_1y_1 - x_1y_2 - x_2y_1 + x_2y_2 \\
-y_1 + y_2 & \le & 2x_1y_1 - x_1y_2 - x_2y_1 + x_2y_2 \\
x_1 - x_2 & \le & x_1y_1 - x_1y_2 - x_2y_1 + 2x_2y_2 \\
-x_1 + 2x_2 & \le & x_1y_1 - x_1y_2 - x_2y_1 + 2x_2y_2 \\
x_1, x_2, y_1, y_2 & \ge & 0 \\
x_1 + x_2 = y_1 + y_2 & = & 1.
\end{array}$$

It is easy to see that the solutions given in the previous example solve this system, so they are equilibria.

We can also rewrite the equilibrium problem of bimatrix games as an equality-inequality system with mixed variables. Assume first that all elements of $\mathbf{A}$ and $\mathbf{B}$ are between 0 and 1. This condition is not really restrictive, since by using linear transformations

$$\overline{\mathbf{A}} = a_1\mathbf{A} + b_1\mathbf{1} \quad \text{and} \quad \overline{\mathbf{B}} = a_2\mathbf{B} + b_2\mathbf{1},$$

where $a_1, a_2 > 0$, and $\mathbf{1}$ is the matrix all elements of which equal 1, the equilibria remain the same and all matrix elements can be transformed into interval $[0, 1]$.

Theorem 8.8 *Vectors* **x**, **y** *are an equilibrium if and only if there are real numbers* α, β *and zero-one vectors* **u**, *and* **v** *such that*

$$\begin{array}{ccccccc} 0 & \le & \alpha\mathbf{1}_m - \mathbf{A}\mathbf{y} & \le & \mathbf{1}_m - \mathbf{u} & \le & \mathbf{1}_m - \mathbf{x} \\ 0 & \le & \beta\mathbf{1}_n - \mathbf{B}^T\mathbf{x} & \le & \mathbf{1}_n - \mathbf{v} & \le & \mathbf{1}_n - \mathbf{y} \\ & & & & \mathbf{x} & \ge & \mathbf{0}_m \\ & & & & \mathbf{y} & \ge & \mathbf{0}_n \\ & & \mathbf{1}_m^T\mathbf{x} & = & \mathbf{1}_n^T\mathbf{y} & = & 1, \end{array} \tag{8.20}$$

where **1** denotes the vector with all unit elements.

Proof. Assume first that **x**, **y** is an equilibrium, then with some α and β, (8.19) is satisfied. Define

$$u_i = \begin{cases} 1, & \text{if } x_i > 0, \\ 0, & \text{if } x_i = 0, \end{cases} \quad \text{and} \quad v_j = \begin{cases} 1, & \text{if } y_j > 0, \\ 0, & \text{if } y_j = 0. \end{cases}$$

Since all elements of **A** and **B** are between 0 and 1, the values $\alpha = \mathbf{x}^T\mathbf{A}\mathbf{y}$ and $\beta = \mathbf{x}^T\mathbf{B}\mathbf{y}$ are also between 0 and 1. Notice that

$$0 = \mathbf{x}^T(\alpha\mathbf{1}_m - \mathbf{A}\mathbf{y}) = \mathbf{y}^T(\beta\mathbf{1}_n - \mathbf{B}^T\mathbf{x}),$$

which implies that (8.20) holds.

Assume next that (8.20) is satisfied. Then

$$\mathbf{0} \le \mathbf{x} \le \mathbf{u} \le \mathbf{1}_m \ \text{ and } \ \mathbf{0} \le \mathbf{y} \le \mathbf{v} \le \mathbf{1}_n.$$

If $u_i = 1$, then $\alpha - \mathbf{e}_i^T\mathbf{A}\mathbf{y} = 0$, (where $\mathbf{e}_i$ is the i-th basis vector), and if $u_i = 0$, then $x_i = 0$. Therefore

$$\mathbf{x}^T(\alpha\mathbf{1}_m - \mathbf{A}\mathbf{y}) = 0,$$

implying that $\alpha = \mathbf{x}^T\mathbf{A}\mathbf{y}$. We can similarly show that $\beta = \mathbf{x}^T\mathbf{B}\mathbf{y}$. Thus (8.19) is satisfied, so, **x**, **y** is an equilibrium. ■

The computation cost of the solution of system (8.20) depends on the particular method being seleced.

8.12. Example. In the case of the bimatrix game introduced earlier in Example 8.10. we have the following:

$$\begin{array}{ccccccc} 0 & \le & \alpha - 2y_1 + y_2 & \le & 1 - u_1 & \le & 1 - x_1 \\ 0 & \le & \alpha + y_1 - y_2 & \le & 1 - u_2 & \le & 1 - x_2 \\ 0 & \le & \beta - x_1 + x_2 & \le & 1 - v_1 & \le & 1 - y_1 \\ 0 & \le & \beta + x_1 - 2x_2 & \le & 1 - v_2 & \le & 1 - y_2 \\ & & x_1 + x_2 & = & y_1 + y_2 & = & 1 \\ & & & & x_1, x_2, y_1, y_2 & \ge & 0 \\ & & & & u_1, u_2, v_1, v_2 & \in & \{0, 1\}. \end{array}$$

Notice that all three solutions given in Example 8.10. satisfy these relations with

$$\mathbf{u} = (1, 0), \quad \mathbf{v} = (0, 1)$$

$$\mathbf{u} = (0, 1), \quad \mathbf{v} = (1, 0)$$

and

$$\mathbf{u} = (1, 1), \quad \mathbf{v} = (1, 1),$$

respectively.

Matrix games

In the special case of $\mathbf{B} = -\mathbf{A}$, bimatrix games are called ***matrix games*** and they are represented by matrix $\mathbf{A}$. Sometimes we refer to the game as matrix $\mathbf{A}$ game. Since $\mathbf{A} + \mathbf{B} = \mathbf{0}$, the quadratic optimization problem (8.18) becomes linear:

$$\begin{array}{ll} \text{minimize} & \alpha + \beta \\ \text{subject to} & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{y} \geq \mathbf{0} \\ & \mathbf{1}_m \mathbf{x} = 1 \\ & \mathbf{1}_n \mathbf{y} = 1 \\ & \mathbf{A}\mathbf{y} \leq \alpha \mathbf{1}_m \\ & \mathbf{A}^T \mathbf{x} \geq -\beta \mathbf{1}_n . \end{array} \tag{8.21}$$

From this formulation we see that the set of the equilibrium strategies is a convex polyhedron. Notice that variables $(\mathbf{x}, \beta)$ and $(\mathbf{y}, \alpha)$ can be separated, so we have the following result.

Theorem 8.9 *Vectors $\mathbf{x}^\star$ and $\mathbf{y}^\star$ give an equilibrium of the matrix game if and only if with some $\alpha^\star$ and $\beta^\star$, $(\mathbf{x}^\star, \beta^\star)$ and $(\mathbf{y}^\star, \alpha^\star)$ are optimal solutions of the linear programming problems:*

$$\begin{array}{llll} \textit{minimize} & \alpha & \textit{minimize} & \beta \\ \textit{subject to} & \mathbf{y} \geq \mathbf{0}_n & \textit{subject to} & \mathbf{x} \geq \mathbf{0}_m \\ & \mathbf{1}_n^T \mathbf{y} = 1 & & \mathbf{1}_m^T \mathbf{x} = 1 \\ & \mathbf{A}\mathbf{y} \leq \alpha \mathbf{1}_m & & \mathbf{A}^T \mathbf{x} \geq -\beta \mathbf{1}_n . \end{array} \tag{8.22}$$

Notice that at the optimum, $\alpha + \beta = 0$. The optimal α value is called the ***value of the matrix game*** .

Solving problem (8.22) requires exponential number of operations if the simplex method is chosen. With polynomial algorithm (such as the interior point method) the number of operations is only polynomial.

8.13. Example. Consider the matrix game with matrix:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 0 & 3 \\ -1 & 3 & 3 \end{pmatrix}.$$

In this case problems (8.22) have the from:

$$\begin{array}{llcll} \text{minimize} & \alpha & \text{and} & \text{minimize} & \beta \\ \text{subject to} & y_1, y_2, y_3 \geq 0 & & \text{subject to} & x_1, x_2, x_3 \geq 0 \\ & y_1 + y_2 + y_3 = 1 & & & x_1 + x_2 + x_3 = 1 \\ & 2y_1 + y_2 - \alpha \leq 0 & & & 2x_1 + 2x_2 - x_3 + \beta \geq 0 \\ & 2y_1 + 3y_3 - \alpha \leq 0 & & & x_1 + 3x_3 + \beta \geq 0 \\ & -y_1 + 3y_2 + 3y_3 - \alpha \leq 0 & & & 3x_2 + 3x_3 + \beta \geq 0. \end{array}$$

The application of the simplex method shows that the optimal solutions are $\alpha = 9/7$, $\mathbf{y} = (3/7, 3/7, 1/7)^T$, $\beta = -9/7$, and $\mathbf{x} = (4/7, 4/21, 5/21)^T$.

We can also obtain the equilibrium by finding feasible solutions of a certain set of linear constraints. Since at the optimum of problem (8.21), $\alpha + \beta = 0$, vectors $\mathbf{x}$, $\mathbf{y}$ and scalers α

and β are optimal solutions if and only if

$$\begin{array}{rcl} \mathbf{x}, \mathbf{y} & \geq & \mathbf{0} \\ \mathbf{1}_m^T \mathbf{x} & = & 1 \\ \mathbf{1}_n^T \mathbf{y} & = & 1 \\ \mathbf{A}\mathbf{y} & \leq & \alpha \mathbf{1}_m \\ \mathbf{A}^T \mathbf{x} & \geq & \alpha \mathbf{1}_n. \end{array} \tag{8.23}$$

The first phase of the simplex method has to be used to solve system (8.23), where the number of operations might be experimental. However in most practical examples much less operations are needed.

8.14. Example. Consider again the matrix game of the previous example. In this case system (8.23) has the following form:

$$\begin{array}{rcl} x_1, x_2, x_3, y_1, y_2, y_3 & \geq & 0 \\ x_1 + x_2 + x_3 = y_1 + y_2 + y_3 & = & 1 \\ 2y_1 + y_2 & \leq & \alpha \\ 2y_1 + 3y_3 & \leq & \alpha \\ -y_1 + 3y_2 + 3y_3 & \leq & \alpha \\ 2x_1 + 2x_2 - x_3 & \geq & \alpha \\ x_1 + 3x_3 & \geq & \alpha \\ 3x_2 + 3x_3 & \geq & \alpha. \end{array}$$

It is easy to see that $\alpha = 9/7$, $\mathbf{x} = (4/7, 4/21, 5/21)^T$, $\mathbf{y} = (3/7, 3/7, 1/7)^T$ satisfy these relations, so $\mathbf{x}$, $\mathbf{y}$ is an equilibrium.

8.2.5. Method of Fictitious Play

Consider now a matrix game with matrix $\mathbf{A}$. The main idea of this method is that at each step both players determine their best pure strategy choices against the average strategies of the other player of all previous steps. Formally the method can be described as follows.

Let $\mathbf{x}_1$ be the initial (mixed) strategy of player $\mathcal{P}_1$. Select $\mathbf{y}_1 = \mathbf{e}_{j_1}$ (the j_1st basis vector) such that

$$\mathbf{x}_1^T \mathbf{A} \mathbf{e}_{j_1} = \min_j \{\mathbf{x}_1^T \mathbf{A} \mathbf{e}_j\}. \tag{8.24}$$

In any further step $k \geq 2$, let

$$\overline{\mathbf{y}}_{k-1} = \frac{1}{k-1}((k-2)\overline{\mathbf{y}}_{k-2} + \mathbf{y}_{k-1}), \tag{8.25}$$

and select $\mathbf{x}_k = \mathbf{e}_{i_k}$ so that

$$\mathbf{e}_{i_k}^T \mathbf{A} \overline{\mathbf{y}}_{k-1} = \max_i \{\mathbf{e}_i^T \mathbf{A} \overline{\mathbf{y}}_{k-1}\}. \tag{8.26}$$

Let then

$$\overline{\mathbf{x}}_k = \frac{1}{k}((k-1)\overline{\mathbf{x}}_{k-1} + \mathbf{x}_k), \tag{8.27}$$

and select $\mathbf{y}_k = \mathbf{e}_{j_k}$ so that

$$\overline{\mathbf{x}}_k^T \mathbf{A}\mathbf{e}_{j_k} = \min_j \{\overline{\mathbf{x}}_k^T \mathbf{A}\mathbf{e}_j\}. \tag{8.28}$$

By repeating the general step for $k = 2, 3, \ldots$ two sequences are generated: $\{\overline{\mathbf{x}}_k\}$, and $\{\overline{\mathbf{y}}_k\}$. We have the following result:

Theorem 8.10 *Any cluster point of these sequences is an equilibrium of the matrix game. Since all $\overline{\mathbf{x}}_k$ and $\overline{\mathbf{y}}_k$ are probability vectors, they are bounded. Therefore there is at least one cluster point.*

Assume, matrix $\mathbf{A}$ is $m \times n$. In (8.24) we need mn multiplications. In (8.25) and (8.27) $m + n$ multiplications and divisions. In (8.26) and (8.28) mn multiplications. If we make L iteration steps, then the total number of multiplications and divisions is:

$$mn + L[2(m+n) + 2mn] = \ominus(Lmn).$$

The formal algorithm is as follows:

```
 1  $k \leftarrow 1$
 2  define $j_1$ such that $\mathbf{x}_1^T \mathbf{A}\mathbf{e}_{j_1} = \min_j\{\mathbf{x}_1^T \mathbf{A}\mathbf{e}_j\}$
 3  $\mathbf{y}_1 \leftarrow \mathbf{e}_{j_1}$
 4  $k \leftarrow k + 1$
 5  $\overline{\mathbf{y}}_{k-1} \leftarrow \frac{1}{k-1}((k-2)\overline{\mathbf{y}}_{k-2} + \mathbf{y}_{k-1})$
 6  define $i_k$ such that $\mathbf{e}_{i_k}^T \mathbf{A}\overline{\mathbf{y}}_{k-1} = \max_i\{\mathbf{e}_i^T \mathbf{A}\overline{\mathbf{y}}_{k-1}\}$
 7  $\mathbf{x}_k \leftarrow \mathbf{e}_{i_k}$
 8  $\overline{\mathbf{x}}_k \leftarrow \frac{1}{k}((k-1)\overline{\mathbf{x}}_{k-1} + \mathbf{x}_k)$
 9  define $j_k$ such that $\overline{\mathbf{x}}_k^T \mathbf{A}\mathbf{e}_{j_k} = \min_j\{\overline{\mathbf{x}}_k^T \mathbf{A}\mathbf{e}_j\}$
10  $\mathbf{y}_k \leftarrow \mathbf{e}_{j_k}$
11  if $\|\overline{\mathbf{x}}_k - \overline{\mathbf{x}}_{k-1}\| < \varepsilon$ and $\|\overline{\mathbf{y}}_{k-1} - \overline{\mathbf{x}}_{k-2}\| < \varepsilon$
12      then $(\overline{\mathbf{x}}_k, \overline{\mathbf{y}}_{k-1})$ is equilibrium
13  else go back to 4
```

Here $\varepsilon > 0$ is a user selected error tolerance.

8.15. Example. We applied the above method for the matrix game of the previous example and started the procedure with $\mathbf{x}_1 = (1, 0, 0)^T$. After 100 steps we obtained $\overline{\mathbf{x}}_{101} = (0.446, 0.287, 0.267)^T$ and $\overline{\mathbf{y}}_{101} = (0.386, 0.436, 0.178)^T$. Comparing it to the true values of the equilibrium strategies we see that the error is below 0.126, showing the very slow convergence of the method.

8.2.6. Symmetric Matrix Games

A matrix game with skew-symmetric matrix is called symmetric. In this case $\mathbf{A}^T = -\mathbf{A}$ and the two linear programming problems are identical. Therefore at the optimum $\alpha = \beta = 0$, and the equilibrium strategies of the two players are the same. Hence we have the following result:

Theorem 8.11 *A vector $\mathbf{x}^\star$ is equilibrium of the symmetric matrix game if and only if*

$$\begin{array}{rcl} \mathbf{x} & \geq & \mathbf{0} \\ \mathbf{1}^T\mathbf{x} & = & 1 \\ \mathbf{A}\mathbf{x} & \leq & \mathbf{0}. \end{array} \tag{8.29}$$

Solving system (8.29) the first phase of the simplex method is needed, the number of operations is exponential in the worst case but in practical case usually much less.

8.16. Example. Consider the symmetric matrix game with matrix $\mathbf{A} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$. In this case relations (8.29) simplify as follows:

$$\begin{array}{rcl} x_1, x_2 & \geq & 0 \\ x_1 + x_2 & = & 1 \\ x_2 & \leq & 0 \\ -x_1 & \leq & 0. \end{array}$$

Clearly the only solution is $x_1 = 1$ and $x_2 = 0$, that is the first pure strategy.

We will see in the next subsection that linear programming problems are equivalent to symmetric matrix games so any method for solving such games can be applied to solve linear programming problems, so they serve as alternative methodology to the simplex method. As we will see next, symmetry is not a strong assumption, since any matrix game is equivalent to a symmetric matrix game.

Consider therefore a matrix game with matrix $\mathbf{A}$, and construct the skew-symmetric matrix

$$\mathbf{P} = \begin{pmatrix} \mathbf{0}_{m\times m} & \mathbf{A} & -\mathbf{1}_m \\ -\mathbf{A}^T & \mathbf{0}_{n\times n} & \mathbf{1}_n \\ \mathbf{1}_m^T & -\mathbf{1}_n^T & 0 \end{pmatrix},$$

where all components of vector $\mathbf{1}$ equal 1. Matrix games $\mathbf{A}$ and $\mathbf{P}$ are equivalent in the following sense. Assume that $\mathbf{A} > \mathbf{0}$, which is not a restriction, since by adding the same constant to all element of $\mathbf{A}$ they become positive without changing equilibria.

Theorem 8.12

1. *If $\mathbf{z} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \lambda \end{pmatrix}$ is an equilibrium strategy of matrix game $\mathbf{P}$ then with $\mathbf{a} = (1-\lambda)/2$, $\mathbf{x} = (1/a)\mathbf{u}$ and $\mathbf{y} = (1/a)\mathbf{v}$ is an equilibrium of matrix game $\mathbf{A}$ with value $v = \lambda/a$;*
2. *If $\mathbf{x}, \mathbf{y}$ is an equilibrium of matrix game $\mathbf{A}$ and v is the value of the game, then*

$$\mathbf{z} = \frac{1}{2+v}\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ v \end{pmatrix}$$

is equilibrium strategy of matrix game $\mathbf{P}$.

Proof. Assume first that $\mathbf{z}$ is an equilibrium strategy of game $\mathbf{P}$, then $\mathbf{u} \geq \mathbf{0}$, $\mathbf{v} \geq \mathbf{0}$, $\mathbf{Pz} \leq \mathbf{0}$, so

$$\begin{array}{rcl} \mathbf{Av} - \lambda \mathbf{1}_m & \leq & \mathbf{0} \\ -\mathbf{A}^T\mathbf{u} + \lambda \mathbf{1}_n & \leq & \mathbf{0} \\ \mathbf{1}_m^T\mathbf{u} - \mathbf{1}_n^T\mathbf{v} & \leq & 0. \end{array} \tag{8.30}$$

First we show that $0 < \lambda < 1$, that is $a \neq 0$. If $\lambda = 1$, then (since $\mathbf{z}$ is a probability vector) $\mathbf{u} = \mathbf{0}$ and $\mathbf{v} = \mathbf{0}$, contradicting the second inequality of (8.30). If $\lambda = 0$, then $\mathbf{1}_m^T\mathbf{u} + \mathbf{1}_n^T\mathbf{v} = 1$, and by the third inequality of (8.30), $\mathbf{v}$ must have at least one positive component which makes the first inequality impossible.

Next we show that $\mathbf{1}^T\mathbf{u} = \mathbf{1}^T\mathbf{v}$. From (8.30) we have

$$\begin{array}{rcl} \mathbf{u}^T\mathbf{Av} - \lambda \mathbf{u}^T\mathbf{1}_m & \leq & 0, \\ -\mathbf{v}^T\mathbf{A}^T\mathbf{u} + \lambda \mathbf{u}^T\mathbf{1}_n & \leq & 0 \end{array}$$

and by adding these inequalities we see that

$$\mathbf{v}^T\mathbf{1}_n - \mathbf{u}^T\mathbf{1}_m \leq 0,$$

and combining this relation with the third inequality of (8.30) we see that $\mathbf{1}_m^T\mathbf{u} - \mathbf{1}_n^T\mathbf{v} = 0$.

Select $a = (1 - \lambda)/2 \neq 0$, then $\mathbf{1}_m^T\mathbf{u} = \mathbf{1}_n^T\mathbf{v} = a$, so both $\mathbf{x} = \mathbf{u}/a$, and $\mathbf{y} = \mathbf{v}/a$ are probability vectors, furthermore from (8.30),

$$\begin{array}{rcccl} \mathbf{A}^T\mathbf{x} & = & \frac{1}{a}\mathbf{A}^T\mathbf{u} & \geq & \frac{\lambda}{a}\mathbf{1}_n, \\ \mathbf{Ay} & = & \frac{1}{a}\mathbf{Av} & \leq & \frac{\lambda}{a}\mathbf{1}_m. \end{array}$$

So by selecting $\alpha = \lambda/a$ and $\beta = -\lambda/a$, $\mathbf{x}$ and $\mathbf{y}$ are feasible solutions of the pair (8.22) of linear programming problems with $\alpha + \beta = 0$, therefore $\mathbf{x}, \mathbf{y}$ is an equilibrium of matrix game $\mathbf{A}$. Part 2. can be proved in a similar way, the details are not given here. ■

8.2.7. Linear Programming and Matrix Games

In this section we will show that linear programming problems can be solved by finding the equilibrium strategies of symmetric matrix games and hence, any method for finding the equilibria of symmetric matrix games can be applied instead of the simplex method.

Consider the primal-dual linear programming problem pair:

$$\begin{array}{lllll} \text{maximize} & \mathbf{c}^T\mathbf{x} & \text{and} & \text{minimize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{x} \geq \mathbf{0} & & \text{subject to} & \mathbf{y} \geq \mathbf{0} \\ & \mathbf{Ax} \leq \mathbf{b} & & & \mathbf{A}^T\mathbf{y} \geq \mathbf{c}. \end{array} \tag{8.31}$$

Construct the skew-symmetric matrix:

$$\mathbf{P} = \begin{pmatrix} \mathbf{0} & \mathbf{A} & -\mathbf{b} \\ -\mathbf{A}^T & \mathbf{0} & \mathbf{c} \\ \mathbf{b}^T & -\mathbf{c}^T & 0 \end{pmatrix}.$$

Theorem 8.13 *Assume* $\mathbf{z} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \lambda \end{pmatrix}$ *is an equilibrium strategy of the symmetric matrix game*

P *with* $\lambda > 0$. *Then*

$$\mathbf{x} = \frac{1}{\lambda}\mathbf{v} \quad \text{and} \quad \mathbf{y} = \frac{1}{\lambda}\mathbf{u}$$

are optimal solutions of the primal and dual problems, respectively.

Proof. If $\mathbf{z}$ is an equilibrium strategy, then $\mathbf{Pz} \le \mathbf{0}$, that is,

$$\begin{aligned} \mathbf{Av} - \lambda\mathbf{b} &\le \mathbf{0} \\ -\mathbf{A}^T\mathbf{u} + \lambda\mathbf{c} &\le \mathbf{0} \\ \mathbf{b}^T\mathbf{u} - \mathbf{c}^T\mathbf{v} &\le 0. \end{aligned} \tag{8.32}$$

Since $\mathbf{z} \ge \mathbf{0}$ and $\lambda > 0$, both vectors $\mathbf{x} = (1/\lambda)\mathbf{v}$, and $\mathbf{y} = (1/\lambda)\mathbf{u}$ are nonnegative, and by dividing the first two relations of (8.32) by λ,

$$\mathbf{Ax} \le \mathbf{b} \quad \text{and} \quad \mathbf{A}^T\mathbf{y} \ge \mathbf{c},$$

showing that $\mathbf{x}$ and $\mathbf{y}$ are feasible for the primal and dual, respectively. From the last condition of (8.32) we have

$$\mathbf{b}^T\mathbf{y} \le \mathbf{c}^T\mathbf{x}.$$

However

$$\mathbf{b}^T\mathbf{y} \ge (\mathbf{x}^T\mathbf{A}^T)\mathbf{y} = \mathbf{x}^T(\mathbf{A}^T\mathbf{y}) \ge \mathbf{x}^T\mathbf{c} = \mathbf{c}^T\mathbf{x},$$

consequently, $\mathbf{b}^T\mathbf{y} = \mathbf{c}^T\mathbf{x}$, showing that the primal and dual objective functions are equal. The duality theorem implies the optimality of $\mathbf{x}$ and $\mathbf{y}$. ■

8.17. Example. Consider the linear programming problem:

$$\begin{aligned} \text{maximize} \quad & x_1 + 2x_2 \\ \text{subject to} \quad & x_1 \ge 0 \\ & -x_1 + x_2 \ge 1 \\ & 5x_1 + 7x_2 \le 25. \end{aligned}$$

First we have to rewrite the problem as a primal problem. Introduce the new variables:

$$x_2^+ = \begin{cases} x_2, & \text{if } x_2 \ge 0, \\ 0 & \text{otherwise}, \end{cases}$$

$$x_2^- = \begin{cases} -x_2, & \text{if } x_2 < 0, \\ 0 & \text{otherwise}. \end{cases}$$

and multiply the $\ge$-type constraint by -1. Then the problem becomes the following:

$$\begin{aligned} \text{maximize} \quad & x_1 + 2x_2^+ - 2x_2^- \\ \text{subject to} \quad & x_1, x_2^+, x_2^- \ge 0 \\ & x_1 - x_2^+ + x_2^- \le -1 \\ & 5x_1 + 7x_2^+ - 7x_2^- \le 25. \end{aligned}$$

Hence

$$\mathbf{A} = \begin{pmatrix} 1 & -1 & 1 \\ 5 & 7 & -7 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} -1 \\ 25 \end{pmatrix}, \quad \mathbf{c}^T = (1, 2, -2),$$

and so matrix $\mathbf{P}$ becomes:

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & \vdots & 1 & -1 & 1 & \vdots & 1 \\ 0 & 0 & \vdots & 5 & 7 & -7 & \vdots & -25 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -1 & -5 & \vdots & 0 & 0 & 0 & \vdots & 1 \\ 1 & -7 & \vdots & 0 & 0 & 0 & \vdots & 2 \\ -1 & 7 & \vdots & 0 & 0 & 0 & \vdots & -2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -1 & 25 & \vdots & -1 & -2 & 2 & \vdots & 0 \end{pmatrix} .$$

8.2.8. The Method of Von Neumann

The fictitious play method is an iteration algorithm in which at each step the players adjust their strategies based on the opponent's strategies. This method can therefore be considered as the realization of a discrete system where the strategy selections of the players are the state variables. For symmetric matrix games John von Neumann introduced a continuous systems approach when the players continuously adjust their strategies. This method can be applied to general matrix games, since–as we have seen earlier–any matrix game is equivalent to a symmetric matrix game. The method can also be used to solve linear programming problems as we have seen earlier that any primal-dual pair can be reduced to the solution of a symmetric matrix game.

Let now $\mathbf{P}$ be a skew-symmetric $n \times n$ matrix. The strategy of player $\mathcal{P}_2$, $\mathbf{y}(t)$ is considered as the function of time $t \geq 0$. Before formulating the dynamism of the system, introduce the following notation:

$$\begin{array}{llllll} u_i : & \mathbb{R}^n & \to & \mathbb{R}, & u_i(\mathbf{y}(t)) & = & \mathbf{e}_i^T \mathbf{P} \mathbf{y}(t) \quad (i = 1, 2, \ldots, n) , \\ \phi : & \mathbb{R} & \to & \mathbb{R}, & \phi(u_i) & = & \max\{0, u_i\} , \\ \Phi : & \mathbb{R}^n & \to & \mathbb{R}, & \Phi(\mathbf{y}(t)) & = & \sum_{i=1}^n \phi(u_i(\mathbf{y}(t))) . \end{array} \tag{8.33}$$

For arbitrary probability vector $\mathbf{y}_0$ solve the following nonlinear initial-value problem:

$$y_j'(t) = \phi(u_j(\mathbf{y}(t))) - \Phi(\mathbf{y}(t)) y_j(t), \quad y_j(0) = y_{j0} \quad (1 \leq j \leq n) . \tag{8.34}$$

Since the right-hand side is continuous, there is at least one solution. The right hand side of the equation can be interpreted as follows. Assume that $\phi(u_j(\mathbf{y}(t))) > 0$. If player $\mathcal{P}_2$ selects strategy $\mathbf{y}(t)$, then player $\mathcal{P}_1$ is able to obtain a positive payoff by choosing the pure strategy $\mathbf{e}_j$, which results in a negative payoff for player $\mathcal{P}_2$. However if player $\mathcal{P}_2$ increases $y_j(t)$ to one by choosing the same strategy $\mathbf{e}_j$ its payoff $\mathbf{e}_j^T \mathbf{P} \mathbf{e}_j$ becomes zero, so it increases. Hence it is the interest of player $\mathcal{P}_2$ to increase $y_j(t)$. This is exactly what the first term represents. The second term is needed to ensure that $\mathbf{y}(t)$ remains a probability vector for all $t \geq 0$.

The computation of the right hand side of equations (8.34) for all t requires $n^2 + n$ multiplications. The total computation cost depends on the length of solution interval, on the selected step size, and on the choice of the differential equation solver.

Theorem 8.14 *Assume that $t_1, t_2, \dots$ is a positive strictly increasing sequence converging to ∞, then any cluster point of the sequence $\{\mathbf{y}(t_k)\}$ is equilibrium strategy, furthermore there is a constant c such that*

$$\mathbf{e}_i^T \mathbf{P}\mathbf{y}(t_k) \le \frac{\sqrt{n}}{c + t_k} \quad (i = 1, 2, \dots, n)\ . \tag{8.35}$$

Proof. First we have to show that $\mathbf{y}(t)$ is a probability vector for all $t \ge 0$. Assume that with some j and $t_1 > 0$, $y_j(t_1) < 0$. Define

$$t_0 = \sup\{t | 0 < t < t_1, y_j(t) \ge 0\}\ .$$

Since $y_j(t)$ is continuous and $y_j(0) \ge 0$, clearly $y_j(t_0) = 0$, and for all $\tau \in (t_0, t_1)$, $y_j(\tau) < 0$. Then for all $\tau \in (t_0, t_1]$,

$$y_j'(\tau) = \phi(u_j(\mathbf{y}(\tau))) - \Phi(\mathbf{y}(\tau))y_j(\tau) \ge 0,$$

and the Lagrange mean-value theorem implies that with some $\tau \in (t_0, t_1)$,

$$y_j(t_1) = y_j(t_0) + y_j'(\tau)(t_1 - t_0) \ge 0\ ,$$

which is a contradiction. Hence $y_j(t)$ is nonnegative. Next we show that $\sum_{j=1}^n y_j(t) = 1$ for all t. Let $f(t) = 1 - \sum_{j=1}^n y_j(t)$, then

$$f'(t) = -\sum_{j=1}^n y_j'(t) = -\sum_{j=1}^n \phi(u_j(\mathbf{y}(t))) + \Phi(\mathbf{y}(t))(\sum_{j=1}^n y_j(t)) = -\Phi(\mathbf{y}(t))(1 - \sum_{j=1}^n y_j(t))\ ,$$

so $f(t)$ satisfies the homogeneous equation

$$f'(t) = -\Phi(\mathbf{y}(t))f(t)$$

with the initial condition $f(0) = 1 - \sum_{j=1}^n y_{j0} = 0$. Hence for all $t \ge 0$, $f(t) = 0$, showing that $\mathbf{y}(t)$ is a probability vector.

Assume that for some t, $\phi(u_i(\mathbf{y}(t))) > 0$. Then

$$\begin{aligned}\frac{d}{dt}\phi(u_i(\mathbf{y}(t))) &= \sum_{j=1}^n p_{ij} y_j'(t) = \sum_{j=1}^n p_{ij}[\phi(u_j(\mathbf{y}(t))) - \Phi(\mathbf{y}(t))y_j(t)] \\ &= \sum_{j=1}^n p_{ij}\phi(u_j(\mathbf{y}(t))) - \Phi(\mathbf{y}(t))\phi(u_i(\mathbf{y}(t))).\end{aligned} \tag{8.36}$$

By multiplying both sides by $\phi(u_i(\mathbf{y}(t)))$ and adding the resulted equations for $i = 1, 2, \dots, n$ we have:

$$\begin{aligned}\sum_{i=1}^n \phi(u_i(\mathbf{y}(t)))\frac{d}{dt}\phi(u_i(\mathbf{y}(t))) &= \sum_{i=1}^n \sum_{j=1}^n p_{ij}\phi(u_i(\mathbf{y}(t)))\phi(u_j(\mathbf{y}(t))) \\ &\quad -\Phi(\mathbf{y}(t))(\sum_{i=1}^n \phi^2(u_i(\mathbf{y}(t)))).\end{aligned} \tag{8.37}$$

The first term is zero, since $\mathbf{P}$ is skew-symmetric. Notice that this equation remains valid even as $\phi(u_i(\mathbf{y}(t))) = 0$ except the break-points (where the derivative of $\phi(u_i(\mathbf{y}(t)))$ does not exist) since (8.36) remains true.

Assume next that with a positive t, $\Phi(\mathbf{y}(t)) = 0$. Then for all i, $\phi(u_i(\mathbf{y}(t))) = 0$. Since equation (8.37) can be rewritten as

$$\frac{1}{2}\frac{d}{dt}\Psi(\mathbf{y}(t)) = -\Phi(\mathbf{y}(t))\Psi(\mathbf{y}(t)) \tag{8.38}$$

with

$$\Psi : \mathbb{R}^n \to \mathbb{R} \quad \text{and} \quad \Psi(\mathbf{y}(t)) = \sum_{i=1}^{n} \phi^2(u_i(\mathbf{y}(t))) \ ,$$

we see that $\Psi(\mathbf{y}(t))$ satisfies a homogeneous equation with zero initial solution at t, so the solution remains zero for all $\tau \geq t$. Therefore $\phi(u_i(\mathbf{y}(\tau))) = 0$ showing that $\mathbf{Py}(\tau) \leq \mathbf{0}$, that is, $\mathbf{y}(\tau)$ is equilibrium strategy.

If $\Phi(\mathbf{y}(\mathbf{t})) > 0$ for all $t \geq 0$, then $\Psi(\mathbf{y}(t)) > 0$, and clearly

$$\frac{1}{2}\frac{d}{dt}\Psi(\mathbf{y}(t)) \leq -\sqrt{\Psi(\mathbf{y}(t))}\Psi(\mathbf{y}(t)) \ ,$$

that is

$$\frac{1}{2}\frac{d}{dt}\Psi(\mathbf{y}(t))(\Psi(\mathbf{y}(t)))^{-\frac{3}{2}} \leq -1 \ .$$

Integrate both sides in interval $[0, t]$ to have

$$-\Psi(\mathbf{y}(t))^{-(1/2)} + c \leq -t \ ,$$

with $c = (\Psi(\mathbf{y}(0)))^{-(1/2)}$, which implies that

$$(\Psi(\mathbf{y}(t)))^{1/2} \leq \frac{1}{c+t} \ . \tag{8.39}$$

By using the Cauchy–Schwartz inequality we get

$$\mathbf{e}_i^T\mathbf{Py}(t) = u_i(\mathbf{y}(t)) \leq \phi(u_i(\mathbf{y}(t))) \leq \Phi(\mathbf{y}(t)) \leq \sqrt{n\Psi(\mathbf{y}(t))} \leq \frac{\sqrt{n}}{c+t} \ , \tag{8.40}$$

which is valid even at the break points because of the continuity of functions u_i. And finally, take a sequence $\{\mathbf{y}(t_k)\}$ with t_k increasingly converging to ∞. The sequence is bounded (being probability vectors), so there is at least one cluster point $\mathbf{y}^\star$. From (8.40), by letting $t_k \to \infty$ we have that $\mathbf{Py}^\star \leq \mathbf{0}$ showing that $\mathbf{y}^\star$ is an equilibrium strategy. ■

8.18. Example. Consider the matrix game with matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 0 & 3 \\ -1 & 3 & 3 \end{pmatrix} \ ,$$

which was the subject of our earlier Example 8.13. In order to apply the method of von Neumann we have to find first an equivalent symmetric matrix game. The application of the method given in Theorem 8.12. requires that the matrix has to be positive. Without changing the equilibria we can add

2 to all matrix elements to have

$$\mathbf{A}_{new} = \begin{pmatrix} 4 & 3 & 2 \\ 4 & 2 & 5 \\ 1 & 5 & 5 \end{pmatrix},$$

and by using the method we get the skew-symmetric matrix

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & \vdots & 4 & 3 & 2 & \vdots & -1 \\ 0 & 0 & 0 & \vdots & 4 & 2 & 5 & \vdots & -1 \\ 0 & 0 & 0 & \vdots & 1 & 5 & 5 & \vdots & -1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -4 & -4 & -1 & \vdots & 0 & 0 & 0 & \vdots & 1 \\ -3 & -2 & -5 & \vdots & 0 & 0 & 0 & \vdots & 1 \\ -2 & -5 & -5 & \vdots & 0 & 0 & 0 & \vdots & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \vdots & -1 & -1 & -1 & \vdots & 0 \end{pmatrix}.$$

The differential equations (8.34) were solved by using the 4th order Runge–Kutta method in the interval $[0, 100]$ with the step size $h = 0.01$ and initial vector $\mathbf{y}(0) = (1, 0, \ldots, 0)^T$. From $\mathbf{y}(100)$ we get the approximations

$$\mathbf{x} \approx (0.563619, 0.232359, 0.241988),$$

$$\mathbf{y} \approx (0.485258, 0.361633, 0.115144)$$

of the equilibrium strategies of the original game. Comparing these values to the exact values:

$$\mathbf{x} = \left(\frac{4}{7}, \frac{4}{21}, \frac{5}{21}\right) \quad \text{and} \quad \mathbf{y} = \left(\frac{3}{7}, \frac{3}{7}, \frac{1}{7}\right)$$

we see that the maximum error is about 0.067.

8.2.9. Diagonally Strictly Concave Games

Consider an N-person continuous game and assume that all conditions presented at the beginning of Section 8.2.3 are satisfied. In addition, assume that for all k, S_k is bounded, all components of g_k are concave and f_k is concave in s_k with any fixed $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$. Under these conditions there is at least one equilibrium (Theorem 8.3). The uniqueness of the equilibrium is not true in general, even if all f_k are strictly concave in s_k. Such an example is shown next.

8.19. Example. Consider a two-person game with $S_1 = S_2 = [0, 1]$ and $f_1(s_1, s_2) = f_2(s_1, s_2) = 1 - (s_1 - s_2)^2$. Clearly both payoff functions are strictly concave and there are infinitely many equilibria: $s_1^\star = s_2^\star \in [0, 1]$.

Select an arbitrary nonnegative vector $\mathbf{r} \in \mathbb{R}^N$ and define function

$$\mathbf{h} : \mathbb{R}^M \to \mathbb{R}^M, \quad \mathbf{h}(\mathbf{s}, \mathbf{r}) = \begin{pmatrix} r_1 \nabla_1 f_1(\mathbf{s})^T \\ r_2 \nabla_2 f_2(\mathbf{s})^T \\ \vdots \\ r_N \nabla_N f_N(\mathbf{s})^T \end{pmatrix}, \tag{8.41}$$

where $M = \sum_{k=1}^N n_k$, and $\nabla_k f_k$ is the gradient (as a row vector) of f_k with respect to s_k. The game is said to be ***diagonally strictly concave*** if for all $\mathbf{s}^{(1)} \neq \mathbf{s}^{(2)}$, $\mathbf{s}^{(1)}, \mathbf{s}^{(2)} \in S$ and for some $\mathbf{r} \geq \mathbf{0}$,

$$(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T (\mathbf{h}(\mathbf{s}^{(1)}, \mathbf{r}) - \mathbf{h}(\mathbf{s}^{(2)}, \mathbf{r})) < 0. \tag{8.42}$$

Theorem 8.15 *Under the above conditions the game has exactly one equilibrium.*

Proof. The existence of the equilibrium follows from Theorem 8.3. In proving uniqueness assume that $\mathbf{s}^{(1)}$ and $\mathbf{s}^{(2)}$ are both equilibria, and both satisfy relations (8.9). Therefore for $l = 1, 2$,

$$\begin{aligned} \mathbf{u}_k^{(l)^T} \mathbf{g}_k(s_k^{(l)}) &= 0 \\ \nabla_k f_k(\mathbf{s}^{(l)}) + \mathbf{u}_k^{(l)^T} \nabla_k \mathbf{g}_k(s_k^{(l)}) &= \mathbf{0}^T, \end{aligned}$$

and the second equation can be rewritten as

$$\nabla_k f_k(\mathbf{s}^{(l)}) + \sum_{j=1}^{m_k} u_{kj}^{(l)} \nabla_k g_{kj}(s_k^{(l)}) = 0 , \tag{8.43}$$

where $u_{kj}^{(l)}$ and g_{kj} are the jth components of $\mathbf{u}_k^{(l)}$ and g_k, respectively. Multiplying (8.43) by $(r_k(s_k^{(2)} - s_k^{(1)})^T)$ for $l = 1$ and by $r_k(s_k^{(1)} - s_k^{(2)})^T$ for $l = 2$ and adding the resulted equalities for $k = 1, 2, \ldots, N$ we have

$$\begin{aligned} 0 = \{(\mathbf{s}^{(2)} - \mathbf{s}^{(1)})^T \mathbf{h}(\mathbf{s}^{(1)}, \mathbf{r}) + (\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T \mathbf{h}(\mathbf{s}^{(2)}, \mathbf{r})\} \\ + \sum_{k=1}^N \sum_{j=1}^{m_k} r_k [u_{kj}^{(1)} (s_k^{(2)} - s_k^{(1)})^T \nabla_k g_{kj}(s_k^{(1)}) + u_{kj}^{(2)} (s_k^{(1)} - s_k^{(2)})^T \nabla_k g_{kj}(s_k^{(2)})] . \end{aligned} \tag{8.44}$$

Notice that the sum of the first two terms is positive by the diagonally strict concavity of the game, the concavity of the components of g_k implies that

$$(s_k^{(2)} - s_k^{(1)})^T \nabla_k g_{kj}(s_k^{(1)}) \geq g_{kj}(s_k^{(2)}) - g_{kj}(s_k^{(1)})$$

and

$$(s_k^{(1)} - s_k^{(2)})^T \nabla_k g_{kj}(s_k^{(2)}) \geq g_{kj}(s_k^{(1)}) - g_{kj}(s_k^{(2)}) .$$

Therefore from (8.44) we have

$$\begin{aligned} 0 > \sum_{k=1}^N \sum_{j=1}^{m_k} r_k [u_{kj}^{(1)} (g_{kj}(s_k^{(2)}) - g_{kj}(s_k^{(1)})) + u_{kj}^{(2)} (g_{kj}(s_k^{(1)}) - g_{kj}(s_k^{(2)}))] \\ = \sum_{k=1}^N \sum_{j=1}^{m_k} r_k [u_{kj}^{(1)} g_{kj}(s_k^{(2)}) + u_{kj}^{(2)} g_{kj}(s_k^{(1)})] \geq 0 , \end{aligned}$$

where we used the fact that for all k and l,

$$0 = \mathbf{u}_k^{(l)^T} \mathbf{g}_k(s_k^{(l)}) = \sum_{j=1}^{m_k} u_{kj}^{(l)} g_{kj}(s_k^{(l)}).$$

This is an obvious contradiction, which completes the proof. ■

Checking for Uniqueness of Equilibrium

In practical cases the following result is very useful in checking diagonally strict concavity of N-person games.

Theorem 8.16 *Assume S is convex, f_k is twice continuously differentiable for all k, and $\mathbf{J}(\mathbf{s},\mathbf{r}) + \mathbf{J}(\mathbf{s},\mathbf{r})^T$ is negative definite with some $\mathbf{r} \geq \mathbf{0}$, where $\mathbf{J}(\mathbf{s},\mathbf{r})$ is the Jacobian of $\mathbf{h}(\mathbf{s},\mathbf{r})$. Then the game is diagonally strictly concave.*

Proof. Let $\mathbf{s}^{(1)} \neq \mathbf{s}^{(2)}$, $\mathbf{s}^{(1)}, \mathbf{s}^{(2)} \in S$. Then for all $\alpha \in [0,1]$, $\mathbf{s}(\alpha) = \alpha\mathbf{s}^{(1)} + (1-\alpha)\mathbf{s}^{(2)} \in S$ and

$$\frac{d}{d\alpha}\mathbf{h}(\mathbf{s}(\alpha),\mathbf{r}) = \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})(\mathbf{s}^{(1)} - \mathbf{s}^{(2)}) \ .$$

Integrate both side in $[0,1]$ to have

$$\mathbf{h}(\mathbf{s}^{(1)},\mathbf{r}) - \mathbf{h}(\mathbf{s}^{(2)},\mathbf{r}) = \int_0^1 \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})d\alpha \ ,$$

and by premultiplying both sides by $(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T$ we see that

$$\begin{aligned}(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T(\mathbf{h}(\mathbf{s}^{(1)},\mathbf{r}) - \mathbf{h}(\mathbf{s}^{(2)},\mathbf{r})) &= \int_0^1 (\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})d\alpha \\ &= \frac{1}{2}\int_0^1 (\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T(\mathbf{J}(\mathbf{s}(\alpha),\mathbf{r}) + \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})^T)(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})d\alpha < 0 \ ,\end{aligned}$$

completing the proof. ■

8.20. Example. Consider a simple two-person game with strategy sets $S_1 = S_2 = [0,1]$, and payoff functions

$$f_1(s_1,s_2) = -s_1^2 + s_1 - s_1 s_2$$

and

$$f_2(s_1,s_2) = -s_2^2 + s_2 - s_1 s_2 \ .$$

Clearly all conditions, except diagonally strict concavity, are satisfied. We will use Theorem 8.16 to show this additional property. In this case

$$\nabla_1 f_1(s_1,s_2) = -2s_1 + 1 - s_2, \quad \nabla_2 f_2(s_1,s_2) = -2s_2 + 1 - s_1 \ ,$$

so

$$\mathbf{h}(\mathbf{s},\mathbf{r}) = \begin{pmatrix} r_1(-2s_1 + 1 - s_2) \\ r_2(-2s_2 + 1 - s_1 \end{pmatrix}$$

with Jacobian

$$\mathbf{J}(\mathbf{s},\mathbf{r}) = \begin{pmatrix} -2r_1 & -r_1 \\ -r_2 & -2r_2 \end{pmatrix}.$$

We will show that

$$\mathbf{J}(\mathbf{s},\mathbf{r}) + \mathbf{J}(\mathbf{s},\mathbf{r})^T = \begin{pmatrix} -4r_1 & -r_1 - r_2 \\ -r_1 - r_2 & -4r_2 \end{pmatrix}$$

is negative definite with some $\mathbf{r} \geq \mathbf{0}$. For example, select $r_1 = r_2 = 1$, then this matrix becomes

$$\begin{pmatrix} -4 & -2 \\ -2 & -4 \end{pmatrix}$$

with characteristic polynomial

$$\phi(\lambda) = \det\begin{pmatrix} -4-\lambda & -2 \\ -2 & -4-\lambda \end{pmatrix} = \lambda^2 + 8\lambda + 12\ ,$$

having negative eigenvalues $\lambda_1 = -2$, $\lambda_2 = -6$.

Iterative Computation of Equilibrium

We have see earlier in Theorem 8.4 that $\mathbf{s}^\star \in S$ is an equilibrium if and only if

$$\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star) \geq \mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}) \tag{8.45}$$

for all $\mathbf{s} \in S$, where $\mathbf{H}_\mathbf{r}$ is the aggregation function (8.4). In the following analysis we assume that the N-person game satisfies all conditions presented at the beginning of Section 8.2.9 and (8.42) holds with some positive $\mathbf{r}$.

We first show the equivalence of (8.45) and a variational inequality.

Theorem 8.17 *A vector* $\mathbf{s}^\star \in S$ *satisfies* (8.45) *if and only if*

$$\mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^\star) \leq 0 \tag{8.46}$$

for all $\mathbf{s} \in S$*, where* $\mathbf{h}(\mathbf{s},\mathbf{r})$ *is defined in* (8.41).

Proof. Assume $\mathbf{s}^\star$ satisfies (8.45). Then $\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s})$ as function of $\mathbf{s}$ obtains maximum at $\mathbf{s} = \mathbf{s}^\star$, therefore

$$\nabla_\mathbf{s}\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star)(\mathbf{s} - \mathbf{s}^\star) \leq 0$$

for all $\mathbf{s} \in S$, and since $\nabla_\mathbf{s}\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star)$ is $\mathbf{h}(\mathbf{s}^\star, \mathbf{r})$, we proved that $\mathbf{s}^\star$ satisfies (8.46).

Assume next that $\mathbf{s}^\star$ satisfies (8.46). By the concavity of $\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s})$ in $\mathbf{s}$ and the diagonally strict concavity of the game we have

$$\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star) - \mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}) \geq \mathbf{h}(\mathbf{s},\mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) \geq \mathbf{h}(\mathbf{s},\mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) + \mathbf{h}(\mathbf{s}^\star,\mathbf{r})^T(\mathbf{s} - \mathbf{s}^\star) > 0\ ,$$

so $\mathbf{s}^\star$ satisfies (8.45). ■

Hence any method available for solving variational inequalities can be used to find equilibria.

Next we construct a special two-person, game the equilibrium problem of which is equivalent to the equilibrium problem of the original N-person game.

Theorem 8.18 *Vector* $\mathbf{s}^\star \in S$ *satisfies* (8.45) *if and only if* $(\mathbf{s}^\star, \mathbf{s}^\star)$ *is an equilibrium of the two-person game* $D = \{2; S, S; f, -f\}$ *where* $f(\mathbf{s},\mathbf{z}) = \mathbf{h}(\mathbf{z},\mathbf{r})^T(\mathbf{s} - \mathbf{z})$.

Proof.

- Assume first that $\mathbf{s}^\star \in S$ satisfies (8.45). Then it satisfies (8.46) as well, so

  $$f(\mathbf{s}, \mathbf{s}^\star) \leq 0 = f(\mathbf{s}^\star, \mathbf{s}^\star).$$

  We need in addition to show that

  $$-f(\mathbf{s}^\star, \mathbf{s}) \leq 0 = -f(\mathbf{s}^\star, \mathbf{s}^\star).$$

  In contrary assume that with some $\mathbf{s}$, $f(\mathbf{s}^\star, \mathbf{s}) < 0$. Then

  $$\begin{aligned} 0 > f(\mathbf{s}^\star, \mathbf{s}) &= \mathbf{h}(\mathbf{s}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) > \mathbf{h}(\mathbf{s}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) + (\mathbf{s} - \mathbf{s}^\star)^T(\mathbf{h}(\mathbf{s}, \mathbf{r}) - \mathbf{h}(\mathbf{s}^\star, \mathbf{r})) \\ &= \mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) \geq 0, \end{aligned}$$

  where we used (8.42) and (8.46). This is a clear contradiction.
- Assume next that $(\mathbf{s}^\star, \mathbf{s}^\star)$ is an equilibrium of game D. Then for any $\mathbf{s}, \mathbf{z} \in S$,

  $$f(\mathbf{s}, \mathbf{s}^\star) \leq f(\mathbf{s}^\star, \mathbf{s}^\star) = 0 \leq f(\mathbf{s}^\star, \mathbf{z}).$$

  The first part can be rewritten as

  $$\mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^\star) \leq 0,$$

  showing that (8.46) is satisfied, so is (8.45).

■

Consider the following iteration procedure.
Let $\mathbf{s}^{(1)} \in S$ be arbitrary, and solve problem

$$\begin{array}{ll} \text{maximize} & f(\mathbf{s}, \mathbf{s}^{(1)}) \\ \text{subject to} & \mathbf{s} \in S. \end{array} \tag{8.47}$$

Let $\mathbf{s}^{(2)}$ denote an optimal solution and define $\mu_1 = f(\mathbf{s}^{(2)}, \mathbf{s}^{(1)})$. If $\mu_1 = 0$, then for all $\mathbf{s} \in S$,

$$f(\mathbf{s}, \mathbf{s}^{(1)}) = \mathbf{h}(\mathbf{s}^{(1)}, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^{(1)}) \leq 0,$$

so by Theorem 8.17, $\mathbf{s}^{(1)}$ is an equilibrium. Since $f(\mathbf{s}^{(1)}, \mathbf{s}^{(1)}) = 0$, we assume that $\mu_1 > 0$. In the general step $k \geq 2$ we have already k vectors $\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, \ldots, \mathbf{s}^{(k)}$, and $k-1$ scalers $\mu_1, \mu_2, \ldots, \mu_{k-1} > 0$. Then the next vector $\mathbf{s}^{(k+1)}$ and next scaler μ_k are the solutions of the following problem:

$$\begin{array}{lll} \text{maximize} & \mu & \\ \text{subject to} & f(\mathbf{s}, \mathbf{s}^{(i)}) \geq \mu & (i = 1, 2, \ldots, k) \\ & \mathbf{s} \in S. & \end{array} \tag{8.48}$$

Notice that

$$f(\mathbf{s}^{(k)}, \mathbf{s}^{(i)}) \geq \mu_{k-1} \geq 0 \qquad (i = 1, 2, \ldots, k-1)$$

and

$$f(\mathbf{s}^{(k)}, \mathbf{s}^{(k)}) = 0,$$

so we know that $\mu_k \geq 0$.

The formal algorithm is as follows:

```
1  $k \leftarrow 1$
2  solve problem (8.47), let $\mathbf{s}^{(2)}$ be optimal solution
3  **if** $f(\mathbf{s}^{(2)}, \mathbf{s}^{(1)}) = 0$
4      **then** $\mathbf{s}^{(1)}$ is equilibrium and stop
5  $k \leftarrow k + 1$
6  solve problem (8.48), let $\mathbf{s}^{(k+1)}$ be optimal solution
7  **if** $\|\mathbf{s}^{(k+1)} - \mathbf{s}^{(k)}\| < \varepsilon$
8      **then** $\mathbf{s}^{(k+1)}$ is equilibrium
9  **else** go to 4
```

Before stating the convergence theorem of the algorithm we notice that in the special case when the strategy sets are defined by linear inequalities (that is, all functions g_k are linear) then all constraints of problem (8.48) are linear, so at each iteration step we have to solve a linear programming problem.

In this linear case the simplex method has to be used in each iteration step with exponential computational cost, so the overall cost is also exponential (with prefixed number of steps).

Theorem 8.19 *There is a subsequence $\{\mathbf{s}^{(k_i)}\}$ of $\{\mathbf{s}^{(k)}\}$ generated by the method that converges to the unique equilibrium of the N-person game.*

Proof. The proof consists of several steps.

First we show that $\mu_k \to 0$ as $k \to \infty$. Since at each new iteration an additional constraint is added to (8.48), sequence $\{\mu_k\}$ is nonincreasing. Since it is also nonnegative, it must be convergent. Sequence $\{\mathbf{s}^{(k)}\}$ is bounded, since it is from the bounded set S, so it has a convergent subsequence $\{\mathbf{s}^{(k_i)}\}$. Notice that from (8.48) we have

$$0 \le \mu_{k_i - 1} = \min_{1 \le k \le k_i - 1} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T (\mathbf{s}^{(k_i)} - \mathbf{s}^{(k)}) \le \mathbf{h}(\mathbf{s}^{(k_{i-1})}, \mathbf{r})^T (\mathbf{s}^{(k_i)} - \mathbf{s}^{(k_{i-1})}) \ ,$$

where the right hand side tends to zero. Thus $\mu_{k_i - 1} \to 0$ and since the entire sequence $\{\mu_k\}$ is monotonic, the entire sequence converges to zero.

Let next $\mathbf{s}^\star$ be an equilibrium of the N-person game, and define

$$\delta(t) = \min\{(\mathbf{h}(\mathbf{s}, \mathbf{r}) - \mathbf{h}(\mathbf{z}, \mathbf{r}))^T (\mathbf{z} - \mathbf{s}) | \|\mathbf{s} - \mathbf{z}\| \ge t, \ \mathbf{z}, \mathbf{s} \in S\}. \tag{8.49}$$

By (8.42), $\delta(t) > 0$ for all $t > 0$. Define the indices k_i so that

$$\delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) = \min_{1 \le k \le i} \delta(\|\mathbf{s}^{(k)} - \mathbf{s}^\star\|) \quad (i = 1, 2, \ldots),$$

then for all $k = 1, 2, \ldots, i$,

$$\begin{aligned}
\delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) &\le (\mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r}) - \mathbf{h}(\mathbf{s}^\star, \mathbf{r}))^T (\mathbf{s}^\star - \mathbf{s}^{(k)}) \\
&= \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T (\mathbf{s}^\star - \mathbf{s}^{(k)}) - \mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T (\mathbf{s}^\star - \mathbf{s}^{(k)}) \\
&\le \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T (\mathbf{s}^\star - \mathbf{s}^{(k)}),
\end{aligned}$$

which implies that

$$\begin{aligned}\delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) &\leq \min_{1\leq k\leq i} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}^{(k)}) \\ &\leq \max_{\mathbf{s}\in S} \min_{1\leq k\leq i} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^{(k)}) \\ &= \min_{1\leq k\leq i} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s}^{(i+1)} - \mathbf{s}^{(k)}) \\ &= \mu_i\end{aligned}$$

where we used again problem (8.48). From this relation we conclude that $\delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) \to 0$ as $i \to \infty$. And finally, notice that function $\delta(t)$ satisfies the following properties:

1. $\delta(t)$ is continuous in t;
2. $\delta(t) > 0$ if $t > 0$ (as it was shown just below relation (8.49));
3. if for a convergent sequence $\{t^{(k)}\}$, $\delta(t^{(k)}) \to 0$, then necessarily $t^{(k)} \to 0$.

By applying property 3. with sequence $\{\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|\}$ it is clear that $\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\| \to 0$ so $\mathbf{s}^{(k_i)} \to \mathbf{s}^\star$. Thus the proof is complete. ■

Exercises

8.2-1 Consider a 2-person game with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions $f_1(x_1, x_2) = x_1^2 + x_1x_2 + 2$ and $f_2(x_1, x_2) = x_1 + x_2$. Show the existence of a unique equilibrium point by computing it. Show that Theorem 8.3. cannot be applied to prove existence.

8.2-2 Consider the "price war" game in which two firms are price setting. Assume that p_1 and p_2 are the strategies of the players, $p_1, p_2 \in [0, p_{max}]$ and the payoff functions are:

$$f_1(p_1, p_2) = \begin{cases} p_1, & \text{if } p_1 \leq p_2, \\ p_1 - c, & \text{if } p_1 > p_2, \end{cases}$$

$$f_2(p_1, p_2) = \begin{cases} p_2, & \text{if } p_2 \leq p_1, \\ p_2 - c, & \text{if } p_2 > p_1, \end{cases}$$

by assuming that $c < p_{max}$. Is there an equilibrium? How many equilibria were found?

8.2-3 A portion of the sea is modeled by the unit square in which a submarine is hiding. The strategy of the submarine is the hiding place $\mathbf{x} \in [0, 1] \times [0, 1]$. An airplane drops a bomb in a location $\mathbf{y} = [0, 1] \times [0, 1]$,j which is its strategy. The payoff of the airplane is the damage $\alpha e^{-\beta\|\mathbf{x}-\mathbf{y}\|}$ occurred by the bomb, and the payoff of the submarine is its negative. Does this 2-person game have an equilibrium?

8.2-4 In the second-price auction they sell one unit of an item to N bidders. They value the item as $v_1 < v_2 < \cdots < v_N$. Each of them offers a price for the item simultaneously without knowing the offers of the others. The bidder with the highest offer will get the item, but he has to pay only the second highest price. So the strategy of bidder k is $[0, \infty]$, so $x_k \in [0, \infty]$, and the payoff function for this bidder is:

$$f_k(x_1, x_2, \ldots, x_N) = \begin{cases} v_k - \max_{j\neq k} x_j, & \text{if } x_k = \max_j x_j, \\ 0 & \text{otherwise.} \end{cases}$$

What is the best response function of bidder k? Does this game have equilibrium?

8.2-5 Formulate Fan's inequality for the Problem 8.2-1.

8.2-6 Formulate and solve Fan's inequality for Problem 8.2-2.

8.2-7 Formulate and solve Fan's inequality for Problem 8.2-4.
8.2-8 Consider a 2-person game with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions

$$f_1(x_1, x_2) = -(x_1 - x_2)^2 + 2x_1 - x_2 + 1$$

$$f_2(x_1, x_2) = -(x_1 - 2x_2)^2 - 2x_1 + x_2 - 1.$$

Formulate Fan's inequality.
8.2-9 Let $n = 2$, $S_1 = S_2 = [0, 10]$, $f_1(x_1, x_2) = f_2(x_1, x_2) = 2x_1 + 2x_2 - (x_1 + x_2)^2$. Formulate the Kuhn–Tucker conditions to find the equilibrium. Solve the resulted system of inequalities and equations.
8.2-10 Consider a 3-person game with $S_1 = S_2 = S_3 = [0, 1]$, $f_1(x_1, x_2, x_3) = (x_1 - x_2)^2 + x_3$, $f_2(x_1, x_2, x_3) = (x_2 - x_3)^2 + x_1$ and $f_3(x_1, x_2, x_3) = (x_3 - x_1)^2 + x_2$. Formulate the Kuhn–Tucker condition.
8.2-11 Formulate and solve system (8.9) for Problem 8.2-8.
8.2-12 Repeat the previous problem for the game given in Problem 8.2-1.
8.2-13 Rewrite the Kuhn–Tucker conditions for Problem 8.2-8. into the optimization problem (8.10) and solve it.
8.2-14 Formulate the mixed extension of the finite game given in Problem 8.1-1.
8.2-15 Formulate and solve optimization problem (8.10) for the game obtained in the previous problem.
8.2-16 Formulate the mixed extension of the game introduced in Problem 8.2-3.
Formulate and solve the corresponding linear optimization problems (8.22) with $\alpha = 5$, $\beta = 3$, $\gamma = 1$.
8.2-17 Use fictitious play method for solving the matrix game of Problem 8.2-16.
8.2-18 Generalize the fictitious play method for bimatrix games.
8.2-19 Generalize the fictitious play method for the mixed extensions of finite n-person games.
8.2-20 Solve the bimatrix game with matrics $\mathbf{A} = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$ and $\mathbf{B} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}$ with the method you have developed in Problem 8.2-18.
8.2-21 Solve the symmetric matrix game $\mathbf{A} = \begin{pmatrix} 0 & 1 & 5 \\ -1 & 0 & -3 \\ -5 & 3 & 0 \end{pmatrix}$ by linear programming.
8.2-22 Repeat problem 8.2-21. with the method of fictitious play.
8.2-23 Develop the Kuhn–Tucker conditions (8.9) for the game given in Problem 8.2-21. above.
8.2-24★ Repeat Problems 8.2-21., 8.2-22. and 8.2-23. for the matrix game $\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \end{pmatrix}$. (First find the equivalent symmetric matrix game!).
8.2-25 Formulate the linear programming problem to solve the matrix game with matrix $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$.
8.2-26 Formulate a linear programming solver based on the method of fictitious play and

solve the LP problem:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & x_1, x_2 \geq 0 \\ & 3x_1 + x_2 \leq 4 \\ & x_1 + 3x_2 \leq 4. \end{aligned}$$

8.2-27 Solve the LP problem given in Example 8.17 by the method of fictitious play.
8.2-28 Solve problem 8.2-21. by the method of von Neumann.
8.2-29 Solve Problem 8.2-24. by the method of von Neumann.
8.2-30 Solve Problem 8.2-17. by the method of von Neumann.
8.2-31★ Check the solution obtained in the previous problems by verifying that all constraints of (8.21) are satisfied with zero objective function. *Hint.* What α and β should be selected?
8.2-32 Solve problem 8.2-26. by the method of von Neumann.
8.2-33 Let $N = 2$, $S_1 = S_2 = [0, 10]$, $f_1(x_1, x_2) = f_2(x_1, x_2) = 2x_1 + 2x_2 - (x_1 + x_2)^2$. Show that both payoff functions are strictly concave in x_1 and x_2 respectively. Prove that there are infinitely many equilibria, that is , the strict concavity of the payoff functions does not imply the uniqueness of the equilibrium.
8.2-34 Can matrix games be strictly diagonally concave?
8.2-35 Consider a two-person game with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions $f_1(x_1, x_2) = -2x_1^2 + x_1(1 - x_2)$, $f_2(x_1, x_2) = -3x_2^2 + x_2(1 - x_1)$. Show that this game satisfies all conditions of Theorem 8.16.
8.2-36 Solve the problem of the previous exercise by algorithm (8.47)–(8.48).

8.3. The Oligopoly Problem

The previous sections presented general methodology, however special methods are available for almost all special classes of games. In the following parts of this chapter a special game, the ***oligopoly game*** will be examined. It describes a real-life economic situation when N-firms produce a homogeneous good to a market, or offers the same service. This model is known as the ***classical Cournot model***. The firms are the players. The strategy of each player is its production level x_k with strategy set $S_k = [0, L_k]$, where L_k is its capacity limit. It is assumed that the market price depends on the total production level $s = x_1 + x_2 + \cdots + x_N$ offered to the market: $p(s)$, and the cost of each player depends on its own production level: $c_k(x_k)$. The profit of each firm is given as

$$f_k(x_1, \ldots, x_N) = x_k p\left(\sum_{l=1}^{N} x_l\right) - c_k(x_k). \tag{8.50}$$

In this way an N-person game $G = \{N; S_1, \ldots, S_N; f_1, \ldots, f_N\}$ is defined.

It is usually assumed that functions p and c_k $(k = 1, 2, \ldots, N)$ are twice continuously differentiable, furthermore

1. $p'(s) < 0$;

2. $p'(s) + x_k p''(s) \le 0$;
3. $p'(s) - c_k''(x_k) < 0$

for all k, $x_k \in [0, L_k]$ and $s \in [0, \sum_{l=1}^{N} L_l]$. Under assumptions 1.–3. the game satisfies all conditions of Theorem 8.3, so there is at least one equilibrium.

Best Reply Mappings

Notice that with the notation $s_k = \sum_{l \ne k} x_l$, the payoff function of player $\mathcal{P}_k$ can be rewritten as

$$x_k p(x_k + s_k) - c_k(x_k) \ . \tag{8.51}$$

Since S_k is a compact set and this function is strictly concave in x_k, with fixed s_k there is a unique profit maximizing production level of player $\mathcal{P}_k$, which is its best reply and is denoted by $B_k(s_k)$.

It is easy to see that there are three cases: $B_k(s_k) = 0$ if $p(s_k) - c_k'(0) \le 0$, $B_k(s_k) = L_k$ if $p(s_k + L_k) + L_k p'(s_k + L_k) - c_k'(L_k) \ge 0$, and otherwise $B_k(s_k)$ is the unique solution of the monotonic equation

$$p(s_k + x_k) + x_k p'(s_k + x_k) - c_k'(x_k) = 0.$$

Assume that $x_k \in (0, L_k)$. Then implicit differentiation with respect to s_k shows that

$$p'(1 + B_k') + B_k' p' + x_k p''(1 + B_k') - c_k'' B_k' = 0$$

showing that

$$B_k'(s_k) = -\frac{p' + x_k p''}{2p' + x_k p'' - c_k''}.$$

Notice that from assumptions 2. and 3.,

$$-1 < B_k'(s_k) \le 0 \ , \tag{8.52}$$

which is also true for the other two cases except for the break points.

As in Section 8.2.1. we can introduce the best reply mapping:

$$\mathbf{B}(x_1, \ldots, x_N) = \left(B_1 \left(\sum_{l \ne 1} x_l \right), \ldots, B_N \left(\sum_{l \ne N} x_l \right) \right) \tag{8.53}$$

and look for its fixed points. Another alternative is to introduce dynamic process which converges to the equilibrium.

Similarly to the method of fictitious play a discrete system can be developed in which each firm selects its best reply against the actions of the competitors chosen at the previous time period:

$$x_k(t+1) = B_k(\sum_{l \ne k} x_l(t)) \qquad (k = 1, 2, \ldots, N) \ . \tag{8.54}$$

Based on relation (8.52) we see that for $N = 2$ the right hand side mapping $\mathbb{R}^2 \to \mathbb{R}^2$ is a contraction, so it converges, however if $N > 2$, then no convergence can be established. Consider next a slight modification of this system: with some $K_k > 0$:

$$x_k(t+1) = x_k(t) + K_k(B_k(\sum_{l \ne k} x_l(t)) - x_k(t)) \tag{8.55}$$

for $k = 1, 2, \dots, N$. Clearly the steady-states of this system are the equilibria, and it can be proved that if K_k is sufficiently small, then sequences $x_k(0), x_k(1), x_k(2), \dots$ are all convergent to the equilibrium strategies.

Consider next the continuous counterpart of model (8.55), when (similarly to the method of von Neumann) continuous time scales are assumed:

$$\dot{x}_k(t) = K_k(B_k(\sum_{l \neq k} x_l(t)) - x_k(t)) \quad (k = 1, 2, \dots, N) \ . \tag{8.56}$$

The following result shows the convergence of this process.

Theorem 8.20 *Under assumptions 1–3, system (8.56) is asymptotically stable, that is, if the initial $x_k(0)$ values are selected close enough to the equilibrium, then as $t \to \infty$, $x_k(t)$ converges to the equilibrium strategy for all k.*

Proof. It is sufficient to show that the eigenvalues of the Jacobian of the system have negative real parts. Clearly the Jacobian is as follows:

$$\mathbf{J} = \begin{pmatrix} -K_1 & K_1 b_1 & \cdots & K_1 b_1 \\ K_2 b_2 & -K_2 & \cdots & K_2 b_2 \\ \vdots & \vdots & & \vdots \\ K_N b_N & K_N b_N & \cdots & -K_N \end{pmatrix} , \tag{8.57}$$

where $b_k = B'_k(\sum_{l \neq k} x_l)$ at the equilibrium. From (8.52) we know that $-1 < b_k \leq 0$ for all k. In order to compute the eigenvalues of $\mathbf{J}$ we will need a simple but very useful fact. Assume that $\mathbf{a}$ and $\mathbf{b}$ are N-element real vectors. Then

$$\det(\mathbf{I} + \mathbf{a}\mathbf{b}^T) = 1 + \mathbf{b}^T \mathbf{a} \ , \tag{8.58}$$

where $\mathbf{I}$ is the $N \times N$ identity matrix. This relation can be easily proved by using finite induction with respect to N. By using (8.58), the characteristic polynomial of $\mathbf{J}$ can be written as

$$\begin{aligned} \phi(\lambda) &= \det(\mathbf{J} - \lambda \mathbf{I}) = \det(\mathbf{D} + \mathbf{a}\mathbf{b}^T - \lambda \mathbf{I}) \\ &= \det(\mathbf{D} - \lambda \mathbf{I}) \det(\mathbf{I} + (\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{a}\mathbf{b}^T) \\ &= \det(\mathbf{D} - \lambda \mathbf{I})[1 + \mathbf{b}^T (\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{a}] \\ &= \Pi_{k=1}^N (-K_k(1 + b_k) - \lambda)[1 + \sum_{k=1}^N \frac{K_k b_k}{-K_k(1 + b_k) - \lambda}], \end{aligned}$$

where we used the notation

$$\mathbf{a} = \begin{pmatrix} K_1 b_1 \\ K_2 b_2 \\ \vdots \\ K_N b_N \end{pmatrix}, \quad \mathbf{b}^T = (1, 1, \dots, 1), \quad \mathbf{D} = \begin{pmatrix} -K_1(1 + b_1) & & \\ & \ddots & \\ & & -K_N(1 + b_N) \end{pmatrix} .$$

The roots of the first factor are all negative: $\lambda = -K_k(1 + b_k)$, and the other eigenvalues are the roots of equation

$$1 + \sum_{k=1}^{N} \frac{K_k b_k}{-K_k(1+b_k) - \lambda} = 0.$$

Notice that by adding the terms with identical denominators this equation becomes

$$1 + \sum_{l=1}^{m} \frac{\alpha_k}{\beta_k + \lambda} = 0 \tag{8.59}$$

with $\alpha_k, \beta_k > 0$, and the $\beta_k s$ are different. If $g(\lambda)$ denotes the left hand side then clearly the values $\lambda = -\beta_k$ are the poles,

$$\lim_{\lambda\to\pm\infty} g(\lambda) = 1, \quad \lim_{\lambda\to-\beta_k\pm 0} g(\lambda) = \pm\infty,$$

$$g'(\lambda) = \sum_{l=1}^{m} \frac{-\alpha_l}{(\beta_l + \lambda)^2} < 0,$$

so $g(\lambda)$ strictly decreases locally. The graph of the function is shown in Figure 8.3. Notice first that (8.59) is equivalent to a polynomial equation of degree m, so there are m real or complex roots. The properties of function $g(\lambda)$ indicate that there is one root below $-\beta_1$, and one root between each $-\beta_k$ and $-\beta_{k+1}$ $(k = 1, 2, \ldots, m-1)$. Therefore all roots are negative, which completes the proof. ■

The general discrete model (8.55) can be examined in the same way. If $K_k = 1$ for all k, then model (8.55) reduces to the simple dynamic process (8.54).

8.21. Example. Consider now a 3-person oligopoly with price function

$$p(s) = \begin{cases} 2 - 2s - s^2, & \text{if} \quad 0 \le s \le \sqrt{3} - 1 \ , \\ 0 & \text{otherwise} \ , \end{cases}$$

strategy sets $S_1 = S_2 = S_3 = [0, 1]$, and cost functions

$$c_k(x_k) = kx_k^3 + x_k \qquad (k = 1, 2, 3) \ .$$

The profit of firm k is therefore the following:

$$x_k(2 - 2s - s^2) - (kx_k^3 + x_k) = x_k(2 - 2x_k - 2s_k - x_k^2 - 2x_k s_k - s_k^2) - kx_k^3 - x_k \ .$$

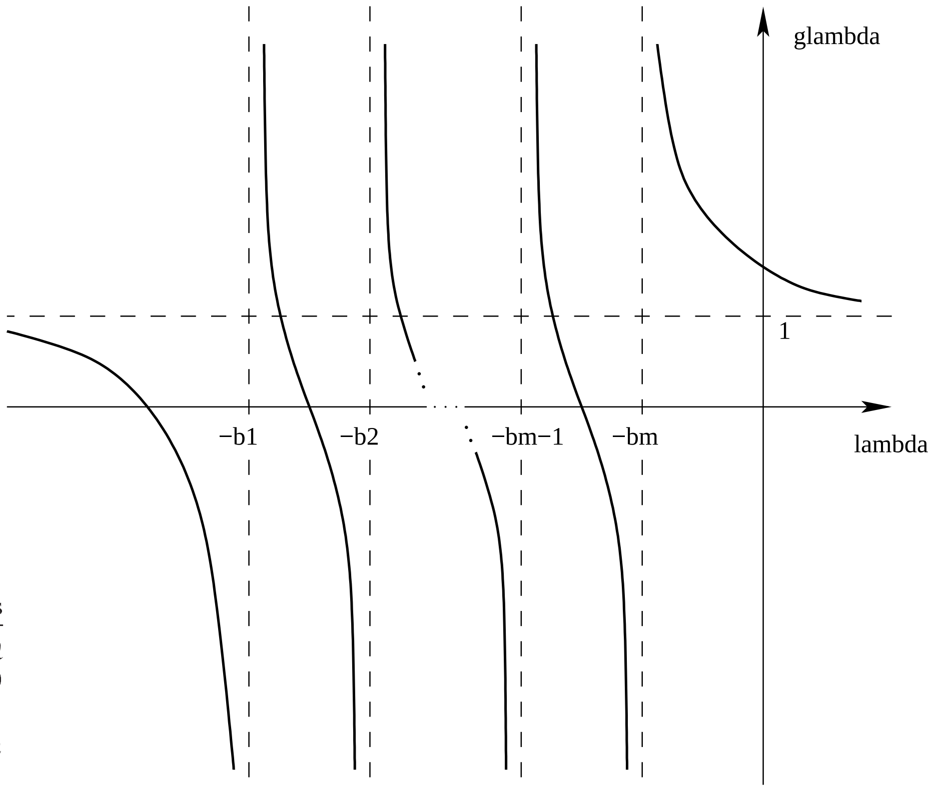
The best reply of play k can be obtained as follows. Following the method outlined at the beginning of Section 8.3. we have the following three cases. If $1 - 2s_k - s_k^2 \le 0$, then $x_k = 0$ is the best choice. If $(-6 - 3k) - 6s_k - s_k^2 \ge 0$, then $x_k = 1$ is the optimal decision. Otherwise x_k is the solution of equation

$$\begin{aligned} &\frac{\partial}{\partial x_k}[x_k(2 - 2x_k - 2s_k - s_k^2 - 2s_k x_k - x_k^2) - kx_k^3 - x_k] \\ &= 2 - 4x_k - 2s_k - s_k^2 - 4s_k x_k - 3x_k^2 - 3kx_k^2 - 1 = 0 \ , \end{aligned}$$

where the only positive solution is

$$x_k = \frac{-(4 + 4s_k) + \sqrt{(4 + 4s_k)^2 - 12(1 + k)(s_k^2 + 2s_k - 1)}}{6(1 + k)} \ .$$

After the best replies are found, we can easily construct any of the methods presented before.



PSfrag replacements
λ
$g(\lambda)$
$-\beta_1$
$-\beta_2$
$-\beta_{m-1}$
$-\beta_m$

Figure 8.5. The graph of function $g(\lambda)$

Reduction to Single-Dimensional Fixed Point Problems

Consider an N-firm oligopoly with price function p and cost functions c_k $(k = 1, 2, \dots, N)$. Introduce the following function

$$\Psi_k(s, x_k, t_k) = t_k p(s - x_k + t_k) - c_k(t_k) \ , \tag{8.60}$$

and define

$$X_k(s) = \{x_k | x_k \in S_k, \ \ \Psi_k(s, x_k, x_k) = \max_{t_k \in S_k} \Psi_k(s, x_k, t_k)\} \tag{8.61}$$

for $k = 1, 2, \dots, N$ and let

$$X(s) = \{u | u = \sum_{k=1}^{N} x_k, \ \ x_k \in X_k(s), \ \ k = 1, 2, \dots, N\}. \tag{8.62}$$

Notice that if $s \in [0, \sum_{k=1}^{N} L_k]$, then all elements of $X(s)$ are also in this interval, therefore X is a single-dimensional point-to-set mapping. Clearly $(x_1^\star, \dots, x_N^\star)$ is an equilibrium of the N-firm oligopoly game if and only if $s^\star = \sum_{k=1}^{N} x_k^\star$ is a fixed point of mapping X and for all

k, $x_k^\star \in X_k(s^\star)$. Hence the equilibrium problem has been reduced to find fixed points of only one-dimensional mappings. This is a significant reduction in the difficulty of the problem, since best replies are N-dimensional mappings.

If conditions 1–3 are satisfied, then $X_k(s)$ has exactly one element for all s and k:

$$X(s) = \begin{cases} 0, & \text{if } p(s) - c_k'(0) \le 0\ , \\ L_k, & \text{if } p(s) + L_k p_k'(s) - c_k'(L_k) \ge 0, \\ z^\star & \text{otherwise,} \end{cases} \tag{8.63}$$

where $z^\star$ is the unique solution of the monotonic equation

$$p(s) + zp'(s) - c_k'(z) = 0 \tag{8.64}$$

in the interval $(0, L_k)$. In the third case, the left hand side is positive at $z = 0$, negative at $z = L_k$, and by conditions 2–3, it is strictly decreasing, so there is a unique solution.

In the entire interval $[0, \sum_{k=1}^N L_k]$, $X_k(s)$ is nonincreasing. In the first two cases it is constant and in the third case strictly decreasing. Consider finally the single-dimensional equation

$$\sum_{k=1}^N X_k(s) - s = 0. \tag{8.65}$$

At $s = 0$ the left hand side is nonnegative, at $s = \sum_{k=1}^N L_k$ it is nonpositive, and is strictly decreasing. Therefore there is a unique solution (that is, fixed point of mapping X), which can be obtained by any method known to solve single-dimensional equations.

Let $[0, S_{max}]$ be the initial interval for the solution of equation (8.65). After K bisection steps the accuracy becomes $S_{max}/2^K$, which will be smaller than an error tolerance $\epsilon > 0$ if $K > \log_2(S_{max}/\epsilon)$.

```
1  solve equation (8.65) for $s$
2  for $k \leftarrow 1$ to $n$
3      do solve equation (8.64), and let $x_k \leftarrow z$
4  $(x_1, \ldots, x_N)$ is equilibrium
```

8.22. Example. Consider the 3-person oligopoly examined in the previous example. From (8.63) we have

$$X(s) = \begin{cases} 0, & \text{if} \quad 1 - 2s - s^2 \le 0, \\ 1, & \text{if} \quad -(1+3k) - 4s - s^2 \ge 0, \\ z^\star & \text{otherwise,} \end{cases}$$

where $z^\star$ is the unique solution of equation

$$3kz^2 + z(2s+2) + (-1 + 2s + s^2) = 0.$$

The first case occurs for $s \ge \sqrt{2} - 1$, the second case never occurs, and in the third case there is a unique positive solution:

$$z^\star = \frac{-(2s+2) + \sqrt{(2s+2)^2 - 12k(-1+2s+s^2)}}{6k}\ . \tag{8.66}$$

And finally equation (8.65) has the special form

$$\sum_{k=1}^{3} \frac{-(s+1)+\sqrt{(s+1)^2-3k(-1+2s+s^2)}}{3k} - s = 0\,.$$

A single program based on the bisection method gives the solution $s^\star \approx 0.2982$ and then equation (8.66) gives the equilibrium strategies $x_1^\star \approx 0.1077$, $x_2^\star \approx 0.0986$, $x_3^\star \approx 0.0919$.

Methods Based on Kuhn–Tucker Conditions

Notice first that in the case of N-player oligopolies $S_k = \{x_k | x_k \ge 0,\ L_k - x_k \ge 0\}$, so we select

$$\mathbf{g}_k(x_k) = \begin{pmatrix} x_k \\ L_k - x_k \end{pmatrix}, \tag{8.67}$$

and since the payoff functions are

$$f_k(x_1, \dots, x_N) = x_k p(x_k + s_k) - c_k(x_k)\,, \tag{8.68}$$

the Kuhn–Tucker conditions (8.9) have the following form. The components of the 2-dimensional vectors $\mathbf{u}_k$ will be denoted by $u_k^{(1)}$ and $u_k^{(2)}$. So we have for $k = 1, 2, \dots, N$,

$$\begin{array}{rcl} u_k^{(1)}, u_k^{(2)} & \ge & 0 \\ x_k & \ge & 0 \\ L_k - x_k & \ge & 0 \\ p(\sum_{l=1}^N x_l) + x_k p'(\sum_{l=1}^N x_l) - c_k'(x_k) + (u_k^{(1)}, u_k^{(2)})\binom{1}{-1} & = & 0 \\ u_k^{(1)} x_k + u_k^{(2)}(L_k - x_k) & = & 0. \end{array} \tag{8.69}$$

One might either look for feasible solutions of these relations or rewrite them as the optimization problem (8.10), which has the following special form in this case:

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^N (u_k^{(1)} x_k + u_k^{(2)}(L_k - x_k)) \\ \text{subject to} & u_k^{(1)}, u_k^{(2)} \ge 0 \\ & x_k \ge 0 \\ & L_k - x_k \ge 0 \\ & p(\sum_{l=1}^N x_l) + x_k p'(\sum_{l=1}^N x_l) - c_k'(x_k) + u_k^{(1)} - u_k^{(2)} = 0 \\ & \qquad\qquad (k = 1, 2, \dots, N). \end{array} \tag{8.70}$$

Computational cost in solving (8.69) or (8.70) depends on the type of functions p and c_k. No general characterization can be given.

8.23. Example. In the case of the three-person oligopoly introduced in Example 8.21. we have

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{3} (u_k^{(1)} x_k + u_k^{(2)}(1 - x_k)) \\ \text{subject to} & u_k^{(1)}, u_k^{(2)} \ge 0 \\ & x_k \ge 0 \\ & 1 - x_k \ge 0 \\ & 1 - 2s - s^2 - 2x_k - 2x_k s - 3k x_k^2 + u_k^{(1)} - u_k^{(2)} = 0 \\ & x_1 + x_2 + x_3 = s. \end{array}$$

A professional optimization software was used to obtain the optimal solutions:

$$x_1^\star \approx 0.1077, \quad x_2^\star \approx 0.0986, \quad x_3^\star \approx 0.0919 \ ,$$

and all $u_k^{(1)} = u_k^{(2)} = 0$.

Reduction to Complementarity Problems

If $(x_1^\star, \ldots, x_N^\star)$ is an equilibrium of an N-person oligopoly, then with fixed $x_1^\star, \ldots, x_{k-1}^\star, x_{k+1}^\star, \ldots, x_N^\star$, $x_k = x_k^\star$ maximizes the payoff f_k of player $\mathcal{P}_k$. Assuming that condition 1–3 are satisfied, f_k is concave in x_k, so $x_k^\star$ maximizes f_k if and only if at the equilibrium

$$\frac{\partial f_k}{\partial x_k}(\mathbf{x}^\star) = \begin{cases} \le 0, & \text{if} \quad x_k^\star = 0, \\ = 0, & \text{if} \quad 0 < x_k^\star < L_k, \\ \ge 0, & \text{if} \quad x_k^\star = L_k. \end{cases}$$

So introduce the slack variables

$$z_k = \begin{cases} = 0, & \text{if} \quad x_k > 0, \\ \ge 0, & \text{if} \quad x_k = 0 \end{cases}$$

$$v_k = \begin{cases} = 0, & \text{if} \quad x_k < L_k, \\ \ge 0, & \text{if} \quad x_k = L_k \end{cases}$$

and

$$w_k = L_k - x_k. \tag{8.71}$$

Then clearly at the equilibrium

$$\frac{\partial f_k}{\partial x_k}(\mathbf{x}) - v_k + z_k = 0 \tag{8.72}$$

and by the definition of the slack variables

$$z_k x_k = 0 \tag{8.73}$$

$$v_k w_k = 0, \tag{8.74}$$

and if we add the nonnegativity conditions

$$x_k, z_k, v_k, w_k \ge 0 \ , \tag{8.75}$$

then we obtain a system of nonlinear relations (8.71)–(8.75) which are equivalent to the equilibrium problem.

We can next show that relations (8.71)–(8.75) can be rewritten as a nonlinear complementarity problem, for the solution of which standard methods are available. For this purpose introduce the notation

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}, \quad \mathbf{L} = \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_N \end{pmatrix}, \quad \mathbf{h}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_2}(\mathbf{x}) \\ \vdots \\ \frac{\partial f_N}{\partial x_N}(\mathbf{x}) \end{pmatrix},$$

$$\mathbf{t} = \begin{pmatrix} \mathbf{x} \\ \mathbf{v} \end{pmatrix}, \quad \text{and} \quad \mathbf{g}(\mathbf{t}) = \begin{pmatrix} -\mathbf{h}(\mathbf{x}) + \mathbf{v} \\ \mathbf{L} - \mathbf{x} \end{pmatrix},$$

then system (8.72)–(8.75) can be rewritten as

$$\begin{aligned} \mathbf{t} &\geq \mathbf{0} \\ \mathbf{g}(\mathbf{t}) &\geq \mathbf{0} \\ \mathbf{t}^T\mathbf{g}(\mathbf{t}) &= 0. \end{aligned} \tag{8.76}$$

This problem is the usual formulation of ***nonlinear complementarity*** problems. Notice that the last condition requires that in each component either **t** or **g**(**t**) or both must be zero.

The computational cost in solving problem (8.76) depends on the type of the involved functions and the choice of method.

8.24. Example. In the case of the 3-person oligopoly introduced and examined in the previous examples we have:

$$\mathbf{t} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \quad \text{and} \quad \mathbf{g}(\mathbf{t}) = \begin{pmatrix} -1 + 2\sum_{l=1}^3 x_l + (\sum_{l=1}^3 x_l)^2 + 2x_1 + 2x_1\sum_{l=1}^3 x_l + 3x_1^2 + v_1 \\ -1 + 2\sum_{l=1}^3 x_l + (\sum_{l=1}^3 x_l)^2 + 2x_2 + 2x_2\sum_{l=1}^3 x_l + 6x_2^2 + v_2 \\ -1 + 2\sum_{l=1}^3 x_l + (\sum_{l=1}^3 x_l)^2 + 2x_3 + 2x_3\sum_{l=1}^3 x_l + 9x_3^2 + v_3 \\ 1 - x_1 \\ 1 - x_2 \\ 1 - x_3 \end{pmatrix}.$$

Linear Oligopolies and Quadratic Programming

In this section N-player oligopolies will be examined under the special condition that the price and all cost functions are linear :

$$p(s) = As + B, \quad c_k(x_k) = b_k x_k + c_k \quad (k = 1, 2, \ldots, N)\ ,$$

where B, b_k, and c_k are positive, but $A < 0$. Assume again that the strategy set of player $\mathcal{P}_k$ is the interval $[0, L_k]$. In this special case

$$f_k(x_1, \ldots, x_N) = x_k(Ax_1 + \cdots + Ax_N + B) - (b_k x_k + c_k) \tag{8.77}$$

for all k, therefore

$$\frac{\partial f_k}{\partial x_k}(\mathbf{x}) = 2Ax_k + A\sum_{l \neq k} x_l + B - b_k, \tag{8.78}$$

and relations (8.71)–(8.75) become more special:

$$\begin{aligned} 2Ax_k + A\sum_{l \neq k} x_l + B - b_k - v_k + z_k &= 0 \\ z_k x_k = v_k w_k &= 0 \\ x_k + w_k &= L_k \\ x_k, v_k, z_k, w_k &\geq 0, \end{aligned}$$

where we changed the order of them. Introduce the following vectors and matrixes:

$$\mathbf{Q} = \begin{pmatrix} 2A & A & \cdots & A \\ A & 2A & \cdots & A \\ \vdots & \vdots & & \vdots \\ A & A & \cdots & 2A \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B \\ B \\ \vdots \\ B \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix},$$

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{pmatrix}, \quad \mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix}, \quad \text{and} \quad \mathbf{L} = \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_N \end{pmatrix}.$$

Then the above relations can be summarized as:

$$\begin{aligned} \mathbf{Qx} + \mathbf{B} - \mathbf{b} - \mathbf{v} + \mathbf{z} &= \mathbf{0} \\ \mathbf{x} + \mathbf{w} &= \mathbf{L} \\ \mathbf{x}^T\mathbf{z} = \mathbf{v}^T\mathbf{w} &= 0 \\ \mathbf{x}, \mathbf{v}, \mathbf{z}, \mathbf{w} &\geq \mathbf{0}\ . \end{aligned} \tag{8.79}$$

Next we prove that matrix $\mathbf{Q}$ is negative definite. With any nonzero vector $\mathbf{a} = (a_i)$,

$$\mathbf{a}^T\mathbf{Qa} = 2A\sum_i a_i^2 + A\sum_i\sum_{j\neq i} a_i a_j = A(\sum_i a_i^2 + (\sum_i a_i)^2) < 0\ ,$$

which proves the assertion.

Observe that relations (8.79) are the Kuhn–Tucker conditions of the strictly concave quadratic programming problem:

$$\begin{aligned} &\text{maximize} && \tfrac{1}{2}\mathbf{x}^T\mathbf{Qx} + (\mathbf{B} - \mathbf{b})\mathbf{x} \\ &\text{subject to} && \mathbf{0} \leq \mathbf{x} \leq \mathbf{L}, \end{aligned} \tag{8.80}$$

and since the feasible set is a bounded linear polyhedron and the objective function is strictly concave, the Kuhn–Tucker conditions are sufficient and necessary. Consequently a vector $\mathbf{x}^\star$ is an equilibrium if and only if it is the unique optimal solution of problem (8.80). There are standard methods to solve problem (8.80) known from the literature.

Since (8.79) is a convex quadratic programming problem, several algorithms are available. Their costs are different, so computation cost depends on the particular method being selected.

8.25. Example. Consider now a duopoly (two-person oligopoly) where the price function is $p(s) = 10 - s$ and the cost functions are $c_1(x_1) = 4x_1 + 1$ and $c_2(x_2) = x_2 + 1$ with capacity limits $L_1 = L_2 = 5$. That is,

$$B = 10,\ \ A = -1,\ \ b_1 = 4,\ \ b_2 = 1,\ \ c_1 = c_2 = 1\ .$$

Therefore,

$$\mathbf{Q} = \begin{pmatrix} -2 & -1 \\ -1 & -2 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 10 \\ 10 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}, \quad \mathbf{L} = \begin{pmatrix} 5 \\ 5 \end{pmatrix},$$

so the quadratic programming problem can be written as:

$$\begin{aligned} &\text{maximize} && \frac{1}{2}(-2x_1^2 - 2x_1x_2 - 2x_2^2) + 6x_1 + 9x_2 \\ &\text{subject to} && 0 \leq x_1 \leq 5 \\ & && 0 \leq x_2 \leq 5. \end{aligned}$$

It is easy to see by simple differentiation that the global optimum at the objective function without the constraints is reached at $x_1^\star = 1$ and $x_2^\star = 4$. They however satisfy the constraints, so they are the optimal solutions. Hence they provide the unique equilibrium of the duopoly.

Exercises

8.3-1 Consider a duopoly with $S_1 = S_2 = [0, 1]$, $p(s) = 2 - s$ and costs $c_1(x) = c_2(x) = x^2 + 1$. Examine the convergence of the iteration scheme (8.55).
8.3-2 Select $n = 2$, $S_1 = S_2 = [0, 1.5]$, $c_k(x_k) = 0.5x_k$ $(k = 1, 2)$ and

$$p(s) = \begin{cases} 1.75 - 0.5s, & \text{if} \quad 0 \le s \le 1.5, \\ 2.5 - s, & \text{if} \quad 1.5 \le s \le 2.5, \\ 0, & \text{if} \quad 2.5 \le s. \end{cases}$$

Show that there are infinitely many equilibria:

$$\{(x_1^\star, x_2^\star) | 0.5 \le x_1 \le 1,\ \ 0.5 \le x_2 \le 1,\ \ x_1 + x_2 = 1.5\}.$$

8.3-3 Consider the duopoly of Problem 8.3-1. above. Find the best reply mappings of the players and determine the equilibrium.
8.3-4 Consider again the duopoly of the previous problem.
(a) Construct the one-dimensional fixed point problem of mapping (8.62) and solve it to obtain the equilibrium.
(b) Formulate the Kuhn–Tucker equations and inequalities (8.69).
(c) Formulate the complementarity problem (8.76) in this case.

Chapter notes

(Economic) Nobel Prize was given only once, in 1994 in the field of game theory. One of the winner was John Nash , who received this honor for his equilibrium concept, which was introduced in 1951 [42].

Backward induction is a more restrictive equilibrium concept. It was developed by Kuhn and can be found in [34]. Since it is more restrictive equilibrium, it is also a Nash equilibrium.

The existence and computation of equilibria can be reduced to those of fixed points. the different variants of fixed point theorems-such as that of Brouwer [8], Kakutani[29], Tarski [58] are successfully used to prove existence in many game classes. The article [44] uses the fixed point theorem of Kakutani. The books [57] and [21] discuss computer methods for computing fixed points. The most popular existence result is the well known theorem of Nikaido and Isoda [44].

The Fan inequality is discussed in the book of Aubin [3]. The Kuhn–Tucker conditions are presented in the book of Martos [40]. By introducing slack and surplus variables the Kuhn–Tucker conditions can be rewritten as a system of equations. For their computer solutions well known methods are available ([57] and [40]).

The reduction of bimatrix games to mixed optimization problems is presented in the papers of Mills [41] and Shapiro [52]. The reduction to quadratic programming problem is given in ([39]).

The method of fictitious play is discussed in the paper of Robinson [49]. In order to use the Neumann method we have to solve a system of nonlinear ordinary differential equations. The Runge–Kutta method is the most popular procedure for doing it. It can be found in [57].

The paper of Rosen [50] introduces diagonally strictly concave games. The computer method to find the equilibria of N-person concave games is introduced in Zuhovitsky et al. [64].

The different extensions and generalizations of the classical Cournot model can be found in the books of Okuguchi and Szidarovszky [45, 46]. The proof of Theorem 8.20 is given in [56]. For the proof of lemma (8.58) see the monograph [46]. The bisection method is described in [57]. The paper [30] contains methods which are applicable to solve nonlinear complementarity problems. The solution of problem (8.80) is discussed in the book of Hadley [23].

The book of von Neumann and Morgenstern [43] is considered the classical textbook of game theory. There is a large variety of game theory textbooks (see for example [21]).

9. Online Scheduling

In online computation, an algorithm must make its decisions based only on past events without secure information on future. Such algorithms are called ***on-line algorithms***. Online algorithms have many applications in different areas such as computer science, economics and operations research.

The first results in this area appeared around 1970, and later since 1990 more and more researchers started research on problems related to on-line algorithms. Many subfields were developed and investigated. Nowadays new results of the area are presented on the most important conferences about algorithms. It is not the goal of this chapter to give a detailed overview about the results, this would not be possible in this framework. The goal of the chapter is to show some of the main methods of analysing and developing on-line algorithms by presenting some subareas in more details.

In the next section we define the basic notions used in the analysis of on-line algorithms. After giving the most important definitions we present one of the most known on-line problems – the on-line k-server problem – and some of the related results. Then we deal with a new area we present on-line problems belonging to computer networks. In the next section the on-line bin packing problem and its multidimensional generalizations are presented. Finally in the last chapter of the section we show some of the basic results concerning the area of on-line scheduling.

9.1. Notions, definitions

Since an on-line algorithm makes its decisions by partial information without knowing the whole instance in advance we cannot expect that it gives the optimal solution which can be given by an algorithm having full information. An algorithms which knows the whole instance in advance is called ***offline algorithm***.

There are two main methods to measure the performance of on-line algorithms. One possibility is to use ***average case analysis*** where we hypothesize some distribution on events and we study the expected total cost.

The disadvantage of this approach is that usually we do not have any information about the distribution of the possible inputs. In this chapter we do not use the average case analysis.

An another approach is a worst case analysis, which is called ***competitive analysis***. In this case we compare the objective function value of the solution produced by the on-line algorithm to the optimal offline objective function value. Since we use this measure in this chapter we give the related definitions below.

In case of on-line minimization problems an on-line algorithm is called ***C-competitive***, if the cost of the solution produced by the on-line algorithm is at most C times more than the optimal offline cost for each input. The ***competitive ratio of an algorithm*** is the smallest such C for which the algorithm is C-competitive.

For an arbitrary on-line algorithm ALG we denote the objective function value achieved on input I by ALG(I). The optimal offline objective function value on I is denoted by OPT(I). Using this notation we can define the competitiveness as follows.

Algorithm ALG is C-competitive, if $\mathrm{ALG}(I) \le C \cdot \mathrm{OPT}(I)$ is valid for each input I.

Two further versions of the competitiveness are often used. For a minimization problem an algorithm ALG is called ***weakly C-competitive***, if there exists such a constant B that $\mathrm{ALG}(I) \le C \cdot \mathrm{OPT}(I) + B$ is valid for each input I.

The ***weak competitive ratio of an algorithm*** is the smallest such C for which the algorithm is weakly C-competitive.

A further version of the competitive ratio is the asymptotic competitive ratio. For minimization problems the ***asymptotic competitive ratio*** of algorithm ALG ($R^{\infty}_{\mathrm{ALG}}$) can be defined as follows:

$$R^{n}_{\mathrm{ALG}} = \sup\{\frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)} \mid \mathrm{OPT}(I) = n\} ,$$

$$R^{\infty}_{\mathrm{ALG}} = \limsup_{n\to\infty} R^{n}_{\mathrm{ALG}} .$$

An algorithm is called ***asymptotically C-competitive*** if its asymptotic competitive ratio is at most C.

The main property of the asymptotic ratio that it considers the performance of the algorithm under the assumption that the size of the input tends to ∞. This means that this ratio is not effected by the behaviour of the algorithm on the small size inputs.

We defined the basic notions of the competitive analysis for minimization problems. Similar definitions can be given for maximization problems. Then algorithm ALG is called C-competitive, if $\mathrm{ALG}(I) \ge C \cdot \mathrm{OPT}(I)$ is valid for each input I, and the algorithm is weakly C-competitive if there exists such a constant B that $\mathrm{ALG}(I) \ge C \cdot \mathrm{OPT}(I) + B$ is valid for each input I. The asymptotic ratio for maximization problems can be given as follows:

$$R^{n}_{\mathrm{ALG}} = \inf\{\frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)} \mid \mathrm{OPT}(I) = n\} ,$$

$$R^{\infty}_{\mathrm{ALG}} = \liminf_{n\to\infty} R^{n}_{\mathrm{ALG}} .$$

Then the algorithm is called ***asymptotically C-competitive*** if its asymptotic ratio is at least C.

Many scientific papers consider ***randomized on-line algorithms,*** in this case the objective function value achieved by the algorithm is a random variable and the expected value of this variable is used in the definition of the competitive ratio. Since we consider only deterministic on-line algorithms in this chapter we do not detail the notions related to randomized on-line algorithms.

9.2. The k-server problem

One of the most known online problems is the online k-server problem.To give the definition of the general problem we need the notion of the metric space. A pair (M, d) (where M contains the points of the space, d is the distance function defined on the set $M \times M$) is called metric space if the following properties are valid:

- $d(x, y) \geq 0$ for all $x, y \in M$,
- $d(x, y) = d(y, x)$ for all $x, y \in M$,
- $d(x, y) + d(y, z) \geq d(x, z)$ for all $x, y, z \in M$,
- $d(x, y) = 0$ holds if and only if $x = y$.

In the k-server problem a metric space is given and there are k servers which can move in the space. The decision maker has to satisfy a list of request appearing at the points of the metric space by sending a server to the point where the request appears.

The problem is online which means that the requests arrive one by one and we must satisfy each request without any information about the further requests. The goal is to minimize the total distance travelled by the servers. The model and its special versions have many applications. In the remaining parts of the section the multiset which contains the points where the servers are is called the ***configuration of the servers***. We use multisets since different servers can be at the same points of the space.

The first important results for the k-server problem were achieved by Manasse McGeoch and Sleator. They developed the following algorithm called BALANCE, which we denote by BAL. During the procedure the servers are in different points. The algorithm maintains for each server the total distance travelled by the server. We denote by $s_1, \ldots, s_k$ the servers and also the points in the space where the servers are located. Denote by $D_1, \ldots, D_k$ the total distance travelled by the servers. Then after the arrival at point P of a request algorithm BAL uses the server i for which the value $D_i + d(s_i, P)$ is minimal. This means that the algorithm tries to balance the distances travelled by the servers. Therefore the algorithm maintains the $S = \{s_1, \ldots, s_k\}$ server configuration and the distances travelled by the servers which distances have the starting values $D_1 = \cdots = D_k = 0$. Then the behaviour of the algorithm on input $I = P_1, \ldots, P_n$ can be given by the following pseudocode:

```
BAL($I$)

1 for $j \leftarrow 1$ to $n$
2     $i \leftarrow \text{argmin}\{D_i + d(s_i, P_j)\}$
3     serve the request with server $i$
4     $D_i \leftarrow D_i + d(s_i, P_j)$
5     $s_i \leftarrow P_j$
```

9.1. Example. Consider the two dimension Euclidean space as the metric space. The points are two dimensional real vectors (x, y), and the distance between (a, b) and (c, d) is $\sqrt{(a-c)^2 + (b-d)^2}$. Suppose that there are two servers which are located at points $(0, 0)$ and $(1, 1)$ at the beginning. Thus at the beginning $D_1 = D_2 = 0$, $s_1 = (0, 0)$, $s_2 = (1, 1)$. Suppose that the first request appears at point $(1, 4)$. Then $D_1 + d((0, 0), (1, 4)) = \sqrt{17} > D_2 + d((1, 1), (1, 4)) = 3$, thus the second server is used to satisfy the request and after the movement of the server $D_1 = 0, D_2 = 3$, $s_1 = (0, 0)$,

$s_2 = (1, 4)$. Suppose that the second request appears at point $(2, 4)$, then $D_1 + d((0, 0), (2, 4)) = \sqrt{20} > D_2 + d((1, 4), (2, 4)) = 3 + 1 = 4$, thus again the second server is used, and after serving the request $D_1 = 0, D_2 = 4$, $s_1 = (0, 0)$, $s_2 = (2, 4)$. Suppose that the third request appears at point $(1, 4)$, then $D_1 + d((0, 0), (1, 4)) = \sqrt{17} < D_2 + d((2, 4), (1, 4)) = 4 + 1 = 5$, thus the first server is used and after serving the request $D_1 = \sqrt{17}, D_2 = 4$, $s_1 = (1, 4)$, $s_2 = (2, 4)$.

The algorithm is efficient in the cases of some particular metric spaces as it is shown by the following statement. The references where the proof of the following theorem can be found are in the chapter notes at the end of the chapter.

Theorem 9.1 *Algorithm* BALANCE *is weakly k-competitive for the metric spaces containing $k + 1$ points.*

The following statement shows that there is no online algorithm which is better than k-competitive for the general k-server problem.

Theorem 9.2 *There is no such metric space containing at least $k + 1$ points where an on-line algorithm exists with smaller competitive ratio than k.*

Proof. Consider an arbitrary metric space containing at least $k+1$ points and an arbitrary on-line algorithm, denote the algorithm by ONL. Denote the points of the starting configuration of ONL by $P_1, P_2, \ldots, P_k$, and let P_{k+1} be an another point of the metric space. Consider the following long list of requests $I = Q_1, \ldots, Q_n$. The next request appears at the point among $P_1, P_2, \ldots, P_{k+1}$ where ONL has no server.

Calculate first the value ONL(I). The algorithm does not have server at point Q_{j+1} after serving Q_j, thus the request appeared at Q_j is served by the server located at point Q_{j+1}. Therefore the cost of serving Q_j is $d(Q_j, Q_{j+1})$, which yields that

$$\text{ONL}(I) = \sum_{j=1}^{n} d(Q_j, Q_{j+1}) \ ,$$

where Q_{n+1} denote the point from which the server sent to serve Q_n. (This is the point where the $(n + 1)$-th request would appear.) Now consider the cost OPT(I). Instead of calculating the optimal offline cost we define k different offline algorithms, and we use the average of the costs resulted by these algorithms. Since the cost of each offline algorithm is at least as much as the optimal offline cost thus the calculated average is an upper bound on the optimal offline cost.

We define the following k offline algorithms, denoted by $\text{OFF}_1, \ldots, \text{OFF}_k$. Suppose that the servers are in the points $P_1, P_2, \ldots, P_{j-1}, P_{j+1}, \ldots P_{k+1}$ in the starting configuration of OFF_j. We can move the servers into this starting configuration using an extra constant cost C_j.

The algorithms satisfy the requests as follows. If an algorithm OFF_j has a server at point Q_i, then none of the servers moves, otherwise the request is served by the server which is located at point Q_{i-1}. The algorithms are well-defined, if Q_i does not contain a server then each of the other points among $P_1, P_2, \ldots, P_{k+1}$ contains a servers, thus there is a server located at Q_{i-1}. Moreover $Q_1 = P_{k+1}$, thus at the beginning each algorithm has a server at the requested point.

We show that the servers of algorithms $\mathrm{OFF}_1, \dots, \mathrm{OFF}_k$ are always in different configurations. At the beginning this property is valid by the definition of the algorithms. Consider now the step where a request is served. Call the algorithms stable which do not move a server for serving the request, and the other algorithms unstable. The server configurations of the stable algorithms remain unchanged, so these configurations remain different from each other. Any unstable algorithm moves a server from the point Q_{i-1}. This point is the place of the last request thus the stable algorithms have server in it. Therefore, an unstable algorithm and a stable algorithm cannot have the same configuration after serving the request. Furthermore, any of the unstable algorithms moves a server from Q_{i-1} to Q_i, thus the server configurations of the unstable algorithms remain different from each other.

So at the arrival of the request at point Q_i the servers of the algorithms are in different configuration. On the other hand each configuration has a server at point Q_{i-1}, therefore there is only one configuration where no server located at point Q_i. Consequently the cost of serving Q_i is $d(Q_{i-1}, Q_i)$ for one of the algorithms and 0 for the other algorithms.

Therefore

$$\sum_{j=1}^{k} \mathrm{OFF}_j(I) = C + \sum_{i=2}^{n} d(Q_i, Q_{i-1}) \ ,$$

where $C = \sum_{j=1}^{k} C_j$ is an absolute constant which is independent from the input (this is the cost of moving the servers to the starting configuration of the defined algorithms).

On the other hand the optimal offline cost cannot be larger than the cost of any of the above defined algorithms, thus $k \cdot \mathrm{OPT}(I) \leq \sum_{j=1}^{k} \mathrm{OFF}_j(I)$. This yields that

$$k \cdot \mathrm{OPT}(I) \leq C + \sum_{i=2}^{n} d(Q_i, Q_{i-1}) \leq C + \mathrm{ONL}(I),$$

which inequality shows that the weak competitive ratio of ONL cannot be smaller than k, since the value $\mathrm{OPT}(I)$ can be arbitrarily large as the length of the input is increasing. ∎

Many researchers started to work on the problem and some interesting results appeared during the next few years. For the general case the first constant-competitive algorithm ($O(2^k)$-competitive) was developed by Fiat, Rabani and Ravid. Then the researchers could not significantly decrease the gap between the lower and upper bounds for a long time. Later Koutsoupias and Papadimitriou could analyze an algorithm based on the work function technique and they could prove that the algorithm is $(2k-1)$-competitive. They could not determine the competitive ratio of the algorithm, but it is a widely believed hypothesis that the algorithm is k-competitive. To determine the competitive ratio of the algorithm, or to develop a k-competitive algorithm are still among the most important open questions in the area of on-line algorithms. We present the work function algorithm below.

Denote by A_0 the starting configuration of the on-line servers. Then after the t-th request the ***work function*** value belonging to the multiset X is the minimal cost which can be used to serve the first t requests starting at the configuration A_0 and ending at the configuration X. This value is denoted by $w_t(X)$. The Work Function algorithm is based on the above defined work function. Suppose that A_{t-1} is the server configuration before the arrival of the t-th request, and denote the place of the t-th request by R_t. Then the Work Function algorithm uses the server s to serve the request for which the value $w_{t-1}(A_{t-1} \setminus \{P\} \cup \{R_t\}) + d(P, R_t)$ is minimal, where P denotes the point where the server is actually located.

9.2. Example. Consider the metric space which contains three points A, B and C and the distances are $d(A,B) = 1$, $d(B,C) = 2$, $d(A,C) = 3$. Suppose that we have two servers and the starting configuration is $\{A, B\}$. Then the starting work function values are $w_0(\{A,A\}) = 1$, $w_0(\{A,B\}) = 0$, $w_0(\{A,C\}) = 2$, $w_0(\{B,B\}) = 1$, $w_0(\{B,C\}) = 3$, $w_0(\{C,C\}) = 4$. Suppose that the first request appears at point C. Then $w_0(\{A,B\} \setminus \{A\} \cup \{C\}) + d(A,C) = 3 + 3 = 6$ and $w_0(\{A,B\} \setminus \{B\} \cup \{C\}) + d(B,C) = 2 + 2 = 4$, thus algorithm Work Function uses the server from point B to serve the request.

The following statement is valid for the algorithm.

Theorem 9.3 *The* Work Function *algorithm is weakly* $2k - 1$*-competitive.*

Beside the general problem many particular cases were investigated. If the distance of any pair of points is 1, then we obtain the on-line paging problem as the special case. An another well investigated metric space is the line. The points of the line are considered as real numbers, and the distance of points a and b is $|a - b|$. In this special case a k-competitive algorithm was developed by Chrobak and Larmore, the algorithm is called Double-coverage. A request at point P is served by the server s which is closest to P. Moreover, if there are also servers on the opposite side of P then the closest server among them moves distance $d(s, P)$ into the direction of P. In the following parts we denote the Double-coverage algorithm by DC. The input of the algorithm is the list of requests which is a list of points (real numbers) denoted by $I = P_1, \ldots, P_n$ and the starting configuration of the servers denoted by $S = (s_1, \ldots, s_k)$ which contains also points (real numbers). The algorithm can be defined by the following pseudocode:

```
DC($I, S$)

for $j \leftarrow 1$ to $n$
   do $i \leftarrow \mathrm{argmin}_l d(P_j, s_l)$
if $s_i = \max_l s_l$ or $s_i = \max_l s_l$
  then the request is served by the $i$-th server
       $s_i \leftarrow P_j$
else if $s_i \le P_j$
     then $m \leftarrow \mathrm{argmin}_{l: s_l > P_j} d(s_l, P_j)$
          the request is served by the $i$-th server
          $s_m \leftarrow s_m - d(s_i, P_j)$
          $s_i \leftarrow P_j$
else if $s_i \ge P_j$
     then $n \leftarrow \mathrm{argmin}_{l: s_l < P_j} d(s_l, P_j)$
          the request is served by the $i$-th server
          $s_n \leftarrow s_n + d(s_i, P_j)$
          $s_i \leftarrow P_j$
```

9.3. Example. Suppose that there are three servers s_1, s_2, s_3 located at the points $0, 1, 2$. If the next request appears at point 4 then DC uses the closest server s_3 to serve the request. The locations of the other servers remain unchanged, the cost of serving the request is 2 and the servers are at the points $0, 1, 4$. If the next request appears at point 2 then DC uses the closest server s_2 to serve the request, but there are servers on the opposite side of the request thus s_3 also moves distance 1 into the direction of 2. Therefore the cost of serving the request is 2 and the servers will be at the points $0, 2, 3$.

The following statement which can be proved by the potential function technique is valid for algorithm DC. This technique is often used in the analysis of on-line algorithms.

Theorem 9.4 *If the metric space is the line then algorithm* DC *is weakly k-competitive.*

Proof. Consider an arbitrary sequence of requests denote this input by I. During the analysis of the procedure we suppose that one offline optimal algorithm and DC are running parallel on the input. We also suppose that each request is served first by the offline algorithm and then by the on-line algorithm. The servers of the on-line algorithm and also the positions of the servers (which are real numbers) are denoted by $s_1, \dots, s_k$, and the servers of the optimal offline algorithm and also the positions of the servers are denoted by $x_1, \dots, x_k$. We suppose that for the positions $s_1 \le s_2 \le \dots \le s_k$ and $x_1 \le x_2 \le \dots \le x_k$ are always valid, this can be achieved by changing the notations of the servers.

We prove the theorem by the potential function technique. The potential function assigns a value to the actual positions of the servers, the on-line and offline costs are compared using the changes of the potential function. Define the following potential function

$$\Phi = k \sum_{i=1}^{k} |x_i - s_i| + \sum_{i<j} (s_j - s_i) \ .$$

We show that the following statements are valid for the potential function.

- While OPT serves a request the increase of the potential function is not more than k times the distance moved by the servers of OPT.
- While DC serves a request, the decrease of Φ is at least as much as the cost of serving the request.

If the above properties are valid then one can prove easily the theorem. In this case $\Phi_f - \Phi_0 \le k \cdot \mathrm{OPT}(I) - \mathrm{DC}(I)$, where Φ_f and Φ_0 are the final and the starting values of the potential function. Furthermore Φ is nonnegative so we obtain that $\mathrm{DC}(I) \le k\mathrm{OPT}(I) + \Phi_0$, which yields by the definition that the algorithms is weakly k-competitive.

Now we prove the properties of the potential function.

First consider the case when one of the offline servers moves distance d. Then the first part of the potential function increases at most kd. The second part does not change thus we proved the first property of the potential function.

Consider the servers of DC. Suppose that the request appears at point P. Since the request is first served by OPT, $x_j = P$ for some j. Distinguish the following two cases depending on the positions of the on-line servers.

Suppose first that the on-line servers are on the same side of P. We can assume that the positions of the servers are not smaller than P, the other case is completely similar. Then s_1 is the closest server and DC sends s_1 to P, the other on-line servers do not move. Therefore the cost of DC is $d(s_1, P)$. In the first sum of the potential function only $|x_1 - s_1|$ is changing that decreases $d(s_1, P)$, thus the first part decreases $kd(s_1, P)$. The second sum is increasing the increase is $(k-1)d(s_1, P)$, thus the value of Φ decreases $d(s_1, P)$.

Consider the second case. Then there are servers on both sides of P, suppose that the closest servers are s_i and s_{i+1}. We assume that s_i is closer to P, the other case is completely similar. Then the cost of DC is $2d(s_i, P)$. Consider the first sum of the potential function. The i-th and the $i+1$-th part are changing. Since $x_j = P$ for some j thus one of the i-th and the $i+1$-th parts decreases $d(s_i, P)$, the increase of the other one is at most $d(s_i, P)$ thus the first sum does not increase. The change of the second sum of Φ is

$$d(s_i, P)(-(k-i)+(i-1)-(i)+(k-(i+1))) = -2d(s_i, P) .$$

Therefore we proved that the second property of the potential function is also valid in this case.

Since we investigated all of the possible cases we proved the properties of the potential function, and the statement of the theorem follows. ■

Exercises

9.2-1 Suppose that (M, d) is a metric space. Prove that (M, q) is also a metric space where $q(x, y) = \min\{1, d(x, y)\}$.

9.2-2 Consider the greedy algorithm which serves each request by the server which is closest to the place of the request. Prove that the algorithm is not constant competitive for the line.

9.2-3 Prove that for arbitrary k-element multisets X and Z and for arbitrary t the inequality $w_t(Z) \leq w_t(X) + d(X, Z)$ is valid, where $d(X, Z)$ is the cost of the minimal matching of X and Z, (the minimal cost which can be used to move the servers from configuration X to configuration Z).

9.2-4 Consider the line as a metric space. Suppose that the servers of the on-line algorithm are at the points $2, 4, 5, 7$, and the servers of the offline algorithm are at the points $1, 3, 6, 9$. Calculate the value of the potential function used in the proof of theorem 9.4. How this potential function is changed if the on-line server moves from point 7 to point 8?

9.3. Models related to computer networks

The theory of computer networks became one of the most significant areas of the computer science. In the planning of computer networks many optimization problems arise and most of these problems are actually on-line, since neither the traffic nor the changes in the topology of a computer network cannot be precisely predicted. Recently some researchers working at the area of on-line algorithms defined some on-line mathematical models for problems related to computer networks. In this section we consider this area we present three problems and show the basic results. First the data acknowledgement problem is considered, then we present the web caching problem, and the section is closed by the on-line routing problem.

9.3.1. The data acknowledgement problem

In the communication of a computer network the information is sent by packets. If the communication channel is not completely safe then the arrival of the packets are acknowledged. In the data acknowledgement problem we try to determine the time of sending acknowledgements. One acknowledgement can acknowledge many packets but waiting for long time can cause the resending of the packets and that results the congestion of the network. On the other hand sending immediately an acknowledgement about the arrival of each packet would cause again the congestion of the network. The first optimization model for determining the sending time of the acknowledgements was developed by Dooly, Goldman and Scott in 1998. Below we present the developed model and some of the basic results.

In the mathematical model of the data acknowledgement problem the input is the list of the arrival times $a_1, \ldots, a_n$ of the packets. The decision maker has to determine when to send acknowledgements, these times are denoted by $t_1, \ldots, t_k$. In the optimization model the cost function is:

$$k + \sum_{j=1}^{k} v_j \ ,$$

where k is the number of the sent acknowledgements and $v_j = \sum_{t_{j-1} < a_i \le t_j} (t_j - a_i)$ is the total latency collected by the j-th acknowledgement. We consider the on-line problem which means that at time t the decision maker only knows the arrival times of the packets already arrived and has no information about the further packets. We denote the set of the unacknowledged packets at the arrival time a_i by σ_i.

For the solution of the problem the class of the alarming algorithms was developed. An ***alarming algorithm*** works as follows. At the arrival time a_j an alarm is set for time $a_j + e_j$. If no packet arrives before time $a_j + e_j$, then an acknowledgement is sent at time $a_j + e_j$ which acknowledges all of the unacknowledged packets. Otherwise at the arrival of the next packet at time a_{j+1} the alarm is reset for time $a_{j+1} + e_{j+1}$. Below we analyze in details an algorithm from this class. This algorithm sets the alarm to collect total latency 1 by the acknowledgement. The algorithm is called ALARM. We obtain the above defined rule from the general definition using the solution of the following equation as value e_j

$$1 = |\sigma_j| e_j + \sum_{a_i \in \sigma_j} (a_j - a_i) \ .$$

9.4. Example. Consider the following example. The first packet arrives at time 0 ($a_1 = 0$), then ALARM sets an alarm with the value $e_1 = 1$ for time 1. Suppose that the next arrival time is $a_2 = 1/2$. This arrival is before the alarm time thus the first packet is not acknowledged yet and we reset the alarm with the value $e_2 = (1 - 1/2)/2 = 1/4$ for time $1/2 + 1/4$. Suppose that the next arrival time is $a_3 = 5/8$. This arrival is before the alarm time thus the first two packets are not acknowledged yet and we reset the alarm with value $e_3 = (1 - 5/8 - 1/8)/3 = 1/12$ for time $5/8 + 1/12$. Suppose that the next arrival time is $a_4 = 1$. Then no packet arrived before the alarm time $5/8 + 1/12$, thus at that time the first three packets were acknowledged and the alarm is set for the new packet with the value $e_4 = 1$ for time 2.

The following theorem is valid for the competitive ratio of the algorithm.

Theorem 9.5 *Algorithm* ALARM *is 2-competitive.*

Proof. Suppose that algorithm ALARM sends k acknowledgements. These acknowledgements divide the time into k time intervals. The cost of the algorithm is $2k$, since k is the cost of the acknowledgements, and the alarm is set to have total latency 1 for each acknowledgement.

Suppose that the optimal offline algorithm sends k^* acknowledgements. If $k^* \ge k$, then $\mathrm{OPT}(I) \ge k = \mathrm{ALARM}(I)/2$ is obviously valid, thus we have that the algorithm is 2-competitive. If $k^* < k$, then at least $k - k^*$ among the time intervals defined by the acknowledgements of algorithm ALARM do not contain any of the offline acknowledgements. This yields that the offline total latency is at most $k - k^*$, thus we obtain that $\mathrm{OPT}(I) \geq k$ which inequality proves that ALARM is 2-competitive. ■

As the following theorem shows algorithm ALARM has the smallest possible competitive ratio.

Theorem 9.6 *There is not such on-line algorithm for the data acknowledgement problem which has smaller competitive ratio than* 2.

Proof. Consider an arbitrary on-line algorithm denote it by ONL. Analyze the following input. Consider a long sequence of packets where the packets always arrive immediately after the time when ONL sends an acknowledgement. Then the on-line cost of a sequence containing $2n$ packets is $\mathrm{ONL}(I_{2n}) = 2n + t_{2n}$, since the cost resulted from the acknowledgements is $2n$, and the latency of the i-th acknowledgement is $t_i - t_{i-1}$, where the value $t_0 = 0$ is used.

Consider the following two on-line algorithms. ODD sends the acknowledgements after the odd numbered packets and EV sends the acknowledgements after the even numbered packets.

Then the costs achieved by these algorithms are

$$\mathrm{EV}(I_{2n}) = n + \sum_{i=0}^{n-1}(t_{2i+1} - t_{2i}) \ ,$$

and

$$\mathrm{ODD} = n + 1 + \sum_{i=1}^{n}(t_{2i} - t_{2i-1}) \ .$$

Therefore $\mathrm{EV}(I_{2n}) + \mathrm{ODD}(I_{2n}) = \mathrm{ONL}(I_{2n}) + 1$. On the other hand none of the costs achieved by ODD and EV is greater than the optimal offline cost thus $\mathrm{OPT}(I_{2n}) \leq \min\{\mathrm{EV}(I_{2n}), \mathrm{ODD}(I_{2n})\}$, which yields that $\mathrm{ONL}(I_{2n})/\mathrm{OPT}(I_{2n}) \geq 2 - 1/\mathrm{OPT}(I_{2n})$. By this inequality it follows that the competitive ratio of ONL is not smaller than 2, because using a sufficiently long sequence of packets the value $\mathrm{OPT}(I_{2n})$ can be arbitrarily large. ■

9.3.2. The file caching problem

The file caching problem is a generalization of the caching problem presented in the memory management chapter. The world-wide-web browsers are using caches to store some files. This makes it possible to use the stored files if a user wants to see some web-page many times during a short time interval. If the cache becomes full then some files must be eliminated to make place for the new file. The file caching problem models this scenario, the goal is to find good strategies for determining which files should be eliminated. The difference to the standard paging problem is that the files have size and retrieval cost (the problem is reduced to the paging if each size and each retrieval cost are 1). So the following mathematical model describes the problem.

There is a given cache of size k, the input is a sequence of pages. Each page p has a ***size*** denoted by $s(p)$ and a ***retrieval cost*** denoted by $c(p)$. The pages arrive from a list one by one and after the arrival of a page the algorithm has to place it into the cache. If the page is not contained in the cache and there is not enough place to put it into the cache then the algorithm has to delete some pages from the cache to make enough place for the requested page. If the required page is in the cache then the cost of serving the request is 0 otherwise the cost is $c(p)$. The objective is to minimize the total cost. The problem is on-line which means that for the decisions (which pages should be deleted from the cache) only the earlier pages and decisions can be used, the algorithm has no information about the further pages. We assume that the size of the cache and also the sizes of the pages are positive integers.

For the solution of the problem and for its particular cases many algorithms were developed. Here we present algorithm Landlord which algorithm was developed by Young.

The algorithm maintains a credit value $0 \leq cr(f) \leq c(f)$ for each page f which is contained in the actual cache. In the following part of the section the set of the pages in the actual cache of Landlord is denoted by LA. If Landlord has to retrieve a page g then the following steps are performed.

```
Landlord($LA, g$)

1  **if** $g$ is not contained in $LA$
2    **then while** there is not enough place for $g$
3    $\Delta \leftarrow \min_{f \in LA} cr(f)/s(f)$
4    for each $f \in LA$ let $cr(f) \leftarrow cr(f) - \Delta \cdot s(f)$
5    evict some pages with $cr(f) = 0$
6          place $g$ into the cache $LA$ and let $cr(g) \leftarrow c(g)$
7  **else** reset $cr(g)$ to any value between $cr(g)$ and $c(g)$
```

9.5. Example. Suppose that $k = 10$ and LA contains the following three pages: g_1 with $s(g_1) = 2, cr(g_1) = 1$, g_2 with $s(g_2) = 4, cr(g_2) = 3$ and g_3 with $s(g_3) = 3, cr(g_3) = 3$. Suppose that the next requested page is g_4, with the parameters $s(g_4) = 4, c(g_4) = 4$. Then there is not enough place for it in the cache, so some pages must be evicted. Landlord determines the value $\Delta = 1/2$ and changes the credits as follows: $cr(g_1) = 0, cr(g_2) = 1, cr(g_3) = 3/2$, thus g_1 is evicted from the cache LA. Still there is not enough place for g_4 in the cache. The new Δ value is $\Delta = 1/4$ and the new credits are: $cr(g_2) = 0, cr(g_3) = 3/4$, thus g_2 is evicted from the cache. Then there is enough place for g_4, thus it is placed into the cache LA with the credit value $cr(g_4) = 4$.

Landlord is weakly k-competitive, but a stronger statement is also true. For the web caching problem an on-line algorithm ALG is called (C, k, h)-competitive, if there exists such an absolute constant B, that $\mathrm{ALG}_k(I) \leq C \cdot \mathrm{OPT}_h(I) + B$ is valid for each input, where $\mathrm{ALG}_k(I)$ is the cost of ALG using a cache of size k and $\mathrm{OPT}_h(I)$ is the optimal offline cost using a cache of size h. The following statement is true for algorithm Landlord.

Theorem 9.7 *If $h \leq k$, then algorithm* Landlord *is $(k/(k - h + 1), k, h)$-competitive.*

Proof. Consider an arbitrary input sequence of pages, denote the input by I. We use the potential function technique. During the analysis of the procedure we suppose that an offline optimal algorithm with cache size h and Landlord with cache size k are running parallel on the input. We also suppose that each page is placed first by the offline algorithm into the offline cache and then it is placed by the on-line algorithm into LA. We denote the set of the pages contained in the actual cache of the optimal offline algorithm by OPT. Consider the following potential function

$$\Phi = (h-1) \sum_{f \in LA} cr(f) + k \sum_{f \in OPT} (c(f) - cr(f)) \ .$$

Investigate the changes of the potential function during the retrievals of a page g.

- OPT places g into its cache.

  Then OPT has cost $c(g)$. In the potential function only the second part may change. On the other hand $cr(g) \geq 0$, thus the increase of the potential function is at most $k \cdot c(g)$.

- Landlord decreases the credit value for each $f \in LA$.

  In this case for each $f \in LA$ the decrease of $cr(f)$ is $\Delta \cdot s(f)$, thus the decrease of Φ is

  $$\Delta((h-1)s(LA) - ks(OPT \cap LA)) \ ,$$

  where $s(LA)$ and $s(OPT \cap LA)$ denote the total size of the pages contained in sets LA and $OPT \cap LA$ respectively. At the time when this step is performed OPT have already placed the page g into its cache OPT, but the page is not contained in the cache LA. Therefore $s(OPT \cap LA) \leq h - s(g)$. On the other hand this step is performed if there is not enough place for the page in LA thus $s(LA) > k - s(g)$, which yields $s(LA) \geq k - s(g) + 1$ because the sizes are positive integers. Therefore we obtain that the decrease of Φ is at least

  $$\Delta((h-1)(k - s(g) + 1) - k(h - s(g))) \ .$$

  Since $s(g) \geq 1$ and $k \geq h$, this decrease is at least $\Delta((h-1)(k-1+1) - k(h-1)) = 0$.

- Landlord evicts a page f from cache LA.

  Since Landlord evicts only pages having credit 0, thus during this step Φ remains unchanged.

- Landlord places page g into the cache LA and sets the value $cr(g) = c(g)$.

  Then the cost of Landlord is $c(g)$. On the other hand g was not contained in the cache LA before the performance of this step, thus $cr(g) = 0$ was valid. Furthermore first OPT places the page into its cache thus $g \in OPT$ is also valid. Therefore the decrease of Φ is $-(h-1)c(g) + kc(g) = (k - h + 1)c(g)$.

- Landlord resets for a page $g \in HA$ the credit to a value between $cr(g)$ and $c(g)$.

  In this case $g \in OPT$ is valid, since OPT places first the page g into its cache. The value $cr(g)$ is not decreased and $k > h - 1$, thus Φ can not increase during this step.

We have investigated the possible steps of the algorithms and we proved the following properties of the potential function.

- If OPT places a page into its cache, then the increase of the potential function is at most k times more than the cost of OPT.
- If Landlord places a page into its cache, then the decrease of Φ is $(k-h+1)$ times more than the cost of Landlord.
- During the other steps Φ does not increase.

By the above properties we obtain that $\Phi_f - \Phi_0 \le k \cdot \mathrm{OPT}_h(I) - (k-h+1) \cdot \textsc{Landlord}_k(I)$, where Φ_0 and Φ_f are the starting and final values of the potential function. The potential function is nonnegative, thus we obtain that $(k-h+1) \cdot \textsc{Landlord}_k(I) \le k \cdot \mathrm{OPT}_h(I) + \Phi_0$, which proves that Landlord is $(k/(k-h+1), k, h)$-competitive. ■

9.3.3. On-line routing

In computer networks the congestion of the communication channels decreases the speed of the communication and may cause loss of information. Thus congestion control is one of the most important problems in the area of computer networks. A related important problem is the routing of the communication where we have to determine the path in the network for the messages. Since we have no information about the further traffic of the network thus routing is an on-line problem. Here we present two on-line optimization models for the routing problem.

The mathematical model

The network is given by a graph, each edge e has a maximal available bandwidth denoted by $u(e)$, the number of edges is denoted by m. The input is a sequence of requests, where the j-th request is given by a vector $(s_j, t_j, r_j, d_j, b_j)$, and to satisfy the request bandwidth r_j must be reserved on a path from s_j to t_j for time duration d_j, the benefit of serving a request is b_j. In the followings we assume the assumption $d_j = \infty$, and we omit the value of d_j from the requests. The problem is on-line which means that after the arrival of a request the algorithm has to make the decisions without any information about the further requests. We consider the following two models.

Load balancing model: In this model all requests must be satisfied. The objective is to minimize the maximum of the overload of the edges. The overload is the ratio of the sum of the bandwidths reserved on the edge and the available bandwidth. Since each request is served thus the benefit is not significant in this model.

Throughput model: In this model the decision maker is allowed to reject some requests. The sum of the bandwidths reserved on an edge can not be more than the available bandwidth. The goal is to maximize the sum of the benefits of the accepted requests. We investigate this model in details. It is important to note that this is a maximization problem thus we use the notion of competitiveness in the form defined for maximization problems.

Below we define the exponential algorithm. We need the following notations to define and analyze the algorithm. Denote P_i the path which is assigned to the accepted request i. Let A denote the set of requests accepted by the on-line algorithm. Then $l_e(j) = \sum_{i \in A, i<j, e \in P_i} r_i/u(e)$ is the ratio of the reserved bandwidth and the available bandwidth on e before the arrival of request j.

The basic idea of the exponential algorithm is the following. The algorithm assigns a cost which is exponential in $l_e(j)$ to each e and chooses the path which has the minimal cost. Below we define and analyze the exponential algorithm for the throughput model. Let μ be a constant which depends on the parameters of the problem, its value will be given later. Let $c_e(j) = \mu^{l_e(j)}$, for each request j and edge e. Then the exponential algorithm performs the following steps after the arrival of a request (s_j, t_j, r_j, b_j).

```
EXP($s_j, t_j, r_j, b_j$)
1 let $U_j$ be the set of the paths $(s_j, t_j)$
2 $P_j \leftarrow \mathrm{argmin}_{P \in U_j}\{\sum_{e \in P} \frac{r_j}{u(e)} c_e(j)\}$
3 if $C(P_j) = \sum_{e \in P_j} \frac{r_j}{u(e)} c_e(j) \leq 2mb_j$
4     then reserve the bandwidth $r_j$ on path $P_j$
5 else reject the request
```

Remark. If we modify this algorithm to accept each request then we obtain an exponential algorithm for the load balancing model.

9.6. Example. Consider the network which contains the vertices A, B, C, D and the edges (A, B), (B, D), (A, C), (C, D), where the available bandwidths of the edges are $u(A, B) = 1$, $u(B, D) = 3/2$, $u(A, C) = 2$, $u(C, D) = 3/2$. Suppose that $\mu = 10$ and that the reserved bandwidths are: $3/4$ on the path (A, B, D), $5/4$ on the path (A, C, D), $1/2$ on the path (B, D), $1/2$ on the path (A, C). The next request j is to reserve bandwidth $1/8$ on some path between A and D. Then the values $l_e(j)$ are: $l_{(A,B)}(j) = (3/4) : 1 = 3/4$, $l_{(B,D)}(j) = (3/4 + 1/2) : (3/2) = 5/6$, $l_{(A,C)}(j) = (5/4 + 1/2) : 2 = 7/8$, $l_{(C,D)}(j) = (5/4) : (3/2) = 5/6$. There are two paths between A and D, the costs are:

$$C(A, B, D) = 1/8 \cdot 10^{3/4} + 1/12 \cdot 10^{5/6} = 1.269 ,$$

$$C(A, C, D) = 1/16 \cdot 10^{7/8} + 1/12 \cdot 10^{5/6} = 1.035 .$$

The minimal cost belongs to the path (A, C, D). Therefore, if $2mb_j = 8b_j \geq 1,035$, then the request is accepted and the bandwidth is reserved on the path (A, C, D). Otherwise the request is rejected.

To analyze the algorithm consider an arbitrary input sequence I. Denote A the set of the requests accepted by EXP, and denote A^* the set of the requests which are accepted by OPT and rejected by EXP. Furthermore denote P_j^* the path reserved by OPT for each request j accepted by OPT. Define the value $l_e(v) = \sum_{i \in A, e \in P_i} r_i/u(e)$ for each e, this value gives the ratio of the reserved bandwidth and the available bandwidth for e at the end of the on-line algorithm.

Let $\mu = 4mPB$, where B is an upper bound on the benefits and for each request and each edge the inequality

$$\frac{1}{P} \leq \frac{r(j)}{u(e)} \leq \frac{1}{\lg \mu}$$

is valid. Then the following statements hold.

Lemma 9.8 *The solution given by algorithm* EXP *is feasible, the sum of the reserved bandwidths is not more than the available bandwidth for each edge.*

Proof. We prove the statement by contradiction. Suppose that there is an edge f where the available bandwidth is violated. Let j be the first such accepted request which violates the available bandwidth on f.

The inequality $r_j/u(f) \leq 1/\lg \mu$ is valid for j and f (it is valid for all edges and requests). Furthermore after the acceptance of request j the sum of the bandwidths is greater than the available bandwidth on edge f, thus we obtain that $l_f(j) > 1 - 1/\lg \mu$. On the other hand this yields that the inequality

$$C(P_j) = \sum_{e \in P_j} \frac{r_j}{u(e)} c_e(j) \geq \frac{r_j}{u(f)} c_f(j) > \frac{r_j}{u(f)} \mu^{1-1/\lg \mu}$$

holds for the value $C(P_j)$ used in algorithm EXP. Using the assumption on P we obtain that $\frac{r_j}{u(e)} \geq \frac{1}{P}$, and $\mu^{1-1/\lg m} = \mu/2$, thus by the above inequality we obtain that

$$C(P) > \frac{1}{P}\frac{\mu}{2} = 2mB \ .$$

On the other hand this inequality is a contradiction since EXP would reject the request. Therefore we obtained a contradiction thus we proved the statement of the lemma. ■

Lemma 9.9 *For the solution given by* OPT *the following inequality holds:*

$$\sum_{j \in A^*} b_j \leq \frac{1}{2m} \sum_{e \in E} c_e(v) \ .$$

Proof. Since EXP rejected j for each $j \in A^*$, thus $b_j < \frac{1}{2m} \sum_{e \in P_j^*} \frac{r_j}{u(e)} c_e(j)$ for each $j \in A^*$, because this inequality is valid for all paths between s_j and t_j. Therefore

$$\sum_{j \in A^*} b_j < \frac{1}{2m} \sum_{j \in A^*} \sum_{e \in P_j^*} \frac{r_j}{u(e)} c_e(j) \ .$$

On the other hand $c_e(j) \leq c_e(v)$ holds for each e, thus we obtain that

$$\sum_{j \in A^*} b_j < \frac{1}{2m} \sum_{e \in E} c_e(v) \Big(\sum_{j \in A^* : e \in P_j^*} \frac{r_j}{u(e)} \Big) \ .$$

The sum of the bandwidths reserved by OPT is at most the available bandwidth $u(e)$ for each e, thus $\sum_{j \in A^* : e \in P^*(j)} \frac{r_j}{u(e)} \leq 1$.

Consequently

$$\sum_{j \in A^*} b_j \leq \frac{1}{2m} \sum_{e \in E} c_e(v)$$

which inequality is the one which we wanted to prove. ■

Lemma 9.10 *For the solution given by algorithm* EXP *the following inequality holds*

$$\frac{1}{2m} \sum_{e \in E} c_e(v) \leq (1 + \lg \mu) \sum_{j \in A} b_j \ .$$

Proof. To prove the lemma it is enough to show that the inequality $\sum_{e \in P_j} (c_e(j+1) - c_e(j)) \leq 2mb_j \log_2 \mu$ is valid for each request $j \in A$. On the other hand

$$c_e(j+1) - c_e(j) = \mu^{l_e(j) + \frac{r_j}{u(e)}} - \mu^{l_e(j)} = \mu^{l_e(j)} (2^{\log_2 \mu \frac{r_j}{u(e)}} - 1) \ .$$

Since $2^x - 1 < x$, if $0 \leq x \leq 1$, and by the assumptions $0 \leq \log_2 \mu \frac{r_j}{u(e)} \leq 1$, thus we obtain that

$$c_e(j+1) - c_e(j) \leq \mu^{l_e(j)} \log_2 \mu \frac{r_j}{u(e)}.$$

Summarizing the bounds given above we obtain that

$$\sum_{e\in P_j}(c_e(j+1)-c_e(j)) \leq \log_2\mu \sum_{e\in P_j}\mu^{l_e(j)}\frac{r_j}{u(e)} = \log_2\mu\cdot C(P_j)\ .$$

Since EXP accepts the requests with the property $C(P_j)\leq 2mb_j$, thus the above inequality proves the required statement. ■

By the above lemmas we can prove the following theorem.

Theorem 9.11 *Algorithm* EXP *is* $1/O(\lg\mu)$*-competitive, if* $\mu = 4mPB$*, where B is an upper bound on the benefits, and for all edges and requests*

$$\frac{1}{P}\leq\frac{r(j)}{u(e)}\leq\frac{1}{\lg\mu}.$$

Proof. By lemma 9.8 it follows that the algorithm results in a feasible solution where the available bandwidths are not violated. Using the notations defined before the lemmas we obtain that the benefit of algorithm EXP on the input I is $\mathrm{EXP}(I)=\sum_{j\in A}b_j$, and the benefit of OPT is at most $\sum_{j\in A\cup A^*}b_j$. Therefore by lemma 9.9 and lemma 9.10 it follows that

$$\mathrm{OPT}(I)\leq\sum_{j\in A\cup A^*}b_j\leq(2+\log_2\mu)\sum_{j\in A}b_j\leq(2+\log_2\mu)\mathrm{EXP}(I)\ ,$$

which inequality proves the theorem. ■

Exercises

9.3-1 Consider the modified version of the data acknowledgement problem with the objective function $k+\sum_{j=1}^{k}\mu_j$, where k is the number of acknowledgements and $\mu_j=\max_{t_{j-1}<a_i\leq t_j}\{t_j-a_i\}$ is the maximal latency of the j-th acknowledgement. Prove that algorithm Alarm is also 2-competitive in this modified model.

9.3-2 Represent the special case of the web caching problem, where $s(g)=c(g)=1$ for each page g as a special case of the k-server problem. Define the metric space which can be used.

9.3-3 In the web caching problem the cache LA of size 8 contains three pages a,b,c with the following sizes and credits: $s(a)=3, s(b)=2, s(c)=3, cr(a)=2, cr(b)=1/2, cr(c)=2$. We want to retrieve a page d of size 3, and retrieval cost 4. The optimal offline algorithm OPT with cache of size 6 already placed the page into its cache, its cache contains the pages d and c. Which pages are evicted by Landlord to place d? What kind of changes the potential function defined in the proof of theorem 9.7 has?

9.3-4 Prove that if in the throughput model no bounds are given for the ratios $r(j)/u(e)$ then there is not constant-competitive on-line algorithm.

9.4. On-line bin packing models

In this section we consider the on-line bin packing problem and its multidimensional generalizations. First we present the classical on-line bin packing problem and some fundamental results of the area. Then we define the multidimensional generalizations and present some details from the area of on-line strip packing.

9.4.1. On-line bin packing

In the bin packing problem the input is a list of items, where the i-th item is given by its size $a_i \in (0, 1]$. The goal is to pack the items into unit size bins and minimize the number of the used bins. In a more formal way we can say that we have to divide the items into groups where each group has the property that the total size of the items is at most 1 and the goal is to minimize the number of groups. This problem also appears in the area of memory management.

In this section we investigate the on-line problem which means that the decision maker has to make decisions about the packing of the i-th item based on the values $a_1, \dots, a_i$ without any information about the further items.

NF algorithm, bounded space algorithms

First we consider the model where the number of the open bins is limited. The k-bounded space model means that if the number of open bins reaches the bound k then the algorithm can open a new bin only after closing some of the bins, and the closed bins cannot be used for packing further items. If only one bin can be open then the natural algorithm packs the item into the open bin if it fits otherwise it closes the bin, opens a new one and put the item into it. We call this algorithm NF (Next Fit) algorithm. We do not present the pseudocode of the algorithm it can be found in this book in the chapter about memory management. The asymptotic competitive ratio of algorithm NF is determined by the following theorem.

Theorem 9.12 *The asymptotic competitive ratio of* NF *is 2.*

Proof. Consider an arbitrary sequence of items, denote it by σ. Denote n the number of bins used by OPT and denote m the number of bins used by NF. Furthermore let S_i, $i = 1, \dots, m$ the total size of the items packed into the i-th bin by NF.

Then $S_i + S_{i+1} > 1$, since in the opposite case the first item of the $(i + 1)$-th bin fits into the i-th bin which contradicts to the definition of the algorithm. Therefore the total size of the items is more than $\lfloor m/2 \rfloor$.

On the other hand the optimal offline algorithm cannot put items with total size more than 1 into the same bin, thus we obtain that $n > \lfloor m/2 \rfloor$. This yields that $m \le 2n - 1$, thus

$$\frac{\mathrm{NF}(\sigma)}{\mathrm{OPT}(\sigma)} \le \frac{2n-1}{n} = 2 - 1/n \ .$$

Consequently we proved that the algorithm is asymptotically 2-competitive.

Now we prove that the bound is tight. Consider for each n the following sequence denoted by σ_n. The sequence contains $4n - 2$ items, the size of the $2i - 1$-th item is $1/2$, the size of the $2i$-th item is $1/4n$, $i = 1, \dots, 2n$. Then algorithm NF puts the $(2i - 1)$-th and the $2i$-th items into the i-th bin for each bin, thus $\mathrm{NF}(\sigma_n) = 2n - 1$. The optimal offline algorithm puts pairs of $1/2$ size items into the first $n - 1$ bins and it puts one $1/2$ size item and the small items into the n-th bin, thus $\mathrm{OPT}(\sigma_n) = n$. Since $\mathrm{NF}(\sigma_n)/\mathrm{OPT}(\sigma_n) = 2 - 1/n$ and this function tends to 2 as n tends to ∞, thus we proved that the asymptotic competitive ratio of the algorithm is at least 2. ■

If $k > 1$ then better algorithms than NF are known for the k-bounded space model. The best known bounded space on-line algorithms belong to the family of ***harmonic algorithms***, where the basic idea is that the interval $(0, 1]$ is divided into subintervals and each item has a type which is the subinterval of its size. The items of the different types are packed into different bins. The algorithm uses parallel versions of NF to pack the items belonging to the same type.

Algorithm FF and the weight function technique

In this part we present a method which is often used in the analysis of the bin packing algorithms. We show the method by analyzing algorithm FF (First Fit).

Algorithm FF can be used if the number of open bins is not bounded. The algorithm puts the item into the earliest opened bin where it fits. If the item does not fit into any of the bins then a new bin is opened and the algorithm puts the item into it. The pseudocode of the algorithm is also presented in the chapter of memory management. The asymptotic competitive ratio of the algorithm is bounded above by the following theorem.

Theorem 9.13 FF *is asymptotically 1.7-competitive.*

Proof. In the proof we use the weight function technique where the idea is that a weight is assigned to each item which measures in some way that how difficult can be to pack the item. Then the weight function and the total size of the items are used to bound the offline and on-line objective function values. Define the following weight function:

$$w(x) = \begin{cases} 6x/5, & \text{ha } 0 \le x \le 1/6 \\ 9x/5 - 1/10, & \text{ha } 1/6 \le x \le 1/3 \\ 6x/5 + 1/10, & \text{ha } 1/3 \le x \le 1/2 \\ 6x/5 + 2/5, & \text{ha } 1/2 < x \ . \end{cases}$$

Let $w(H) = \sum_{i \in H} w(a_i)$ for any set H of items. Then the following statements are valid for the weight function. Both lemmas can be proven by case disjunction based on the sizes of the possible items. The proofs are long and contain many technical details, therefore here we omit them.

Lemma 9.14 *If* $\sum_{i \in H} a_i \le 1$ *is valid for a set* H *of items, then* $w(H) \le 17/10$ *also holds.*

Lemma 9.15 *For an arbitrary list* L *of items* $w(L) \ge \text{FF}(L) - 2$.

Using these lemmas we can prove easily that the algorithm is asymptotically 1.7-competitive. Consider an arbitrary list L of items. The optimal offline algorithm can pack the items of the list into $\text{OPT}(L)$ bins. The algorithm packs items with total size at most 1 into each bin, thus by Lemma 9.14 it follows that $w(L) \le 1.7\text{OPT}(L)$. On the other hand by Lemma 9.15 we obtain that $\text{FF}(L) - 2 \le w(L)$ which yields that $FF(L) \le 1.7\text{OPT}(L) + 2$, and that inequality proves that the algorithm is asymptotically 1.7-competitive.

It is important to note that the above bound is tight, it is also true that the asymptotic competitive ratio of FF is 1.7. ■

Many algorithms were developed with smaller asymptotic competitive ratio than 17/10, the best known algorithm is asymptotically 1.5888-competitive.

Lower bounds

In this part we consider the techniques for proving lower bounds on the possible asymptotic competitive ratio. First we present a simple lower bound and then we show how the idea of the proof can be extended into a general method.

Theorem 9.16 *No on-line algorithm for the bin packing problem can have smaller asymptotic competitive ratio than 4/3.*

Proof. Let A be an arbitrary on-line algorithm. Consider the following sequence of items. Let $\varepsilon < 1/12$ and L_1 be a list of n items of size $1/3 + \varepsilon$, and L_2 be a list of n items of size $1/2 + \varepsilon$. The input is started by L_1. Then A packs two items or one item into the bins. Denote by k the number of bins containing two items. Then the cost of the algorithm is $A(L_1) = k + n - 2k = n - k$. On the other hand the optimal offline algorithm can pack pairs of items into the bins thus $OPT(L_1) = n/2$.

Now suppose that the input is the combined list L_1L_2. The algorithm is an on-line algorithm it does not know at the beginning that it is the input L_1 or L_1L_2, thus it also uses k bins for packing two items from the part L_1. Therefore among the items of size $1/2 + \varepsilon$ only $n - 2k$ can be paired with earlier items the other ones need their own bin. Thus $A(L_1L_2) \geq n - k + (n - (n - 2k)) = n + k$. On the other hand the optimal offline algorithm can pack one smaller (size $1/3 + \varepsilon$) item and one larger (size $1/2 + \varepsilon$) item into each bin, thus $OPT(L_1L_2) = n$.

So we obtained that there is a list for algorithm A where

$$A(L)/OPT(L) \geq \max\left\{\frac{n-k}{n/2}, \frac{n+k}{n}\right\} \geq 4/3 \ .$$

Moreover in the constructed lists $OPT(L)$ is at least $n/2$ which can be arbitrarily large. This yields that the above inequality proves that the asymptotic competitive ratio of A is at least $4/3$, and this is what we wanted to prove. ■

The fundamental idea of the above proof is that a long sequence is considered (in this proof L_1L_2) and depending on the behaviour of the algorithm such prefix of the sequence is selected as input where the ratio of the costs is maximal. It is a natural extension to consider more difficult sequences. Many lower bounds were proven based on different sequences. On the other hand the computations which are necessary to analyze the sequence became more and more difficult. Below we show how the analysis of such sequences can be interpreted as mixed integer programming problem, which makes it possible to use computers to develop lower bounds.

Consider the following sequence of items. Let $L = L_1L_2 \dots L_k$, where L_i contains $n_i = \alpha_i n$ identical items of size a_i. If algorithm A is asymptotically C-competitive then the inequality

$$C \geq \limsup_{n\to\infty} \frac{\mathrm{A}(L_1 \dots L_j)}{\mathrm{OPT}(L_1 \dots L_j)}$$

is valid for each j. It is enough to consider such algorithm for which the technique can achieve the minimal lower bound, thus or goal is to determine the value

$$R = \min\nolimits_{\mathrm{A}} \max\nolimits_{j=1,\dots,k} \limsup_{n\to\infty} \frac{\mathrm{A}(L_1 \dots L_j)}{\mathrm{OPT}(L_1 \dots L_j)} \ ,$$

which value gives a lower bound on the possible asymptotic competitive ratio. We can determine this value as an optimal solution of a mixed integer programming problem. To define this programming problem we need the following definitions.

The contain of a bin can be described by the packing pattern of the bin, which gives how many elements are contained in the bin from the subsequences. Formally a ***packing pattern*** is a k-dimensional vector $(p_1, \dots, p_k)$, where the coordinate p_j is the number of elements contained in the bin from subsequence L_j. For the packing patterns the constraint $\sum_{j=1}^{k} a_j p_j \leq 1$ must hold. (This constraint ensures that the items described by the packing pattern fit into the bin.)

Classify the set T of the possible packing patterns. For each j let T_j be the set of the patterns for which the first positive coordinate is the j-th. (The pattern p belongs to class T_j if $p_i = 0$ for $i < j$ and $p_j > 0$.)

Consider the packing produced by A. Each bin is packed by some packing pattern therefore the packing can be described by the packing patterns. For each $p \in T$ denote by $n(p)$ the number of bins which are packed by the pattern p. The packing produced by the algorithm is given by the variables $n(p)$.

Observe that the bins which are packed by a pattern from class T_j receive their first element from the subsequence L_j. Therefore we obtain that the number of bins opened by A to pack the elements of subsequence $L_1 \dots L_j$ can be given by the variables $n(p)$ as follows:

$$A(L_1 \dots L_j) = \sum_{i=1}^{j} \sum_{p \in T_i} n(p).$$

Consequently for a given n the required value R can be determined by the solution of the following mixed integer programming problem.

Min R

$$\begin{aligned}
&\textstyle\sum_{p \in T} p_j n(p) = n_j, && 1 \leq j \leq k\\
&\textstyle\sum_{i=1}^{j} \sum_{p \in T_i} n_p \leq R \cdot OPT(L_1 \dots L_j), && 1 \leq j \leq k\\
&n(p) \in \{0, 1, \dots\}, && p \in T
\end{aligned}$$

The first k constraints describe that the algorithm has to pack all items. The second k constraints describe that R is at least as large as the ratio of the on-line and offline costs for the subsequences considered.

By the list $L_1 L_2 \dots L_k$ the set T of the possible packing patterns and also the optimal solutions $OPT(L_1 \dots L_j)$ can be determined.

In this programming problem the number and the value of the variables can be large, thus instead of the problem its linear programming relaxation is considered. Moreover we are interested in the solution under the assumption that n tends to ∞ and it can be proven that the integer programming and the linear programming relaxation give the same bound in this case.

The best currently known bound was proven by this method and it states that no on-line algorithm can have smaller asymptotic competitive ratio than 1.5401.

9.4.2. Multidimensional models

The bin packing problem has three different multidimensional generalizations the vector packing, the box packing and the strip packing models. We only consider in details the strip packing problem. For the other generalizations we just give the model. In the ***vector packing problem*** the input is a list of d-dimensional vectors, and the algorithm has to pack these vectors into the minimal number of bins. A packing is legal for a bin if for each coordinate the sum of the values of the elements packed into the bin is at most 1. In the on-line version the vectors are coming one by one and the algorithm has to assign the vectors to the bins without any information about the further vectors. In the ***box packing problem*** the input is a list of d-dimensional boxes and the goal is to pack the items into the minimal number of d-dimensional unit cube without overlapping. In the on-line version the items are coming one by one and the algorithm has to pack them into the cubes without any information about the further items.

On-line strip packing

In the ***strip packing problem*** there is a set of two dimensional rectangles, defined by their widths and heights, and the task is to pack them without rotation into a vertical strip of width w by minimizing the total height of the strip. We assume that the width of the rectangles is at most w and the height of the rectangles is at most 1. This problem appears in many situations. Usually, scheduling of tasks with shared resource involves two dimensions, the resource and the time. We can consider the widths as the resource and the heights as the time. Our goal is to minimize the total amount of time used. Some applications can be found in computer scheduling problems. We consider the on-line version where the rectangles arrive from a list one by one and we have to pack the rectangle into the vertical strip without any information about the further items. Most of the algorithms developed for the strip packing problem belong to the class of shelf algorithms. We consider this family of algorithms below.

Shelf algorithms

One basic way of packing into the strip is to define shelves and pack the rectangles into the shelves. By ***shelf*** we mean a rectangular part of the strip. Shelf packing algorithms place each rectangle into one of the shelves. If the algorithm decides which shelf will contain the rectangle, then the rectangle is placed into the shelf as much to the left as it is possible without overlapping the other rectangles placed earlier into the shelf considered. Therefore, after the arrival of a rectangle, the algorithm has to make two decisions. The first decision is whether to create a new shelf or not. If the algorithm creates a new shelf it also has to decide the height of the new shelf. The created shelves always start from the top of the previous shelf. The first shelf is placed to the bottom of the strip. The algorithm also has to choose the shelf to which it puts the rectangle. In what follows, we will say that it is possible to pack a rectangle into a shelf, if there is enough room for the rectangle in the shelf. It is obvious that if a rectangle is higher than a shelf we cannot place it into the shelf.

We consider only one algorithm in details. This algorithm was developed and analyzed by Baker and Schwarz in 1983 and it is called NFS$_r$ algorithm. The algorithm depends on a parameter $r < 1$. For each j there is at most one active shelf with height r^j. We give the behaviour of the algorithm below.

After the arrival of a rectangle $p_i = (w_i, h_i)$ choose a value for k which satisfies $r^{k+1} < h_i \leq r^k$. If there is an active shelf with height r^k and it is possible to pack the rectangle into it, then pack it there. If there is no active shelf with height r^k, or it is not possible to pack the rectangle into the active shelf with height r^k, then create a new shelf with height r^k, put the rectangle into it, and let this new shelf be the active shelf with height r^k (if we had earlier an active shelf with height r^k then we close it).

9.7. Example. Let $r = 1/2$. Suppose that the size of the first item is $(w/2, 3/4)$. Then it is assigned to a shelf of height 1. We define a shelf of height 1 at the bottom of the strip, this shelf will be the active shelf with height 1 and we place the item into the left corner of this shelf. Suppose that the size of the next item is $(3w/4, 1/4)$. Then it is placed into a shelf of height $1/4$. There is no active shelf with this height so we define a new shelf of height $1/4$ on the top of the previous shelf. This will be the active shelf of height $1/4$ and the item is placed into its left corner. Suppose that the size of the next item is $(3w/4, 5/8)$. Then this item is placed into a shelf of height 1. It is not possible to pack it into the active shelf thus we close the active shelf and we define a new shelf of height 1 on the top of the previous shelf. This will be the active shelf of height 1 and the item is placed into its left corner. Suppose that the size of the next item is $(w/8, 3/16)$. Then this item is placed into a shelf of height $1/4$. We can pack it into the active shelf of height $1/4$ thus we pack it into that shelf as left as it is possible.

For the competitive ratio of NFS_r the following statements are valid.

Theorem 9.17 *Algorithm* NFS_r *is* $(\frac{2}{r} + \frac{1}{r(1-r)})$*-competitive. Algorithm* NFS_r *is asymptotically* $2/r$*-competitive.*

Proof. First we prove that the algorithm is $(\frac{2}{r} + \frac{1}{r(1-r)})$-competitive. Consider an arbitrary list of rectangles denote it by L. Let H_A denote the sum of the heights of the shelves which are active at the end of the packing, and let H_C be the sum of the heights of the other shelves. Let h be the height of the highest active shelf, and let H be the height of the highest rectangle. Since the algorithm created a shelf with height h, we have $H > rh$. As there is at most 1 active shelf for each height

$$H_A \le h \sum_{i=0}^{\infty} r^i = \frac{h}{1-r} \le \frac{H}{r(1-r)} .$$

Consider the shelves which are not active at the end. Consider the shelves of height hr^i for each i, denote the number of the closed shelves by n_i. Let S be one of these shelves with height hr^i. The next shelf S' with height hr^i contains one rectangle which would not fit into S. Therefore, the total width of the rectangles is at least w. Furthermore the height of these rectangles is at least hr^{i+1}, thus the total area of the rectangles packed into S and S' is at least hwr^{i+1}. If we pair the shelves of height hr^k for each i in this way, using the active shelf if the number of the shelves of the considered height is odd, we obtain that the total area of the rectangles assigned to shelves of height hr^i is at least $wn_ihr^{i+1}/2$. Thus the total area of the rectangles is at least $\sum_{i=0}^{\infty} wn_ihr^{i+1}/2$, and this yields that $\mathrm{OPT}(L) \ge \sum_{i=0}^{\infty} n_ihr^{i+1}/2$. On the other hand the total height of the closed shelves is $H_Z = \sum_{i=0}^{\infty} n_ihr^i$, and we obtain that $H_Z \le 2\mathrm{OPT}(L)/r$.

Since $\mathrm{OPT}(L) \ge H$ is valid we proved the required inequality

$$\mathrm{NFS}_r(L) \le \mathrm{OPT}(L)(2/r + 1/r(1-r) .$$

Since the heights of the rectangles are bounded by 1, thus H and H_A are bounded by a constant so we obtain immediately the result about the asymptotic competitive ratio.

■

Besides this algorithm some other shelf algorithms were investigated for the solution of the problem. We can interpret the basic idea of NFS_r as follows. We define classes of items belonging to types of shelves and the rectangles assigned to the classes are packed by the classical bin packing algorithm NF. It is a straightforward idea to use other bin packing algorithms. The best known shelf algorithm was developed by Csirik and Woeginger in 1997, that algorithm uses the harmonic bin packing algorithm to pack the rectangles assigned to the classes.

Exercises

9.4-1 Suppose that the size of the items is bounded above by $1/3$. Prove that under this assumption the asymptotic competitive ratio of NF is $3/2$.

9.4-2 Suppose that the size of the items is bounded above by $1/3$. Prove Lemma 9.15 under this assumption.

9.4-3 Suppose that the sequence of items is given by a list $L_1L_2L_3$, where L_1 contains n items of size $1/2$, L_2 contains n items of size $1/3$, L_3 contains n items of size $1/4$. Which packing patterns can be used? Which patterns belong to the class T_2?

9.4-4 Consider the version of the strip packing problem where one can lengthen the rectangles keeping the area fixed. Consider the extension of NFS_r which lengthen the rectangles before the packing to have the same height as the shelf which is chosen to pack it. Prove that this algorithm is $2 + \frac{1}{r(1-r)}$-competitive.

9.5. On-line scheduling

The area of scheduling theory has a huge literature. The first result in on-line scheduling belongs to Graham, who analyzed in 1966 the List scheduling algorithm. We can say that despite the fact that Graham did not use the terminology which was developed in the area of the on-line algorithms, and he did not consider the algorithm as an on-line algorithm, he analyzed it as an approximation algorithm.

From the area of scheduling we only recall the definitions which are used in this chapter.

In a scheduling problem we have to find an optimal schedule of jobs. In the most fundamental model each job has a known processing time and to schedule the job we have to assign it to a machine and we have to give its starting time and a completion time, where the difference between the completion time and the starting time is the processing time. No machine may simultaneously run two jobs.

Concerning the machine environment three different models are considered. If the processing time of a job is the same for each machine then we call the machines identical machines. If each machine has a speed s_i, the jobs has a processing weight p_j and the processing time of job j on the i-th machine is p_j/s_i, then we call the machines related machines. If the processing time of job j is given by an arbitrary positive $P_j = (p_j(1), \dots, p_j(m))$ vector, where the processing time of the job on the i-th machine is $p_j(i)$, then we call the machines unrelated machines.

Many objective functions are considered for scheduling problems, here we only consider such models where the goal is the minimization of the makespan (the maximal completion time).

In the next subsection we define the two most fundamental on-line scheduling models, and in the following two subsections we consider these models in details.

9.5.1. On-line scheduling models

Probably the following models are the most fundamental examples of online machine scheduling problems.

LIST model

In this model we have a fixed number of machines denoted by $M_1, M_2, \ldots, M_m$ and the jobs arrive from a list. This means that the jobs and their processing times are revealed to the online algorithm one by one. When a job is revealed the online algorithm must irrevocably assign the job to a machine with a starting time and a completion time.

By the ***load*** of a machine, we mean the sum of the processing times of all jobs assigned to the machine. Since the objective function is to minimize the maximal completion time, it is enough to consider the schedules where the jobs are scheduled on the machines without idle time. For these schedules the maximal completion time is the load for each machine. Therefore this scheduling problem is reduced to a load balancing problem, the algorithm has to assign the jobs to the machines minimizing the maximum load, which is the makespan in this case.

9.8. Example. Consider the LIST model and two identical machines. Consider the following sequence of jobs where the jobs are given by their processing time: $I = \{(j_1 = 4, j_2 = 3, j_3 = 2, j_4 = 5)\}$. Then the on-line algorithm first receives job j_1 from the list, the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_1. Next the on-line algorithm receives job j_2 from the list, the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_2. Next the on-line algorithm receives job j_3 from the list, the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_2. Finally the on-line algorithm receives job j_4 from the list, the algorithm has to assign this job to one of the machines, suppose that the job is assigned to machine M_1. Then the loads on the machines are $4 + 5$ and $3 + 2$, and we can give a schedule where the maximal completion times on the machines are the loads: we can schedule the jobs on the first machine in the time intervals $(0, 4)$ and $(4, 9)$, and we can schedule the jobs on the second machine in the time intervals $(0, 3)$ and $(3, 5)$.

TIME model

In this model again there are a fixed number of machines. Each job has a processing time and a ***release time***. A job is revealed to the online algorithm at its release time. For each job the online algorithm must choose which machine it will run on and assign a start time. No machine may simultaneously run two jobs. Note that the algorithm is not required to immediately assign a job at its release time. However, if the online algorithm assigns a job at time t then it cannot use information about jobs released after time t and it cannot start the job before time t. The objective is to minimize the makespan.

9.9. Example. Consider the TIME model with two related machines. Let the first machine be M_1 with speed 1, and the second machine be M_2 with speed 2. Consider the following input $I = \{j_1 = (1, 0), j_2 = (1, 1), j_3 = (1, 1), j_4 = (1, 1)\}$, where the jobs are given by the (processing time, release time) pairs. Thus a job arrives at time 0 with processing time 1, the algorithm can start to process it on one of the machines or it can wait for jobs with larger processing time. Suppose that the algorithm waits till time $1/2$ and then it starts to process the job on machine M_1. The completion time of the job is $3/2$. At time 1 three further jobs arrive, at that time only M_2 can be used. Suppose that the algorithm starts to process job j_2 on this machine. At time $3/2$ both jobs are completed, suppose that the remaining jobs are started on machines M_1 and M_2 the completion times are $5/2$ and 2, thus the makespan achieved by the algorithm is $5/2$. Observe that an algorithm which starts the first job immediately at time 0 can make a better schedule with makespan 2. But it is important to note that in some cases it can be useful to wait for larger jobs before starting a job.

9.5.2. LIST model

The first algorithm in this model was developed by Graham. Algorithm LIST assigns each job to the machine where the actual load is minimal, if there are more machines with this property it uses the machine with the smallest index. This means that the algorithm tries to balance the loads on the machines. The competitive ratio of this algorithm is determined by the following theorem.

Theorem 9.18 *The competitive ratio of algorithm* LIST *is* $2-1/m$ *in the identical machines case.*

Proof. First we prove that the algorithm is $2-1/m$-competitive. Consider an arbitrary input sequence denote it by $\sigma = \{j_1, \dots, j_n\}$, denote the processing times by $p_1, \dots, p_n$. Consider the schedule produced by LIST. Let j_l be a job with maximal completion time. Investigate the starting time S_l of this job. Since LIST chooses the machine with minimal load thus the load was at least S_l on each of the machines when j_l was scheduled. Therefore we obtain that

$$S_l \le \frac{1}{m}\sum_{\substack{j=1\\ j\neq l}}^{n} p_j = \frac{1}{m}(\sum_{j=1}^{n} p_j - p_l) = \frac{1}{m}(\sum_{j=1}^{n} p_j) - \frac{1}{m}p_l \ .$$

This yields that

$$LIST(\sigma) = S_l + p_l \le \frac{1}{m}(\sum_{j=1}^{n} p_j) + \frac{m-1}{m}p_l \ .$$

On the other hand OPT also processes all of the jobs thus we obtain that $OPT(\sigma) \ge \frac{1}{m}(\sum_{j=1}^{n} p_j)$. Furthermore p_l is scheduled on one of the machines of OPT thus $OPT(\sigma) \ge p_l$. By these bounds we obtain that

$$LIST(\sigma) \le (1 + \frac{m-1}{m})OPT(\sigma),$$

which inequality proves that LIST is $2-1/m$-competitive.

Now we prove that the bound is tight. Consider the following input. It contains $m(m-1)$ jobs with processing time $1/m$ and one job with processing time 1. Then LIST assigns $m-1$ small jobs to each machine and the last large job is assigned to M_1. Therefore its makespan is $1+(m-1)/m$. On the other hand the optimal algorithm schedules the large job on M_1 and m small jobs on the other machines, and its makespan is 1. Thus the ratio of the makespans is $2-1/m$ which shows that the competitive ratio of LIST is at least $2-1/m$. ■

It is hard to imagine any other algorithm for the on-line case, but many other algorithms were developed. The competitive ratios of the better algorithms tend to smaller number than

2 as the number of machines tends to ∞. Most of these algorithms are based on the following idea. The jobs are scheduled keeping the load uniformly on most of the machines but in contrast with LIST the loads are kept low on some of the machines, keeping the possibility to use these machines for scheduling large jobs which may arrive later.

Below we consider the more general cases where the machines are not identical. LIST may perform very badly, the processing time of a job can be very large on the machine where the actual load is minimal. But we can easily change the greedy idea of LIST as follows. The extended algorithm is called GREEDY and it assigns the job to the machine where the load with the processing time of the job is minimal. If there are more machines which has minimal value then the algorithm chooses among them the machine where the processing time of the job is minimal, if there are more machines with this property the algorithm chooses among them the one with the smallest index.

9.10. Example. Consider the case of related machines where there are 3 machines M_1, M_2, M_3 and the speeds are $s_1 = s_2 = 1$, $s_2 = 3$. Suppose that the input is $I = \{j_1 = 2, j_2 = 1, j_3 = 1, j_4 = 3, j_5 = 2)\}$, where the jobs are defined by their processing weight. Then the load after the first job is $2/3$ on machine M_3 and 2 on the other machines, thus j_1 is assigned to M_3. The load after job j_2 is 1 on all of the machines, its processing time is minimal on machine M_3, thus GREEDY assigns it to M_3. The load after job j_3 is 1 on M_1 and M_2, and $4/3$ on M_3, thus the job is assigned to M_1. The load after job j_4 is 4 on M_1, 3 on M_2, and 2 on M_3, thus the job is assigned to M_3. Finally the load after job j_5 is 3 on M_1, 2 on and M_2, and $8/3$ on M_3, thus the job is assigned to M_2.

9.11. Example. Consider the unrelated machines case with two machines and the following input $I = \{j_1 = (1,2), j_2 = (1,2), j_3 = (1,3), j_4 = (1,3)\}$, where the jobs are defined by the vectors of processing times. The load after job j_1 is 1 on M_1 and 2 on M_2, thus the job is assigned to M_1. The load after job j_2 is 2 on M_1 and also on M_2, thus the job is assigned to M_1 because it has smaller processing time there. The load after job j_3 is 3 on M_1 and M_2, thus the job is assigned to M_1 because it has smaller processing time there. Finally the load after job j_4 is 4 on M_1 and 3 on M_2, thus the job is assigned to M_2.

The competitive ratio of the algorithm is determined by the following theorems.

Theorem 9.19 *The competitive ratio of algorithm* GREEDY *is m in the unrelated machines case.*

Proof. First we prove that the competitive ratio of the algorithm is at least m. Consider the following input sequence. Let $\varepsilon > 0$ be a small number. The sequence contains m jobs. The processing time of job j_1 is 1 on machine M_1, $1+\varepsilon$ on machine M_m, and ∞ on the other machines, $(p_1(1) = 1, p_1(i) = \infty, i = 2, \ldots, m-1, p_1(m) = 1+\varepsilon)$. For job j_i, $i = 2, \ldots, m$ the processing time is i on machine M_i, $1+\varepsilon$ on machine M_{i-1} and ∞ on the other machines $(p_j(j-1) = 1+\varepsilon$, $p_j(j) = j$, $p_j(i) = \infty$, if $i \neq j-1$ and $i \neq j)$.

Then job j_i is scheduled on M_i by GREEDY and the makespan is m. On the other hand the optimal offline algorithm schedules j_1 on M_m and j_i is scheduled on M_{i-1} for the other jobs thus the optimal makespan is $1+\varepsilon$. The ratio of the makespans is $m/(1+\varepsilon)$. This ratio tends to m, as ε tends to 0, and this proves that the competitive ratio of the algorithm is at least m.

Now we prove that the algorithm is m-competitive. Consider an arbitrary input sequence, denote the makespan in the optimal offline schedule by L^* and let $L(k)$ denote the maximal load in the schedule produced by GREEDY after scheduling the first k jobs. Since the processing time of the i-th job is at least $\min_j p_i(j)$, and the load is at most L^* on the machines in the offline optimal schedule, thus we obtain that $mL^* \geq \sum_{i=1}^{n} \min_j p_i(j)$.

We prove by induction that the inequality $L(k) \leq \sum_{i=1}^{k} \min_j p_i(j)$ is valid. Since the first job is assigned to the machine where its processing time is minimal, thus the statement is obviously true for $k = 1$. Let $1 \leq k < n$ be arbitrary and suppose that the statement is true for k. Consider the $k+1$-th job. Let M_l be the machine where the processing time of this job is minimal. If we assign the job to M_l then we obtain by the induction hypothesis that the load on this machines is at most $L(k) + p_{k+1}(l) \leq \sum_{i=1}^{k+1} \min_j p_i(j)$.

On the other hand the maximal load in the schedule produced by GREEDY can not be more than the maximal load in the case when the job is assigned to M_l, thus $L(k+1) \leq \sum_{i=1}^{k+1} \min_j p_i(j)$, which means that we proved the inequality for $k+1$.

Therefore we obtained that $mL^* \geq \sum_{i=1}^{n} \min_j p_i(j) \geq L(n)$, which yields that the algorithm is m-competitive. ■

To investigate the related machines case consider an arbitrary input. Let L and L^* denote the makespans achieved by GREEDY and OPT respectively. The analysis of the algorithm is based on the following lemmas which give bounds on the loads of the machines.

Lemma 9.20 *The load on the fastest machine is at least $L - L^*$.*

Proof. Consider the schedule produced by GREEDY. Consider a job J which causes the makespan (its completion time is maximal). If this job is scheduled on the fastest machine then the lemma immediately follows, the load on the fastest machine is L. Suppose that J is not scheduled on the fastest machine. The optimal maximal load is L^*, thus the processing time of J on the fastest machine is at most L^*. On the other hand the completion time of J is L, thus at the time when the job was scheduled the load was at least $(L - L^*)$ on the fastest machine, otherwise GREEDY would assign J to the fastest machine. ■

Lemma 9.21 *If the loads are at least l on all machines having speed at least v then the loads are at least $l - 4L^*$ on all machines having speed at least $v/2$.*

Proof. If $l < 4L^*$, then the statement is obviously valid. Suppose that $l \geq 4L^*$. Consider the jobs which are scheduled by GREEDY on the machines having speed at least v in the time interval $[l - 2L^*, l]$. The total processing weight of these jobs is at least $2L^*$ times the total speed of the machines having speed at least v. This yields that there exists such job among them which is assigned by OPT to a machine having speed smaller than v (otherwise the optimal offline makespan would be larger than L^*). Let J be such a job.

Since OPT schedules J on a machine having speed smaller than v, thus the processing weight of J is at most vL^*. This yields that the processing time of J is at most $2L^*$ on the machines having speed at least $v/2$. On the other hand GREEDY produces a schedule where the completion time of J is at least $l - 2L^*$, thus at the time when the job was scheduled the loads were at least $l - 4L^*$ on the machines having speed at most $v/2$ (otherwise GREEDY would assign J to one of these machines).

■

Now we can prove the following statement.

Theorem 9.22 *The competitive ratio of algorithm* GREEDY *is $\Theta(\log m)$ in the related machines case.*

Proof. First we prove that GREEDY is $O(\lg m)$-competitive. Consider an arbitrary input. Let L and L^* denote the makespans achieved by GREEDY and OPT respectively.

Let $v_{\max}$ be the speed of the fastest machine. Then by Lemma 9.20 the load on this machine is at least $L - L^*$. Then using Lemma 9.21 we obtain that the loads are at least $L - L^* - 4iL^*$ on the machines having speed at least $v_{\max}2^{-i}$. Therefore the loads are at least $L - (1 + 4\lceil \lg m \rceil)L^*$ on the machines having speed at least $v_{\max}/m$. Denote I the set of the machines having speed at most $v_{\max}/m$.

Denote W the sum of the processing weights of the jobs. OPT can find a schedule of the jobs which has maximal load L^*, and there are at most m machines having smaller speed than $v_{\max}/m$ thus

$$W \le L^* \sum_{i=1}^{m} v_i \le mL^* v_{\max}/m + L^* \sum_{i \notin I} v_i \le 2L^* \sum_{i \notin I} v_i \ .$$

On the other hand GREEDY schedules the same jobs thus the load on some machine not included in I is smaller than $2L^*$ in the schedule produced by GREEDY (otherwise we would obtain that the sum of the processing weights is greater than W).

Therefore we obtain that

$$L - (1 + 4\lceil \lg m \rceil)L^* \le 2L^* \ ,$$

which yields that $L \le 3 + 4\lceil \lg m \rceil)L^*$, which proves that GREEDY is $O(\lg m)$-competitive.

Now we prove that the competitive ratio of the algorithm is at least $\Omega(\lg m)$. Consider the following set of machines. G_0 contains one machine with speed 1, G_1 contains 2 machines with speed 1/2. For each $i = 1, 2, \ldots, k$, G_i contains machines with speed 2^{-i}, and G_i contains $|G_i| = \sum_{j=0}^{i-1} |G_j| 2^{i-j}$ machines. Observe that the number of jobs of size 2^{-i} which can be scheduled during time 1 is the same on the machines of G_i and on the machines of $G_0 \cup G_1 \ldots, \cup G_{i-1}$. It is easy to calculate that $|G_i| = 2^{2i-1}$, if $i \ge 1$, thus the number of machines is $1 + \frac{2}{3}(4^k - 1)$.

Consider the following input sequence. In the first phase $|G_k|$ jobs arrive having processing weight 2^{-k}, in the second phase $|G_{k-1}|$ jobs arrive having processing weight $2^{-(k-1)}$, in the i-th phase $|G_i|$ jobs arrive with processing weight 2^{-i}, and the sequence ends with the $k+1$-th phase, which contains one job with processing weight 1. An offline algorithm can schedule the jobs of the i-th phase on the machines of set G_{k+1-i} achieving maximal load 1, thus the optimal offline cost is at most 1.

Investigate the behaviour of algorithm GREEDY on this input. The jobs of the first phase can be scheduled on the machines of $G_0, \ldots, G_{k-1}$ during time 1, and it takes also time 1 to schedule these jobs on the machines of G_k. Thus GREEDY schedules these jobs on the machines of $G_0, \ldots, G_{k-1}$, and the loads are 1 on these machines after the first phase. Then the jobs of the second phase are scheduled on the machines of $G_0, \ldots, G_{k-2}$, the jobs of the third phase are scheduled on the machines of $G_0, \ldots, G_{k-3}$ and so on. Finally the jobs of the k-th and $k+1$-th phase are scheduled on the machine of set G_0. Thus the cost of GREEDY is $k+1$, (this is the load on the machine of set G_0). Since $k = \Omega(\lg m)$, thus we proved the required statement. ■

9.5.3. TIME model

In this model we only investigate one algorithm. The basic idea is to divide the jobs into groups by the release time and to use an optimal offline algorithm to schedule the jobs from the groups. This algorithm is called ***interval scheduling algorithm*** and we denote it by INTV. Let t_0 be the release time of the first job, and $i = 0$. Then the algorithm is defined by the following pseudocode:

```
INTV(I)

1  while not end of sequence
2      let $H_i$ be the set of the unscheduled jobs released till $t_i$
3      let $OFF_i$ be an optimal offline schedule of the jobs of $H_i$
4      schedule the jobs as it is determined by $OFF_i$ starting the schedule at $t_i$
5      let $q_i$ be the maximal completion time
6      if new job is released in time interval $(t_i, q_i]$ or the sequence is ended
7          then $t_{i+1} \leftarrow q_i$
7          else  let $t_{i+1}$ be the release time of the next job
8      $i \leftarrow i + 1$
```

9.12. Example. Consider two identical machines. Suppose that the sequence of jobs is $I = \{j_1 = (1,0), j_2 = (1/2,0), j_3 = (1/2,0), j_4 = (1,3/2), j_5 = (1,3/2), j_6 = (2,2)\}$, where the jobs are defined by the (processing time, release time) pairs. In the first iteration j_1, j_2, j_3 are scheduled, an optimal offline algorithm schedules j_1 on machine M_1 and j_2, j_3 on machine M_2, the jobs are completed at time 1. Then no new job released in the time interval $(0, 1]$ thus the algorithm waits for new job till time $3/2$. Then the second iteration starts j_4 and j_5 are scheduled on M_1 and M_2 respectively in the time interval $[3/2, 5/2)$. During this time interval j_6 released thus at time $5/2$ the next iteration starts and INTV schedules j_6 on M_1 in the time interval $[5/2, 9/2]$.

The following statement holds for the competitive ratio of algorithm INTV.

Theorem 9.23 *In the TIME model algorithm* INTV *is 2-competitive.*

Proof. Consider an arbitrary input and the schedule produced by INTV. Denote the number of iterations by i. Let $T_3 = t_{i+1} - t_i$, $T_2 = t_i - t_{i-1}$, $T_1 = t_{i-1}$ and let T_{OPT} denote the optimal offline cost. Then $T_2 \le T_{OPT}$. This inequality is obvious if $t_{i+1} \ne q_i$. If $t_{i+1} = q_i$, then the inequality holds because the optimal offline algorithm also has to schedule the jobs which are scheduled in the i-th iteration by INTV and INTV uses an optimal offline schedule in each iteration. On the other hand $T_1 + T_3 \le T_{OPT}$. To prove this inequality first observe that the release time is at least $T_1 = t_{i-1}$ for the jobs scheduled in the i-th iteration (otherwise the algorithm would schedule them in the $i - 1$-th iteration). Therefore the optimal algorithm also must schedule these jobs after time T_1. On the the other hand it takes at least time T_3 to process these jobs because INTV uses optimal offline algorithm in the iterations. The makespan of the schedule produced by INTV is $T_1 + T_2 + T_3$, and we have shown that $T_1 + T_2 + T_3 \le 2T_{OPT}$ thus we proved that the algorithm is 2-competitive. ■

Some other algorithms are also developed in the TIME model. Vestjens proved that the ***on-line*** **LPT** algorithm is 3/2-competitive. This algorithm schedules the longest unscheduled, released job at any time when some machine is available. The following lower bound for the possible competitive ratios of the on-line algorithms is also given by Vestjens.

Theorem 9.24 *The competitive ratio of any on-line algorithm is at least* 1.3473 *in the TIME model for minimizing the makespan.*

Proof. Let $\alpha = 0.3473$ be the solution of the equation $\alpha^3 - 3\alpha + 1 = 0$ which belongs to the interval $[1/3, 1/2]$. We prove that no on-line algorithm can have smaller competitive ratio than $1 + \alpha$. Consider an arbitrary on-line algorithm, denote it by ALG. Investigate the following input sequence.

At time 0 one job arrives with processing time 1. Let S_1 be the time when the algorithm starts to process the job on one of the machines. If $S_1 > \alpha$, then the sequence is ended and $\text{ALG}(I)/\text{OPT}(I) > 1 + \alpha$, which proves the statement. So we can suppose that $S_1 \leq \alpha$.

The release time of the next job is S_1 and its processing time is $\alpha/(1 - \alpha)$. Denote its starting time by S_2. If $S_2 \leq S_1 + 1 - \alpha/(1 - \alpha)$, then we end the sequence with $m - 1$ jobs having release time S_2, and processing time $1 + \alpha/(1 - \alpha) - S_2$. Then an optimal offline algorithm schedules the first two jobs on the same machine and the last $m - 1$ jobs on the other machines starting them at time S_2, thus its cost is $1 + \alpha/(1 - \alpha)$. On the other hand the on-line algorithm must schedule one of the last $m - 1$ jobs after the completion of the first or the second job thus $\text{ALG}(I) \geq 1 + 2\alpha/(1 - \alpha)$ in this case, which yields that the competitive ratio of the algorithm is at least $1 + \alpha$. Therefore we can suppose that $S_2 > S_1 + 1 - \alpha/(1 - \alpha)$.

Then at time $S_1 + 1 - \alpha/(1 - \alpha)$ further $m - 2$ jobs arrive having processing time $\alpha/(1 - \alpha)$ and one job having processing time $1 - \alpha/(1 - \alpha)$. The optimal offline algorithm schedules the second and the last jobs on the same machine the other jobs are scheduled alone on the other machines and the makespan of the schedule is $1 + S_1$. Since before time $S_1 + 1 - \alpha/(1 - \alpha)$ none of the last m jobs is started by ALG thus after this time ALG must schedule at least two jobs on one of the machines and the maximal completion time is at least $S_1 + 2 - \alpha/(1 - \alpha)$. Since $S_1 \leq \alpha$, thus the ratio $\text{OPT}(I)/\text{ALG}(I)$ is minimal if $S_1 = \alpha$ and in this case the ratio is $1 + \alpha$, which proves the theorem. ■

Exercises

9.5-1 Prove that the competitive ratio is at least $3/2$ for any on-line algorithm in the case of two identical machines.

9.5-2 Prove that LIST is not constant competitive in the unrelated machines case.

9.5-3 Prove that the modification of INTV which uses a c-approximation schedule (a schedule with at most c times more cost than the optimal cost) in each step instead of the optimal offline schedule is $2c$-competitive.

Problems

9-1. Paging problem

Consider the special case of the k-server problem where the distance between each pair of points is 1. (This problem is equivalent with the on-line paging problem.) Analyze the algorithm which serves the requests not having server on their place by the server which was used least recently. (This algorithm is equivalent with the LRU paging strategy.) Prove that the algorithm is k-competitive.

9-2.* ALARM2 *algorithm
Consider the following alarming algorithm for the data acknowledgement problem. ALARM2 is obtained from the general definition with the values $e_j = 1/|\sigma_j|$. Prove that the algorithm is not constant-competitive.

9-3. Bin packing lower bound
Prove, that no on-line algorithm can have smaller competitive ratio than $3/2$ using a sequence which contains items of size $1/7 + \varepsilon$, $1/3 + \varepsilon$, $1/2 + \varepsilon$, where ε is a small positive number.

9-4. Strip packing with modifiable rectangles
Consider the following version of the strip packing problem. In the new model the algorithms are allowed to lengthen the rectangles keeping the area fixed. Develop a 4-competitive algorithm for the solution of the problem.

9-5. On-line* LPT *algorithm
Consider the algorithm in the TIME model which starts the longest unscheduled released job at any time when a machine is available. This algorithm is called on-line LPT. Prove that the algorithm is $3/2$-competitive.

Chapter notes

More details about the results on on-line algorithms can be found in the books [7, 19].

The first results about the k-server problem (Theorems 9.1 and 9.2) are published by Manasse, McGeoch and Sleator in [38]. The presented algorithm for the line (Theorem 9.3) was developed by Chrobak, Karloff, Payne and Viswanathan (see [12]). Later Chrobak and Larmore in [10] extended the algorithm for trees. The first constant-competitive algorithm for the general problem was developed by Fiat, Rabani and Ravid (see [18]). The best known algorithm is based on the work function technique. The first work function algorithm for the problem was developed in [11] by Chrobak and Larmore. Koutsoupias and Papadimitriou proved in [33] that the work function algorithm is $2k - 1$-competitive.

The first mathematical model for the data acknowledgement problem and the first results (Theorems 9.5 and 9.6) are presented in [16] by Dooly, Goldman, and Scott. Albers and Bals considered a different objective function in [1]. Karlin Kenyon and Randall investigated randomized algorithms for the data acknowledgement problem in [31]. The Landlord algorithm was developed in [62] by Young. The detailed description of the results in the area of on-line routing can be found in the survey [36] written by Leonardi. The exponential algorithm for the load balancing model is investigated by Aspnes, Azar, Fiat, Plotkin and Waarts in [2]. The exponential algorithm for the throughput objective function is applied by Awerbuch, Azar and Plotkin in [4].

A detailed survey about the theory of on-line bin packing is written by Csirik and Woeginger (see [13]). The algorithms NF and FF are analyzed with competitive analysis by Johnson, Demers, Ullman, Garey and Graham in [27, 28], further results can be found in the PhD thesis of Johnson ([26]). Van Vliet applied the packing patterns to prove lower bounds for the possible competitive ratios in [60, 65]. For the on-line strip packing problem algorithm NFS_r was developed and analyzed by Baker and Schwarz in [6]. Later further shelf packing algorithms were developed, the best shelf packing algorithm for the strip packing problem was developed by Csirik and Woeginger in [14].

A detailed survey about the results in the area of on-line scheduling was written by Sgall ([51]). The first on-line result is the analysis of algorithm LIST, it was published in [22] by Graham. Many further algorithms were developed and analyzed for the identical machines case, the algorithm with smallest competitive ratio (tends to 1.9201 as the number of machines tends to ∞) was developed by Fleischer and Wahl in [20]. The lower bound for the competitive ratio of GREEDY in the related machines model was proven by Cho and Sahni in [9]. The upper bound, the related machines case and a more sophisticated exponential function based algorithm were presented by Aspnes, Azar, Fiat, Plotkin and Waarts in [2]. A summary of the further results about the applications of the exponential function technique in the area of on-line scheduling can be found in the paper of Azar ([5]). The interval algorithm presented in the TIME model and Theorem 9.23 are based on the results of Shmoys, Wein and Williamson (see [53]). A detailed description of the further results (on-line LPT, lower bounds) in the area TIME model can be found in the PhD thesis of Vestjens [66]. We only presented the most fundamental on-line scheduling models in the chapter, recently an interesting model was developed where the number of the machines is not fixed the algorithm is allowed to purchase machines, the model is investigated in the papers [25] and [17].

Problem *9-1.* is based on [55], Problem *9-2.* is based on [16], Problem *9-3.* is based on [61], Problem *9-4.* is based on [24] and Problem *9-5.* is based on [66].

10. Parallel Computations

11. Network Simulation

12. Systolic Systems

Systolic arrays probably constitute a perfect kind of special purpose computer. In their simplest appearance, they may provide only one operation, that is repeated over and over again. Yet, systolic arrays show an abundance of practice-oriented applications, mainly in fields dominated by iterative procedures: numerical mathematics, combinatorial optimisation, linear algebra, algorithmic graph theory, image and signal processing, speech and text processing, et cetera.

For a systolic array can be tailored to the structure of its one and only algorithm thus accurately! So that time and place of each executed operation are fixed once and for all. And communicating cells are permanently and directly connected, no switching required. The algorithm has in fact become hardwired. Systolic algorithms in this respect are considered to be ***hardware algorithms***.

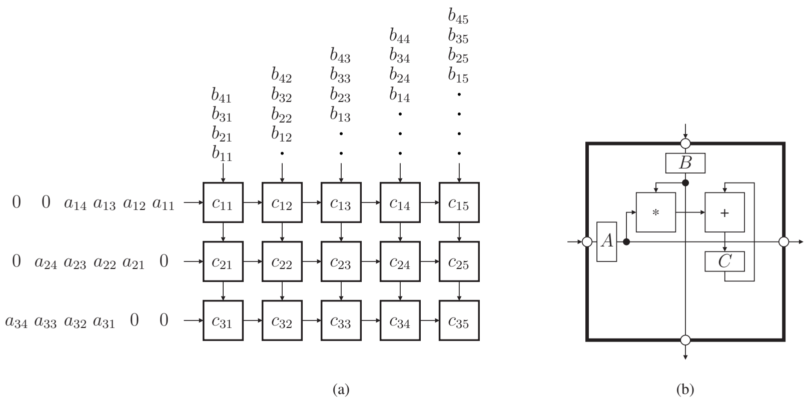
Please note that the term *systolic algorithms* usually does not refer to a set of concrete algorithms for solving a single specific computational problem, as for instance *sorting*. And this is quite in contrast to terms like *sorting algorithms*. Rather, systolic algorithms constitute a special style of specification, programming, and computation. So algorithms from many different areas of application can be *systolic* in style. But probably not all well-known algorithms from such an area might be suited to systolic computation.

Hence, this chapter does not intend to present *all* systolic algorithms, nor will it introduce even the most important systolic algorithms from any field of application. Instead, with a few simple but typical examples, we try to lay the foundations for the readers' general understanding of systolic algorithms.

The rest of this chapter is organised as follows: Section 12.1 shows some basic concepts of systolic systems by means of an introductory example. Section 12.2 explains how systolic arrays formally emerge from space-time transformations. Section 12.3 deals with input/output schemes. Section 12.4 is devoted to all aspects of control in systolic arrays. In section 12.5 we study the class of linear systolic arrays, raising further questions.

12.1. Basic concepts of systolic systems

The designation ***systolic*** follows from the operational principle of the systolic architecture. The systolic style is characterised by an intensive application of both *pipelining* and *pa-*



0 0 a_{14} a_{13} a_{12} a_{11}
0 a_{24} a_{23} a_{22} a_{21} 0
a_{34} a_{33} a_{32} a_{31} 0 0
b_{41} b_{31} b_{21} b_{11}
b_{42} b_{32} b_{22} b_{12}
b_{43} b_{33} b_{23} b_{13}
b_{44} b_{34} b_{24} b_{14}
b_{45} b_{35} b_{25} b_{15}
c_{11} c_{12} c_{13} c_{14} c_{15}
c_{21} c_{22} c_{23} c_{24} c_{25}
c_{31} c_{32} c_{33} c_{34} c_{35}
(a)
B
$*$
$+$
A
C
(b)

Figure 12.1. Rectangular systolic array for matrix product. **(a)** Array structure and input scheme. **(b)** Cell structure.

rallelism, controlled by a global and completely synchronous clock. ***Data streams*** pulsate rhythmically through the communication network, like streams of blood are driven from the heart through the veins of the body. Here, pipelining is not constrained to a single space axis but concerns *all data streams* possibly *moving in different directions* and *intersecting* in the cells of the systolic array.

A ***systolic system*** typically consists of a *host computer*, and the actual *systolic array*. Conceptionally, the host computer is of minor importance, just controlling the operation of the systolic array and supplying the data. The ***systolic array*** can be understood as a specialised network of cells rapidly performing data-intensive computations, supported by *massive parallelism*. A ***systolic algorithm*** is the program collaboratively executed by the cells of a systolic array.

Systolic arrays may appear very differently, but usually share a couple of key features: discrete time scheme, synchronous operation, regular (frequently two-dimensional) geometric layout, communication limited to directly neighbouring cells, and spartan control mechanisms.

In this section, we explain fundamental phenomena in context of systolic arrays, driven by a running example. A computational problem usually allows several solutions, each implemented by a specific systolic array. Among these, the most attractive designs (in whatever respect) may be very complex. Note, however, that in this educational text we are less interested in advanced solutions, but strive to present important concepts compactly and intuitively.

12.1.1. An introductory example: matrix product

Figure 12.1 shows a ***rectangular*** systolic array consisting of 15 ***cells*** for multiplying a $3 \times N$ matrix A by an $N \times 5$ matrix B. The *parameter* N is not reflected in the *structure* of this particular systolic array, but in the *input scheme* and the *running time* of the algorithm.

The input scheme depicted is based on the special choice of parameter $N = 4$. Therefore, Figure 12.1 gives a solution to the following problem instance:

$$A \cdot B = C \ ,$$

where

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix},$$

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \end{pmatrix},$$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \end{pmatrix},$$

and

$$c_{ij} = \sum_{k=1}^{4} a_{ik} \cdot b_{kj} \qquad (1 \le i \le 3, 1 \le j \le 5) \ .$$

The cells of the systolic array can exchange data through ***links***, drawn as arrows between the cells in Figure 12.1(a). ***Boundary cells*** of the systolic array can also communicate with the *outside world*. All cells of the systolic array share a common ***connection pattern*** for communicating with their environment. The completely regular ***structure*** of the systolic array (placement and connection pattern of the cells) induces *regular data flows* along all connecting directions.

Figure 12.1(b) shows the ***internal structure*** of a cell. We find a *multiplier*, an *adder*, three *registers*, and four *ports*, plus some wiring between these units. Each ***port*** represents an interface to some external link that is attached to the cell. All our cells are of the same structure.

Each of the registers A, B, C can store a single data item. The designations of the registers are suggestive here, but arbitrary in principle. Registers A and B get their values from ***input ports***, shown in Figure 12.1(b) as small circles on the left resp. upper border of the cell.

The current values of registers A and B are used as operands of the multiplier and, at the same time, are passed through ***output ports*** of the cell, see the circles on the right resp. lower border. The result of the multiplication is supplied to the adder, with the second operand originating from register C. The result of the addition eventually overwrites the past value of register C.

12.1.2. Problem parameters and array parameters

The 15 cells of the systolic array are organised as a rectangular pattern of three rows by five columns, exactly as with matrix C. Also, these dimensions directly correspond to the number of rows of matrix A and the number of columns of matrix B. The ***size of the systolic***

array, therefore, *corresponds* to the *size of some data structures* for the problem to solve. If we had to multiply an $N_1 \times N_3$ matrix A by an $N_3 \times N_2$ matrix B in the general case, then we would need a systolic array with N_1 rows and N_2 columns.

The quantities N_1, N_2, N_3 are parameters of the problem to solve, because the number of operations to perform depends on each of them; they are thus ***problem parameters***. The size of the systolic array, in contrast, depends on the quantities N_1 and N_2, only. For this reason, N_1 and N_2 become also ***array parameters***, for this particular systolic array, whereas N_3 is *not* an array parameter.

Remark. For matrix product, we will see another systolic array in section 12.2, with dimensions dependent on all three problem parameters N_1, N_2, N_3.

An $N_1 \times N_2$ systolic array as shown in Figure 12.1 would also permit to multiply an $M_1 \times M_3$ matrix A by an $M_3 \times M_2$ matrix B, where $M_1 \leq N_1$ and $M_2 \leq N_2$. This is important if we intend to use the same systolic array for the multiplication of matrices of varying dimensions. Then we would operate on a properly dimensioned rectangular subarray, only, consisting of M_1 rows and M_2 columns, and located, for instance, in the upper left corner of the complete array. The remaining cells would also work, but without any contribution to the solution of the whole problem; they should do no harm, of course.

12.1.3. Space coordinates

Now let's assume that we want to assign unique ***space coordinates*** to each cell of a systolic array, for characterising the geometric position of the cell relative to the whole array. In a *rectangular* systolic array, we simply can use the respective row and column numbers, for instance. The cell marked with c_{11} in Figure 12.1 thus would get the coordinates $(1, 1)$, the cell marked with c_{12} would get the coordinates $(1, 2)$, cell c_{21} would get $(2, 1)$, and so on. For the remainder of this section, we take space coordinates constructed in such a way for granted.

In principle it does not matter where the coordinate origin lies, where the axes are pointing to, which direction in space corresponds to the first coordinate, and which to the second. In the system presented above, the order of the coordinates has been chosen corresponding to the designation of the matrix components. Thus, the first coordinate stands for the rows numbered top to bottom from position 1, the second component stands for the columns numbered left to right, also from position 1.

Of course, we could have made a completely different choice for the coordinate system. But the presented system perfectly matches our particular systolic array: the indices of a matrix element c_{ij} computed in a cell agree with the coordinates of this cell. The entered rows of the matrix A carry the same number as the first coordinate of the cells they pass; correspondingly for the second coordinate, concerning the columns of the matrix B. All links (and thus all passing data flows) are in parallel to some axis, and towards ascending coordinates.

It is not always so clear how expressive space coordinates can be determined; we refer to the systolic array from Figure 12.3(a) as an example. But whatsoever the coordinate system is chosen: it is important that the regular structure of the systolic array is obviously reflected in the coordinates of the cells. Therefore, almost always integral coordinates are used. Moreover, the coordinates of cells with minimum Euclidean distance should differ in one component, only, and then with distance 1.

12.1.4. Serialising generic operators

Each active cell (i, j) from Figure 12.1 computes exactly the element c_{ij} of the result matrix C. Therefore, the cell must evaluate the ***dot product***

$$\sum_{k=1}^{4} a_{ik} \cdot b_{kj} \; .$$

This is done iteratively: in each step, a product $a_{ik} \cdot b_{kj}$ is calculated and added to the current partial sum for c_{ij}. Obviously, the partial sum has to be *cleared*—or set to another initial value, if required—before starting the accumulation. Inspired by the classical notation of imperative programming languages, the general proceeding could be specified in pseudocode as follows:

```
MATRIX-PRODUCT($N_1, N_2, N_3$)
1  for $i \leftarrow 1$ to $N_1$
2     do for $j \leftarrow 1$ to $N_2$
3           do $c(i, j) \leftarrow 0$
4              for $k \leftarrow 1$ to $N_3$
5                 do $c(i, j) \leftarrow c(i, j) + a(i, k) \cdot b(k, j)$
6  return $C$
```

If $N_1 = N_2 = N_3 = N$, we have to perform N^3 multiplications, additions, and assignments, each. Hence the *running time* of this algorithm is of order $\Theta(N^3)$ for any sequential processor.

The sum operator $\sum$ is one of the so-called ***generic operators***, that combine an arbitrary number of operands. In the systolic array from Figure 12.1, all additions contributing to a particular sum are performed in the same cell. However, there are plenty of examples where the individual operations of a generic operator are spread over several cells—see, for instance, the systolic array from Figure 12.3.

Remark. Further examples of generic operators are: product, minimum, maximum, as well as the Boolean operators AND, OR, and EXCLUSIVE OR.

Thus, generic operators usually have to be ***serialised*** before the calculations to perform can be assigned to the cells of the systolic array. Since the distribution of the individual operations to the cells is not unique, generic operators generally must be dealt with in another way than simple operators with fixed arity, as for instance the dyadic addition.

12.1.5. Assignment-free notation

Instead of using an imperative style as in algorithm MATRIX-PRODUCT, we better describe systolic programs by an ***assignment-free notation*** which is based on an *equational calculus*. Thus we avoid *side effects* and are able to *directly express parallelism*. For instance, we may be bothered about the reuse of the program variable $c(i, j)$ from algorithm MATRIX-PRODUCT. So, we replace $c(i, j)$ with a sequence of ***instances*** $c(i, j, k)$, that stand for the successive states of $c(i, j)$. This approach yields a so-called ***recurrence equation***. We are now able to state the general matrix product from algorithm MATRIX-PRODUCT by the following assignment-free expressions:

$$
\begin{array}{ll}
\textit{input operations} & \\
c(i,j,0) = 0 & 1 \le i \le N_1, 1 \le j \le N_2 . \\
& \\
\textit{calculations} & \\
c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 . \\
& \\
\textit{output operations} & \\
c_{ij} = c(i,j,N_3) & 1 \le i \le N_1, 1 \le j \le N_2 .
\end{array}
\tag{12.1}
$$

System (12.1) explicitly describes the fine structure of the executed systolic algorithm. The first equation specifies all ***input data***, the third equation all ***output data***. The systolic array implements these equations by ***input/output operations***. Only the second equation corresponds to real calculations.

Each equation of the system is accompanied, on the right side, by a ***quantification***. The quantification states the set of values the ***iteration variables*** i and j (and, for the second equation, also k) should take. Such a set is called a ***domain***. The iteration variables i, j, k of the second equation can be combined in an ***iteration vector*** (i, j, k). For the input/output equations, the iteration vector would consist of the components i and j, only. To get a closed representation, we augment this vector by a third component k, that takes a fixed value. Inputs then are characterised by $k = 0$, outputs by $k = N_3$. Overall we get the following system:

$$
\begin{array}{ll}
\textit{input operations} & \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0 . \\
& \\
\textit{calculations} & \\
c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 . \\
& \\
\textit{output operations} & \\
c_{ij} = c(i,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3 .
\end{array}
\tag{12.2}
$$

Note that although the domains for the input/output equations now are formally also of dimension 3, as a matter of fact they are only two-dimensional in the classical geometric sense.

12.1.6. Elementary operations

From equations as in system (12.2), we directly can infer the atomic entities to perform in the cells of the systolic array. We find these operations by instantiating each equation of the system with all points of the respective domain. If an equation contains several suboperations corresponding to one point of the domain, these are seen as a ***compound operation***, and are always processed together by the same cell in one working cycle.

In the second equation of system (12.2), for instance, we find the multiplication

$a(i,k) \cdot b(k,j)$ and the successive addition $c(i,j,k) = c(i,j,k-1) + \cdots$. The corresponding ***elementary operations***—multiplication and addition—are indeed executed together as a *multiply-add* compound operation by the cell of the systolic array shown in Figure 12.1(b).

Now we can assign a designation to each elementary operation, also called *coordinates*. A straight-forward method to define suitable coordinates is provided by the iteration vectors (i,j,k) used in the quantifications.

Applying this concept to system (12.1), we can for instance assign the tuple of coordinates (i,j,k) to the calculation $c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j)$. The same tuple (i,j,k) is assigned to the input operation $c(i,j,k) = 0$, but with setting $k = 0$. By the way: all domains are disjoint in this example.

If we always use the iteration vectors as designations for the calculations and the input/output operations, there is no further need to distinguish between coordinates and iteration vectors. Note, however, that this decision also mandates that all operations belonging to a certain point of the domain together constitute a compound operation—even when they appear in different equations and possibly are not related. For simplicity, we always use the iteration vectors as coordinates in the sequel.

12.1.7. Discrete timesteps

The various elementary operations always happen in ***discrete timesteps*** in the systolic cells. All these timesteps driving a systolic array are of equal duration. Moreover, all cells of a systolic array work completely ***synchronous***, i.e., they all start and finish their respective communication and calculation steps at the same time. Successive timesteps controlling a cell seamlessly follow each other.

Remark. But haven't we learned from Albert Einstein that strict simultaneity is physically impossible? Indeed, all we need here are cells that operate almost simultaneously. Technically this is guaranteed by providing to all systolic cells a common ***clock signal*** that switches all registers of the array. Within the bounds of the usually achievable accuracy, the communication between the cells happens sufficiently synchronised, and thus no loss of data occurs concerning send and receive operations. Therefore, it should be justified to assume a conceptional simultaneity for theoretical reasoning.

Now we can slice the physical time into units of a timestep, and number the timesteps consecutively. The origin on the time axis can be arbitrarily chosen, since time is running synchronously for all cells. A reasonable decision would be to take $t = 0$ as the time of the first input in any cell. Under this regime, the elementary compound operation of system (12.1) designated by (i,j,k) would be executed at time $i + j + k - 3$. On the other hand, it would be evenly justified to assign the time $i + j + k$ to the coordinates (i,j,k); because this change would only induce a global time shift by three time units.

So let us assume for the following that the execution of an instance (i,j,k) starts at time $i + j + k$. The first calculation in our example then happens at time $t = 3$, the last at time $t = N_1 + N_2 + N_3$. The *running time* thus amounts to $N_1 + N_2 + N_3 - 2$ timesteps.

12.1.8. External and internal communication

Normally, the data needed for calculation by the systolic array initially are not yet located inside the cells of the array. Rather, they must be infused into the array from the ***outside***

world. The outside world in this case is a ***host computer***, usually a *scalar control processor* accessing a central *data storage*. The control processor, at the right time, fetches the necessary data from the storage, passes them to the systolic array in a suitable way, and eventually writes back the calculated results into the storage.

Each cell (i, j) must access the operands a_{ik} and b_{kj} during the timestep concerning index value k. But only the cells of the leftmost column of the systolic array from Figure 12.1 get the items of the matrix A directly as *input data* from the outside world. All other cells must be provided with the required values a_{ik} from a neighbouring cell. This is done via the horizontal *links* between neighbouring cells, see Figure 12.1(a). The item a_{ik} successively passes the cells $(i, 1)$, $(i, 2)$, $\ldots$, (i, N_2). Correspondingly, the value b_{kj} enters the array at cell $(1, j)$, and then flows through the vertical links, reaching the cells $(2, j)$, $(3, j)$, $\ldots$ up to cell (N_1, j). An arrowhead in the Figure shows in which *direction* the link is oriented.

Frequently, it is considered problematic to transmit a value over large distances within a single *timestep*, in a distributed or parallel architecture. Now suppose that, in our example, cell (i, j) got the value a_{ik} during timestep t from cell $(i, j-1)$, or from the outside world. For the reasons described above, a_{ik} is not passed from cell (i, j) to cell $(i, j+1)$ in the same timestep t, but one timestep later, i.e., at time $t+1$. This also holds for the values b_{kj}. The ***delay*** is visualised in the detail drawing of the cell from Figure 12.1(b): input data flowing through a cell always pass one register, and each passed register induces a delay of exactly one timestep.

Remark. For systolic architectures, it is mandatory that any path between two cells contains at least one register—even when forwarding data to a neighbouring cell, only. All registers in the cells are synchronously switched by the global clock signal of the systolic array. This results in the characteristic rhythmical traffic on all links of the systolic array. Because of the analogy with pulsating veins, the medical term *systole* has been reused for the name of the concept.

To elucidate the delayed forwarding of values, we augment system (12.1) with further equations. Repeatedly *used* values like a_{ik} are represented by separate *instances*, one for each access. The result of this proceeding—that is very characteristic for the design of systolic algorithms—is shown as system (12.3).

$$
\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k) = a_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3\ , \\
b(i,j,k) = b_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3\ , \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0\ . \\
& \\
\textit{calculations and forwarding} & \\
a(i,j,k) = a(i,j-1,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ , \\
b(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ , \\
c(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ . \\
\quad + a(i,j-1,k) \cdot b(i-1,j,k) & \\
& \\
\textit{output operations} & \\
c_{ij} = c(i,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3\ .
\end{array}
\tag{12.3}
$$

Each of the partial sums $c(i, j, k)$ in the progressive evaluation of c_{ij} is calculated in a certain timestep, and then used only once, namely in the next timestep. Therefore, cell (i, j) must provide a register (named C in Figure 12.1(b)) where the value of $c(i, j, k)$ can be stored for one timestep. Once the old value is no longer needed, the register holding $c(i, j, k)$ can be overwritten with the new value $c(i, j, k + 1)$. When eventually the dot product is completed, the register contains the value $c(i, j, N_3)$, that is the final result c_{ij}. Before performing any computation, the register has to be ***cleared***, i.e., preloaded with a zero value—or any other desired value.

In contrast, there is no need to store the values a_{ik} and b_{kj} permanently in cell (i, j). As we can learn from Figure 12.1(a), each row of the matrix A is delayed by one timestep with respect to the preceding row. And so are the columns of the matrix B. Thus the values $a(i, j - 1, k)$ and $b(i - 1, j, k)$ arrive at cell (i, j) exactly when the calculation of $c(i, j, k)$ is due. They are put to the registers A resp. B, then immediately fetched from there for the multiplication, and in the same cycle forwarded to the neighbouring cells. The values a_{ik} and b_{kj} are of no further use for cell (i, j) after they have been multiplied, and need not be stored there any longer. So A and B are overwritten with new values during the next timestep.

It should be obvious from this exposition that we urgently need to make economic use of the memory contained in a cell. Any calculation and any communication must be coordinated in space and time in such a way that storing of values is limited to the shortest-possible time interval. This goal can be achieved by immediately using and forwarding the received values. Besides the overall structure of the systolic array, choosing an appropriate *input/output scheme* and placing the corresponding number of *delays* in the cells essentially facilitates the desired coordination. Figure 12.1(b) in this respect shows the smallest possible delay by one timestep.

Geometrically, the input input scheme of the example resulted from *skewing* the matrices A and B. Thereby some places in the ***input streams*** for matrix A became vacant and had to be filled with zero values; otherwise, the calculation of the c_{ij} would have been garbled. The input streams in length depend on the *problem parameter* N_3.

As can been seen in Figure 12.1, the items of matrix C are calculated ***stationary***, i.e., all additions contributing to an item c_{ij} happen in the same cell. ***Stationary variables*** don't move at all during the calculation in the systolic array. Stationary results eventually must be forwarded to a ***border*** of the array in a supplementary action for getting delivered to the outside world. Moreover, it is necessary to initialise the register for item c_{ij}. Performing these extra tasks requires a high expenditure of runtime and hardware. We will further study this problem in section 12.4.

12.1.9. Pipelining

The characteristic operating style with globally synchronised discrete timesteps of equal duration and the strict separation in time of the cells by registers suggest systolic arrays to be special cases of *pipelined* systems. Here, the registers of the cells correspond to the well-known *pipeline registers*. However, classical pipelines come as linear structures, only, whereas systolic arrays frequently extend into more spatial dimensions—as visible in our example. A *multi-dimensional systolic array* can be regarded as a set of interconnected linear pipelines, with some justification. Hence it should be apparent that basic properties of one-dimensional pipelining also apply to multi-dimensional systolic arrays.

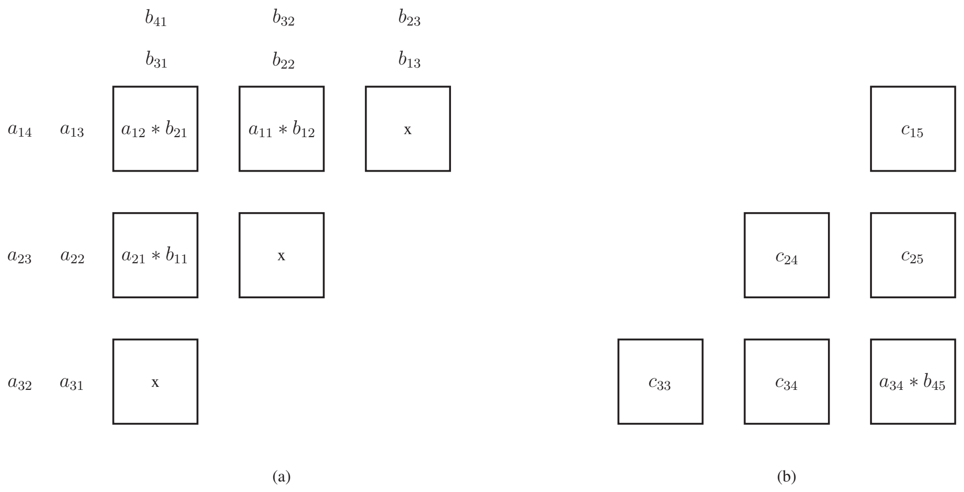


Figure 12.2. Two *snapshots* for the systolic array from Figure 12.1.

A typical effect of pipelining is the reduced ***utilisation*** at startup and during shut-down of the operation. Initially, the pipe is empty, no pipeline stage active. Then, the first stage receives data and starts working; all other stages are still idle. During the next timestep, the first stage passes data to the second stage and itself receives new data; only these two stages do some work. More and more stages become active until all stages process data in every timestep; the pipeline is now fully utilised for the first time. After a series of timesteps at maximum load, with duration dependent on the length of the data stream, the input sequence ceases; the first stage of the pipeline therefore runs out of work. In the next timestep, the second stage stops working, too. And so on, until eventually all stages have been fallen asleep again. Phases of reduced activity diminish the average performance of the whole pipeline, and the relative contribution of this drop in productivity is all the worse, the more stages the pipeline has in relation to the length of the data stream.

We now study this phenomenon to some depth by analysing the two-dimensional systolic array from Figure 12.1. As expected, we find a lot of idling cells when starting or finishing the calculation. In the first timestep, only cell $(1, 1)$ performs some useful work; all other cells in fact do calculations that work like null operations—and that's what they are supposed to do in this phase. In the second timestep, cells $(1, 2)$ and $(2, 1)$ come to real work, see Figure 12.2(a). Data is flooding the array until eventually all cells are doing work. After the last true data item has left cell $(1, 1)$, the latter is no longer contributing to the calculation but merely reproduces the finished value of c_{11}. Step by step, more and more cells drop off. Finally, only cell (N_1, N_2) makes a last necessary computation step; Figure 12.2(b) shows this concluding timestep.

Exercises

12.1-1 What must be changed in the input scheme from Figure 12.1(a) to multiply a 2×6 matrix by a 6×3 matrix on the same systolic array? Could the calculations be organised

such that the result matrix would emerge in the lower right corner of the systolic array?
12.1-2 Why is it necessary to clear spare slots in the input streams for matrix A, as shown in Figure 12.1? Why haven't we done the same for matrix B also?
12.1-3 If the systolic array from Figure 12.1 should be interpreted as a pipeline: how many stages would you suggest to adequately describe the behaviour?

12.2. Space-time transformation and systolic arrays

Although the approach taken in the preceding section should be sufficient for a basic understanding of the topic, we have to work harder to describe and judge the properties of systolic arrays in a quantitative and precise way. In particular the solution of *parametric problems* requires a solid mathematical framework. So, in this section, we study central concepts of a formal theory on *uniform algorithms*, based on *linear transformations*.

12.2.1. Further example: matrix product without stationary variables

System (12.3) can be computed by a multitude of other systolic arrays, besides that from Figure 12.1. In Figure 12.3, for example, we see such an alternative systolic array. Whereas the same function is evaluated by both architectures, the appearance of the array from Figure 12.3 is very different:

- The number of cells now is considerably larger, altogether 36, instead of 15.
- The shape of the array is ***hexagonal***, instead of rectangular.
- Each cell now has three input ports and three output ports.
- The input scheme is clearly different from that of Figure 12.1(a).
- And finally: the matrix C here also flows through the whole array.

The cell structure from Figure 12.3(b) at first view does not appear essentially distinguished from that in Figure 12.1(b). But the differences matter: there are no *cyclic paths* in the new cell, thus *stationary variables* can no longer appear. Instead, the cell is provided with three input ports and three output ports, passing items of all three matrices through the cell. The direction of communication at the ports on the right and left borders of the cell has changed, as well as the assignment of the matrices to the ports.

12.2.2. The space-time transformation as a global view

How system (12.3) is related to Figure 12.3? No doubt that you were able to fully understand the operation of the systolic array from Section 12.1 without any special aid. But for the present example this is considerably more difficult—so now you may be sufficiently motivated for the use of a mathematical formalism.

We can assign two fundamental measures to each elementary operation of an algorithm for describing the execution in the systolic array: the time *when* the operation is performed, and the position of the cell *where* the operation is performed. As will become clear in the sequel, after fixing the so-called *space-time transformation* there are hardly any degrees of freedom left for further design: practically all features of the intended systolic array strictly follow from the chosen space-time transformation.

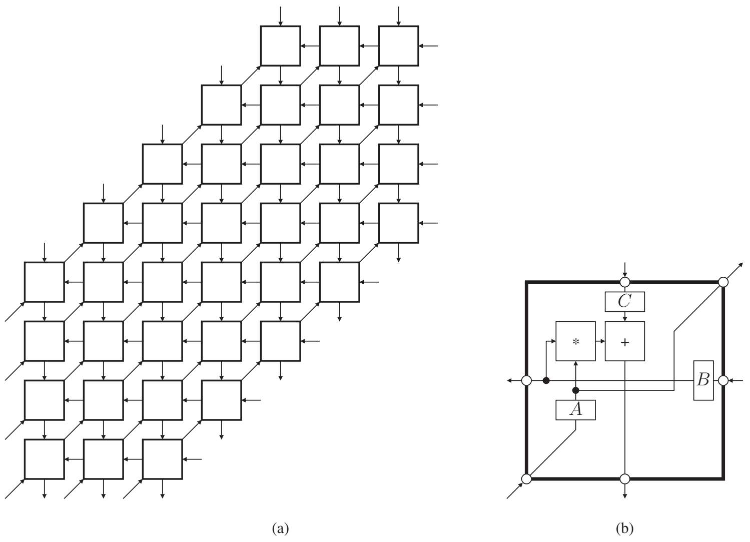


Figure 12.3. Hexagonal systolic array for matrix product. **(a)** Array structure and principle of the data input/output. **(b)** Cell structure.

As for the systolic array from Figure 12.1, the execution of an instance (i, j, k) in the systolic array from Figure 12.3 happens at time $t = i+j+k$. We can represent this expression as the dot product of a ***time vector***

$$\pi = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \tag{12.4}$$

by the iteration vector

$$v = \begin{pmatrix} i & j & k \end{pmatrix} , \tag{12.5}$$

hence

$$t = \pi \cdot v ; \tag{12.6}$$

so in this case

$$t = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = i + j + k . \tag{12.7}$$

The space coordinates $z = (x, y)$ of the executed operations in the example from Figure 12.1 can be inferred as $z = (i, j)$ from the iteration vector $v = (i, j, k)$ according to our decision in Section 12.1.3. The chosen map is a ***projection*** of the space $\mathbb{R}^3$ along the k axis. This linear map can be described by a ***projection matrix***

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} . \tag{12.8}$$

To find the space coordinates, we multiply the projection matrix P by the iteration vector

v, written as

$$z = P \cdot v \,. \tag{12.9}$$

The ***projection direction*** can be represented by any vector u perpendicular to all rows of the projection matrix,

$$P \cdot u = \vec{0} \,. \tag{12.10}$$

For the projection matrix P from (12.8), one of the possible ***projection vectors*** would be $u = (0, 0, 1)$.

Projections are very popular for describing the space coordinates when designing a systolic array. Also in our example from Figure 12.3(a), the space coordinates are generated by projecting the iteration vector. Here, a feasible projection matrix is given by

$$P = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} . \tag{12.11}$$

A corresponding projection vector would be $u = (1, 1, 1)$.

We can combine the projection matrix and the time vector in a matrix T, that fully describes the ***space-time transformation***,

$$\begin{pmatrix} z \\ t \end{pmatrix} = \begin{pmatrix} P \\ \pi \end{pmatrix} \cdot v = T \cdot v \,. \tag{12.12}$$

The first and second rows of T are constituted by the projection matrix P, the third row by the time vector π.

For the example from Figure 12.1, the matrix T giving the space-time transformation reads as

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} ; \tag{12.13}$$

for the example from Figure 12.3 we have

$$T = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} . \tag{12.14}$$

Space-time transformations may be understood as a *global view* to the systolic system. Applying a space-time transformation—that is linear, here, and described by a matrix T—to a system of recurrence equations directly yields the external features of the systolic array, i.e., its ***architecture***—consisting of space coordinates, connection pattern, and cell structure.

Remark. Instead of purely linear maps, we alternatively may consider general affine maps, additionally providing a translative component, $T \cdot v + h$. Though as long as we treat all iteration vectors with a common space-time transformation, affine maps are not really required.

12.2.3. Parametric space coordinates

If the domains are numerically given and contain few points in particular, we can easily calculate the concrete set of space coordinates via equation (12.9). But when the domains are specified parametrically as in system (12.3), the positions of the cells must be determined

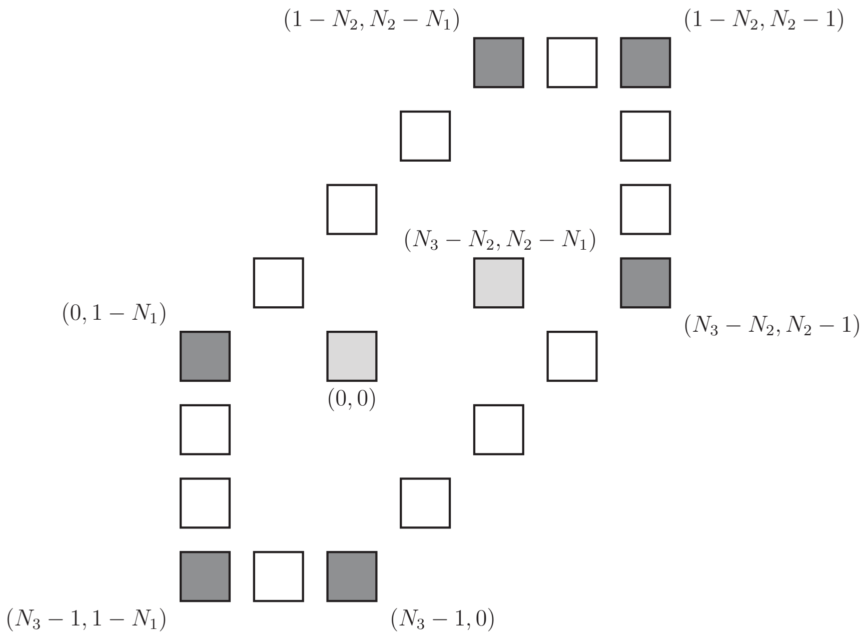


Figure 12.4. Image of a rectangular domain under projection. Most interior points have been suppressed for clarity. Images of previous vertex points are shaded.

by *symbolic evaluation*. The following explanation especially dwells on this problem.

Suppose that each cell of the systolic array is represented geometrically by a point with space coordinates $z = (x, y)$ in the two-dimensional space $\mathbb{R}^2$. From each iteration vector v of the domain S, by equation (12.9) we get the space coordinates z of a certain processor, $z = P \cdot v$: the operations denoted by v are projected onto cell z. The set $P(S) = \{P \cdot v : v \in S\}$ of space coordinates states the positions of all cells in the systolic array necessary for correct operation.

To our advantage, we normally use domains that can be described as the set of all integer points inside a convex region, here a subset of $\mathbb{R}^3$—called ***dense convex domains***. The convex hull of such a domain with a finite number of domain points is a *polytope*, with domain points as vertices. Polytopes map to polytopes again by arbitrary linear transformations. Now we can make use of the fact that each projection is a linear transformation. Vertices of the destination polytope then are images of vertices of the source polytope.

Remark. But not all vertices of a source polytope need to be projected to vertices of the destination polytope, see for instance Figure 12.4.

When projected by an integer matrix P, the *lattice* $\mathbf{Z}^3$ maps to the lattice $\mathbf{Z}^2$ if P can be extended by an integer time vector π to a *unimodular* space-time matrix T. Practically any dense convex domain, apart from some exceptions irrelevant to usual applications, thereby maps to another dense convex set of space coordinates, that is completely characterised by the vertices of the hull polytope. To determine the *shape* and the *size* of the systolic array, it is therefore sufficient to apply the matrix P to the vertices of the convex hull of S.

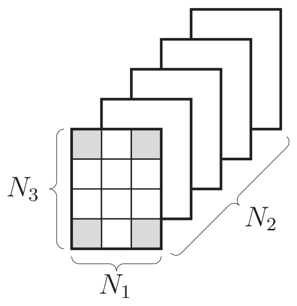


Figure 12.5. Partitioning of the space coordinates.

Remark. Any square integer matrix with determinant ±1 is called ***unimodular***. Unimodular matrices have unimodular inverses.

We apply this method to the integer domain

$$S = [1, N_1] \times [1, N_2] \times [1, N_3] \tag{12.15}$$

from system (12.3). The vertices of the convex hull here are

$$\begin{aligned} &(1,1,1), (N_1,1,1), (1,N_2,1), (1,1,N_3), \\ &(1,N_2,N_3), (N_1,1,N_3), (N_1,N_2,1), (N_1,N_2,N_3) \ . \end{aligned} \tag{12.16}$$

For the projection matrix P from (12.11), the vertices of the corresponding image have the positions

$$\begin{aligned} &(N_3-1,0), (N_3-1,1-N_1), (0,1-N_1), \\ &(1-N_2,N_2-N_1), (1-N_2,N_2-1), (N_3-N_2,N_2-N_1) \ . \end{aligned} \tag{12.17}$$

Since S has eight vertices, but the image $P(S)$ only six, it is obvious that two vertices of S have become *interior points* of the image, and thus are of no relevance for the size of the array; namely the vertices $(1, 1, 1)$ and (N_1, N_2, N_3). This phenomenon is sketched in Figure 12.4.

The settings $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$ yield the vertices $(3, 0)$, $(3, -2)$, $(0, -2)$, $(-4, 2)$, $(-4, 4)$, and $(-1, 4)$. We see that space coordinates in principle can be negative. Moreover, the choice of an origin—that here lies in the interior of the polytope—might not always be obvious.

As the image of the projection, we get a systolic array with *hexagonal* shape and parallel opposite borders. On these, we find N_1, N_2, and N_3 integer points, respectively; cf. Figure 12.5. Thus, as opposed to our first example, *all* problem parameters here are also array parameters.

The area function of this region is of order $\Theta(N_1 \cdot N_2 + N_1 \cdot N_3 + N_2 \cdot N_3)$, and thus depends on all three matrix dimensions. So this is quite different from the situation in Figure 12.1(a), where the area function—for the same problem—is of order $\Theta(N_1 \cdot N_2)$.

Improving on this approximate calculation, we finally count the exact number of cells. For this process, it might be helpful to partition the entire region into subregions for which the number of cells comprised can be easily determined; see Figure 12.5. The points $(0, 0)$, $(N_3 - 1, 0)$, $(N_3 - 1, 1 - N_1)$, and $(0, 1 - N_1)$ are the vertices of a rectangle with $N_1 \cdot N_3$ cells. If we translate this point set up by $N_2 - 1$ cells and right by $N_2 - 1$ cells, we exactly cover the whole region. Each shift by one cell up and right contributes just another $N_1 + N_3 - 1$ cells. Altogether this yields $N_1 \cdot N_3 + (N_2 - 1) \cdot (N_1 + N_3 - 1) = N_1 \cdot N_2 + N_1 \cdot N_3 + N_2 \cdot N_3 - (N_1 + N_2 + N_3) + 1$ cells.

For $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$ we thereby get a number of 36 cells, as we have already learned from Figure 12.3(a).

12.2.4. Symbolically deriving the running time

The running time of a systolic algorithm can be symbolically calculated by an approach similar to that in section 12.2.3. The time transformation according to formula (12.6) as well is a linear map. We find the timesteps of the first and the last calculations as the minimum resp. maximum in the set $\pi(S) = \{\pi \cdot v : v \in S\}$ of execution timesteps. Following the discussion above, it thereby suffices to vary v over the vertices of the convex hull of S.

The ***running time*** is then given by the formula

$$t_\Sigma = 1 + \max P(S) - \min P(S) \ . \tag{12.18}$$

Adding one is mandatory here, since the first as well as the last timestep belong to the calculation.

For the example from Figure 12.3, the vertices of the polytope as enumerated in (12.16) are mapped by (12.7) to the set of images

$$\{3, 2 + N_1, 2 + N_2, 2 + N_3, 1 + N_1 + N_2, 1 + N_1 + N_3, 1 + N_2 + N_3, N_1 + N_2 + N_3\}qkoz.$$

With the basic assumption $N_1, N_2, N_3 \geq 1$, we get a minimum of 3 and a maximum of $N_1 + N_2 + N_3$, thus a running time of $N_1 + N_2 + N_3 - 2$ timesteps, as for the systolic array from Figure 12.1—no surprise, since the domains and the time vectors agree.

For the special problem parameters $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$, a running time of $12 - 3 + 1 = 10$ timesteps can be derived.

If $N_1 = N_2 = N_3 = N$, the systolic algorithm shows a running time of order $\Theta(N)$, using $\Theta(N^2)$ systolic cells.

12.2.5. How to unravel the communication topology

The ***communication topology*** of the systolic array is induced by applying the space-time transformation to the *data dependences* of the algorithm. Each data dependence results from a direct use of a variable instance to calculate another instance of the same variable, or an instance of another variable.

Remark. In contrast to the general situation where a data dependence analysis for imperative programming languages has to be performed by highly optimising compilers, data dependences here always are *flow dependences*. This is a direct consequence from the assignment-free notation employed by us.

The ***data dependences*** can be read off the quantified equations in our assignment-free notation by comparing their right and left sides. For example, we first analyse the equation

$c(i, j, k) = c(i, j, k-1) + a(i, j-1, k) \cdot b(i-1, j, k)$ from system (12.3).

The value $c(i, j, k)$ is calculated from the values $c(i, j, k-1)$, $a(i, j-1, k)$, and $b(i-1, j, k)$. Thus we have a ***data flow*** from $c(i, j, k-1)$ to $c(i, j, k)$, a data flow from $a(i, j-1, k)$ to $c(i, j, k)$, and a data flow from $b(i-1, j, k)$ to $c(i, j, k)$.

All properties of such a data flow that matter here can be covered by a ***dependence vector***, which is the iteration vector of the calculated variable instance minus the iteration vector of the correspondingly used variable instance.

The iteration vector for $c(i, j, k)$ is (i, j, k); that for $c(i, j, k-1)$ is $(i, j, k-1)$. Thus, as the difference vector, we find

$$d_C = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i \\ j \\ k-1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} . \tag{12.19}$$

Correspondingly, we get

$$d_A = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i \\ j-1 \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{12.20}$$

and

$$d_B = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i-1 \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} qkoz. \tag{12.21}$$

In the equation $a(i, j, k) = a(i, j-1, k)$ from system (12.3), we cannot directly recognise which is the calculated variable instance, and which is the used variable instance. This example elucidates the difference between *equations* and *assignments*. When fixing that $a(i, j, k)$ should follow from $a(i, j-1, k)$ by a ***copy operation***, we get the same dependence vector d_A as in (12.20). Correspondingly for the equation $b(i, j, k) = b(i-1, j, k)$.

A variable instance with iteration vector v is calculated in cell $P \cdot v$. If for this calculation another variable instance with iteration vector v' is needed, implying a data dependence with dependence vector $d = v - v'$, the used variable instance is provided by cell $P \cdot v'$. Therefore, we need a communication from cell $z' = P \cdot v'$ to cell $z = P \cdot v$. In systolic arrays, all communication has to be via direct static links between the communicating cells. Due to the linearity of the transformation from (12.9), we have $z - z' = P \cdot v - P \cdot v' = P \cdot (v - v') = P \cdot d$.

If $P \cdot d = \vec{0}$, communication happens exclusively inside the calculating cell, i.e., in time, only—and not in space. Passing values in time is via registers of the calculating cell.

Whereas for $P \cdot d \neq \vec{0}$, a communication between different cells is needed. Then a link along the ***flow direction*** $P \cdot d$ must be provided from/to all cells of the systolic array. The vector $-P \cdot d$, oriented in counter flow direction, leads from space point z to space point z'.

If there is more than one dependence vector d, we need an appropriate *link* for each of them at every cell. Take for example the formulas (12.19), (12.20), and (12.21) together with (12.11), then we get $P \cdot d_A = (-1, 1)$, $P \cdot d_B = (0, -1)$, and $P \cdot d_C = (1, 0)$. In Figure 12.3(a), terminating at every cell, we see three links corresponding to the various vectors $P \cdot d$. This results in a ***hexagonal communication topology***—instead of the ***orthogonal communication topology*** from the first example.

12.2.6. Inferring the structure of the cells

Now we apply the space-related techniques from section 12.2.5 to time-related questions. A variable instance with iteration vector v is calculated in timestep $\pi \cdot v$. If this calculation uses another variable instance with iteration vector v', the former had been calculated in timestep $\pi \cdot v'$. Hence communication corresponding to the dependence vector $d = v - v'$ must take exactly $\pi \cdot v - \pi \cdot v'$ timesteps.

Since (12.6) describes a linear map, we have $\pi \cdot v - \pi \cdot v' = \pi \cdot (v - v') = \pi \cdot d$. According to the systolic principle, each communication must involve at least one register. The dependence vectors d are fixed, and so the choice of a time vector π is constrained by

$$\pi \cdot d \geq 1 \ . \tag{12.22}$$

In case $P \cdot d = \vec{0}$, we must provide ***registers*** for stationary variables in all cells. But each register is overwritten with a new value in every timestep. Hence, if $\pi \cdot d \geq 2$, the old value must be carried on to a further register. Since this is repeated for $\pi \cdot d$ timesteps, the cell needs exactly $\pi \cdot d$ registers per stationary variable. The values of the stationary variable successively pass all these registers before eventually being used. If $P \cdot d \neq \vec{0}$, the transport of values analogously goes by $\pi \cdot d$ registers, though these are not required to belong all to the same cell.

For each dependence vector d, we thus need an appropriate number of registers. In Figure 12.3(b), we see three input ports at the cell, corresponding to the dependence vectors d_A, d_B, and d_C. Since for these we have $P \cdot d \neq \vec{0}$. Moreover, $\pi \cdot d = 1$ due to (12.7) and (12.4). Thus, we need one register per dependence vector. Finally, the regularity of system (12.3) forces three output ports for every cell, opposite to the corresponding input ports.

Good news: we can infer in general that each cell needs only a few registers, because the number of dependence vectors d is statically bounded with a system like (12.3), and for each of the dependence vectors the amount of registers $\pi \cdot d$ has a fixed and usually small value.

The three input and output ports at every cell now permit the use of three moving matrices. Very differently from Figure 12.1, a dot product $\sum_{k=1}^{4} a_{ik} \cdot b_{kj}$ here is not calculated within a single cell, but dispersed over the systolic array. As a prerequisite, we had to dissolve the sum into a sequence of single additions. We call this principle a ***distributed generic operator***.

Apart from the three input ports with their registers, and the three output ports, Figure 12.3(b) shows a multiplier chained to an adder. Both units are induced in each cell by applying the transformation (12.9) to the domain S of the equation $c(i, j, k) = c(i, j, k-1) + a(i, j-1, k) \cdot b(i-1, j, k)$ from system (12.3). According to this equation, the addition has to follow the calculation of the product, so the order of the hardware operators as seen in Figure 12.3(b) is implied.

The source cell for each of the used operands follows from the projection of the corresponding dependence vector. Here, variable $a(i, j-1, k)$ is related to the dependence vector $d_A = (0, 1, 0)$. The projection $P \cdot d_A = (-1, 1)$ constitutes the *flow direction* of matrix A. Thus the value to be used has to be expected, as observed by the calculating cell, in opposite direction $(1, -1)$, in this case from the port in the lower left corner of the cell, passing through register A. All the same, $b(i-1, j, k)$ comes from the right via register B, and $c(i, j, k-1)$ from above through register C. The calculated values $a(i, j, k)$, $b(i, j, k)$, and $c(i, j, k)$ are output into the opposite directions through the appropriate ports: to the upper right, to the left, and downwards.

If alternatively we use the projection matrix P from (12.8), then for d_C we get the direction $(0,0)$. The formula $\pi \cdot d_C = 1$ results in the requirement of exactly one register C for each item of the matrix C. This register provides the value $c(i,j,k-1)$ for the calculation of $c(i,j,k)$, and after this calculation receives the value $c(i,j,k)$. All this reasoning matches with the cell from Figure 12.1(b). Figure 12.1(a) correspondingly shows no links for matrix C between the cells: for the matrix is *stationary*.

Exercises

12.2-1 Each projection vector u induces several corresponding projection matrices P.

a. Show that

$$P = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix}$$

also is a projection matrix fitting with projection vector $u = (1,1,1)$.

b. Use this projection matrix to transform the domain from system (12.3).

c. The resulting space coordinates differ from that in section 12.2.3. Why, in spite of this, both point sets are topologically equivalent?

d. Analyse the cells in both arrangements for common and differing features.

12.2-2 Apply all techniques from section 12.2 to system (12.3), employing a space-time matrix

$$T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} .$$

12.3. Input/output schemes

In Figure 12.3(a), the *input/output scheme* is only sketched by the*flow directions* for the matrices A, B, C. The necessary details to understand the input/output operations are now provided by Figure 12.6.

The ***input/output scheme*** in Figure 12.6 shows some new phenomena when compared with Figure 12.1(a). The input and output cells belonging to any matrix are no longer threaded all on a single straight line; now, for each matrix, they lie along two adjacent *borders*, that additionally may differ in the number of links to the outside world. The data structures from Figure 12.6 also differ from that in Figure 12.1(a) in the angle of inclination. Moreover, the matrices A and B from Figure 12.6 arrive at the *boundary cells* with only one third of the *data rate*, compared to Figure 12.1(a).

Spending some effort, even here it might be possible in principle to construct—item by item—the appropriate input/output scheme fitting the present systolic array. But it is much more safe to apply a formal derivation. The following subsections are devoted to the presentation of the various methodical steps for achieving our goal.

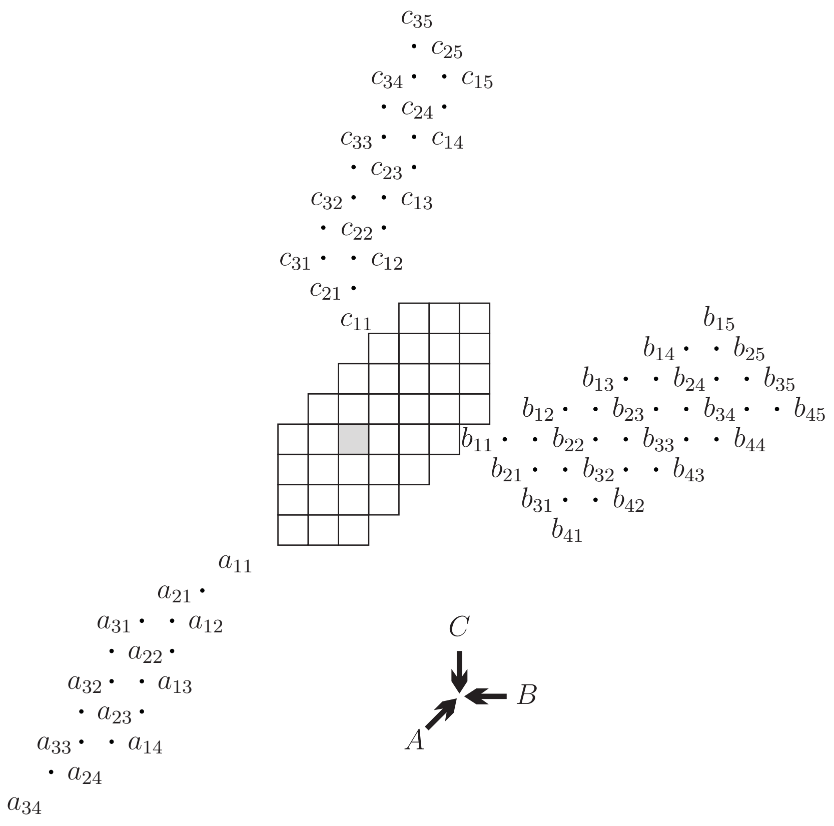


Figure 12.6. Detailed input/output scheme for the systolic array from Figure 12.3(a).

12.3.1. From data structure indices to iteration vectors

First, we need to construct a formal relation between the abstract data structures and the concrete variable instances in the assignment-free representation.

Each item of the matrix A can be characterised by a row index i and a column index k. These ***data structure indices*** can be comprised in a ***data structure vector*** $w = (i, k)$. Item a_{ik} in system (12.3) corresponds to the instances $a(i, j, k)$, with any j. The coordinates of these instances all lie on a line along direction $q = (0, 1, 0)$ in space $\mathbb{R}^3$. Thus, in this case, the formal change from data structure vector (i, k) to coordinates (i, j, k) can be described by the transformation

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ k \end{pmatrix} + j \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{12.23}$$

In system (12.3), the coordinate vector (i, j, k) of every variable instance equals the iteration vector of the domain point representing the calculation of this variable instance. Thus we also may interpret formula (12.23) as a relation between *data structure vectors* and *iteration vectors*. Abstractly, the desired iteration vectors v can be inferred from the data structure vector w by the formula

$$v = H \cdot w + \lambda \cdot q + p \ . \tag{12.24}$$

The affine vector p is necessary in more general cases, though always null in our example.

Because of $b(i, j, k) = b_{kj}$, the representation for matrix B correspondingly is

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} k \\ j \end{pmatrix} + i \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{12.25}$$

Concerning matrix C, each variable instance $c(i, j, k)$ may denote a different value. Nevertheless, all instances $c(i, j, k)$ to a fixed index pair (i, j) can be regarded as belonging to the same matrix item c_{ij}, since they all stem from the serialisation of the sum operator for the calculation of c_{ij}. Thus, for matrix C, following formula (12.24) we may set

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + k \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{12.26}$$

12.3.2. Snapshots of data structures

Each of the three matrices A, B, C is generated by two directions with regard to the *data structure indices*: along a row, and along a column. The difference vector $(0, 1)$ thereby describes a move from an item to the next item of the same row, i.e., in the next column: $(0, 1) = (x, y + 1) - (x, y)$. Correspondingly, the difference vector $(1, 0)$ stands for sliding from an item to the next item in the same column and next row: $(1, 0) = (x + 1, y) - (x, y)$.

Input/output schemes of the appearance shown in Figures 12.1(a) and 12.6 denote ***snapshots***: all positions of data items depicted, with respect to the entire systolic array, are related to a common timestep.

As we can notice from Figure 12.6, the rectangular shapes of the abstract data structures are mapped to parallelograms in the snapshot, due to the linearity of the applied space-time transformation. These parallelograms can be described by difference vectors along their borders, too.

Next we will translate difference vectors Δw from data structure vectors into spatial difference vectors Δz for the snapshot. Therefore, by choosing the parameter λ in formula (12.24), we pick a pair of iteration vectors v, v' that are mapped to the same timestep under our space-time transformation. For the moment it is not important which concrete timestep we thereby get. Thus, we set up

$$\pi \cdot v = \pi \cdot v' \quad \text{with} \quad v = H \cdot w + \lambda \cdot q + p \quad \text{and} \quad v' = H \cdot w' + \lambda' \cdot q + p \ , \tag{12.27}$$

implying

$$\pi \cdot H \cdot (w - w') + (\lambda - \lambda') \cdot \pi \cdot q = 0 \ , \tag{12.28}$$

and thus

$$\Delta\lambda = (\lambda - \lambda') = \frac{-\pi \cdot H \cdot (w - w')}{\pi \cdot q} \ . \tag{12.29}$$

Due to the linearity of all used transformations, the wanted spatial difference vector Δz hence follows from the difference vector of the data structure $\Delta w = w - w'$ as

$$\Delta z = P \cdot \Delta v = P \cdot H \cdot \Delta w + \Delta\lambda \cdot P \cdot q \ , \tag{12.30}$$

or

$$\Delta z = P \cdot H \cdot \Delta w - \frac{\pi \cdot H \cdot \Delta w}{\pi \cdot q} \cdot P \cdot q \ . \tag{12.31}$$

With the aid of formula (12.31), we now can determine the spatial difference vectors Δz for matrix A. As mentioned above, we have

$$H = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, \quad q = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad P = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix}, \quad \pi = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}.$$

Noting $\pi \cdot q = 1$, we get

$$\Delta z = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \Delta w + \Delta\lambda \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \text{with} \quad \Delta\lambda = -\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \Delta w \ .$$

For the rows, we have the difference vector $\Delta w = (0, 1)$, yielding the spatial difference vector $\Delta z = (2, -1)$. Correspondingly, from $\Delta w = (1, 0)$ for the columns we get $\Delta z = (1, -2)$. If we check with Figure 12.6, we see that the rows of A in fact run along the vector $(2, -1)$, the columns along the vector $(1, -2)$.

Similarly, we get $\Delta z = (-1, 2)$ for the rows of B, and $\Delta z = (1, 1)$ for the columns of B; as well as $\Delta z = (-2, 1)$ for the rows of C, and $\Delta z = (-1, -1)$ for the columns of C.

Applying these instruments, we are now able to reliably generate appropriate input/output schemes—although separately for each matrix at the moment.

12.3.3. Superposition of input/output schemes

Now, the shapes of the matrices A, B, C for the snapshot have been fixed. But we still have to adjust the matrices relative to the systolic array—and thus, also relative to each other. Fortunately, there is a simple graphical method for doing the task.

We first choose an arbitrary iteration vector, say $v = (1, 1, 1)$. The latter we map with the projection matrix P to the cell where the calculation takes place,

$$z = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} .$$

The iteration vector $(1, 1, 1)$ represents the calculations $a(1, 1, 1)$, $b(1, 1, 1)$, and $c(1, 1, 1)$; these in turn correspond to the data items a_{11}, b_{11}, and c_{11}. We now lay the input/output schemes for the matrices A, B, C on the systolic array in a way that the entries a_{11}, b_{11}, and c_{11} all are located in cell $z = (0, 0)$.

In principle, we would be done now. Unfortunately, our input/output schemes overlap with the cells of the systolic array, and are therefore not easily perceivable. Thus, we simultaneously retract the input/output schemes of all matrices in counter flow direction, place by place, until there is no more overlapping. With this method, we get exactly the input/output scheme from Figure 12.6.

As an alternative to this nice graphical method, we also could formally calculate an overlap-free placement of the various input/output schemes.

Only after specifying the input/output schemes, we can correctly calculate the number of timesteps effectively needed. The first relevant timestep starts with the first input operation. The last relevant timestep ends with the last output of a result. For the example, we determine from Figure 12.6 the beginning of the calculation with the input of the data item b_{11} in timestep 0, and the end of the calculation after output of the result c_{35} in timestep 14. Altogether, we identify 15 timesteps—five more than with pure treatment of the real calculations.

12.3.4. Data rates induced by space-time transformations

The input schemes of the matrices A and B from Figure 12.1(a) have a dense layout: if we drew the borders of the matrices shown in the Figure, there would be no spare places comprised.

Not so in Figure 12.6. In any input data stream, each data item is followed by two spare places there. For the input matrices this means: the *boundary cells* of the systolic array receive a proper data item only every third timestep.

This property is a direct result of the employed space-time transformation. In both examples, the abstract data structures themselves are dense. But how close the various items really come in the input/output scheme depends on the absolute value of the determinant of the transformation matrix T: in every input/output data stream, the proper items follow each other with a spacing of exactly $|\det(T)|$ places. Indeed $|\det(T)| = 1$ for Figure 12.1; as for Figure 12.6, we now can rate the fluffy spacing as a practical consequence of $|\det(T)| = 3$.

What to do with spare places as those in Figure 12.6? Although each cell of the systolic array from Figure 12.3 in fact does useful work only every third timestep, it would be nonsense to pause during two out of three timesteps. Strictly speaking, we can argue that values on places marked with dots in Figure 12.6 have no influence on the calculation of the shown items c_{ij}, because they never reach an active cell at time of the calculation of a variable $c(i, j, k)$. Thus, we may simply fill spare places with any value, no danger of disturbing the result. It is even feasible to execute three different matrix products at the same time on the systolic array from Figure 12.3, without interference. This will be our topic in section 12.3.7.

12.3.5. Input/output expansion and extended input/output scheme

When further studying Figure 12.6, we can identify another problem. Check, for example, the itinerary of c_{22} through the cells of the systolic array. According to the space-time transformation, the calculations contributing to the value of c_{22} happen in the cells $(-1, 0)$, $(0, 0)$, $(1, 0)$, and $(2, 0)$. But the input/output scheme from Figure 12.6 tells us that c_{22} also passes through cell $(-2, 0)$ before, and eventually visits cell $(3, 0)$, too.

This may be interpreted as some ***spurious calculations*** being introduced into the system

(12.3) by the used space-time transformation, here, for example, at the new domain points $(2, 2, 0)$ and $(2, 2, 5)$. The reason for this phenomenon is that the *domains of the input/output operations* are not in parallel to the chosen *projection direction*. Thus, some input/output operations are projected onto cells that do not belong to the ***boundary*** of the systolic array. But in the interior of the systolic array, no input/output operation can be performed directly. The problem can be solved by extending the trajectory, in flow or counter flow direction, from these inner cells up to the boundary of the systolic array. But thereby we introduce some new calculations, and possibly also some new domain points. This technique is called ***input/output expansion***.

We must avoid that the additional calculations taking place in the cells $(-2, 0)$ and $(3, 0)$ corrupt the correct value of c_{22}. For the matrix product, this is quite easy—though the general case is more difficult. The generic sum operator has a neutral element, namely zero. Thus, if we can guarantee that by new calculations only zero is added, there will be no harm. All we have to do is providing always at least one zero operand to any spurious multiplication; this can be achieved by filling appropriate input slots with zero items.

Figure 12.7 shows an example of a properly extended input/output scheme. Preceding and following the items of matrix A, the necessary zero items have been filled in. Since the entered zeroes count like data items, the input/output scheme from Figure 12.6 has been retracted again by one place. The calculation now begins already in timestep -1, but ends as before with timestep 14. Thus we need 16 timesteps altogether.

12.3.6. Coping with stationary variables

Let us come back to the example from Figure 12.1(a). For inputting the items of matrices A and B, no expansion is required, since these items are always used in *boundary cells* first. But not so with matrix C! The items of C are calculated in stationary variables, hence always in the same cell. Thus most results c_{ij} are produced in inner cells of the systolic array, from where they have to be moved—in a separate action—to *boundary cells* of the systolic array.

Although this new challenge, on the face of it, appears very similar to the problem from section 12.3.5, and thus very easy to solve, in fact we here have a completely different situation. It is not sufficient to extend existing data flows forward or backward up to the boundary of the systolic array. Since for stationary variables the dependence vector is the null vector, which constitutes no extensible direction, there can be no spatial flow induced by this dependency. Possibly, we can construct some auxiliary extraction paths, but usually there are many degrees of freedom. Moreover, we then need a *control mechanism* inside the cells. For all these reasons, the problem is further dwelled on in section 12.4.

12.3.7. Interleaving of calculations

As can be easily noticed, the *utilisation* of the systolic array from Figure 12.3 with input/output scheme from Figure 12.7 is quite poor. Even without any deeper study of the starting phase and the closing phase, we cannot ignore that the average utilisation of the array is below one third—after all, each cell at most in every third timestep makes a proper contribution to the calculation.

A simple technique to improve this behaviour is to ***interleave*** calculations. If we have three independent matrix products, we can successively input their respective data, delayed

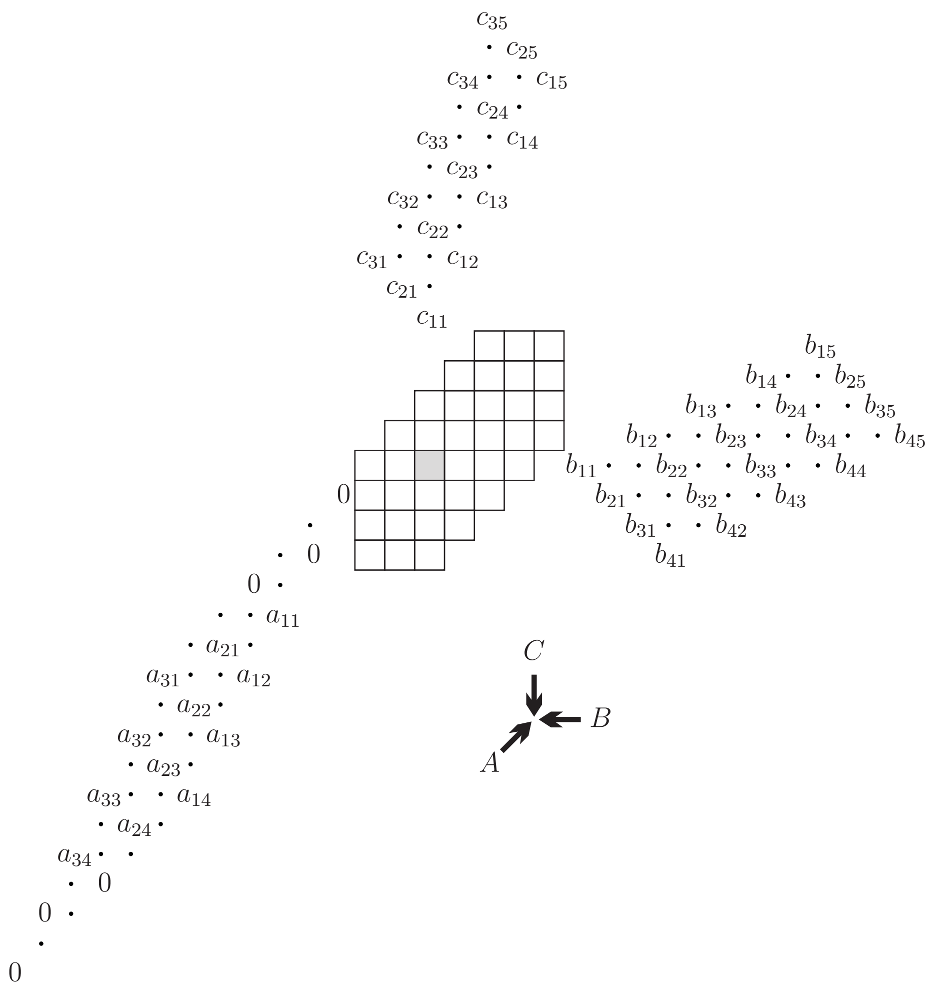


Figure 12.7. Extended input/output scheme, correcting Figure 12.6.

by only one timestep, without any changes to the systolic array or its cells. Figure 12.8 shows a snapshot of the systolic array, with parts of the corresponding input/output scheme.

Now we must check by a formal derivation whether this idea is really working. Therefore, we slightly modify system (12.3). We augment the variables and the domains by a fourth dimension, needed to distinguish the three matrix products:

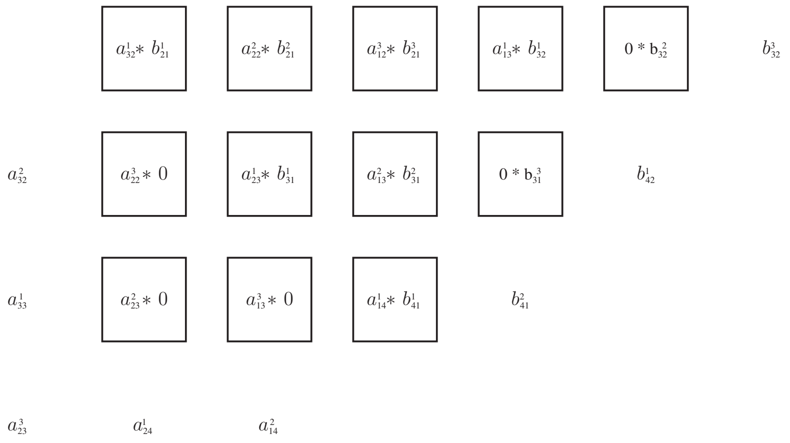


Figure 12.8. Interleaved calculation of three matrix products on the systolic array from Figure 12.3.

$$
\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k,l) = a^l_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, 1 \le l \le 3, \\
b(i,j,k,l) = b^l_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3, \\
c(i,j,k,l) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0, 1 \le l \le 3. \\
 & \\
\textit{calculations and forwarding} & \\
a(i,j,k,l) = a(i,j-1,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3, \\
b(i,j,k,l) = b(i-1,j,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3, \\
c(i,j,k,l) = c(i,j,k-1,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3. \\
\quad + a(i,j-1,k,l) \cdot b(i-1,j,k,l) & \\
 & \\
\textit{output operations} & \\
c^l_{ij} = c(i,j,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3, 1 \le l \le 3\ .
\end{array}
\tag{12.32}
$$

Obviously, in system (12.32), problems with different values of l are not related. Now we must preserve this property in the systolic array. A suitable *space-time matrix* would be

$$
T = \begin{pmatrix} 0 & -1 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} . \tag{12.33}
$$

Notice that T is not square here. But for calculating the space coordinates, the fourth dimension of the iteration vector is completely irrelevant, and thus can simply be neutralised by corresponding zero entries in the fourth column of the first and second rows of T.

The last row of T again constitutes the *time vector* π. Appropriate choice of π embeds the three problems to solve into the space-time continuum, avoiding any intersection. Corresponding instances of the iteration vectors of the three problems are projected to the same

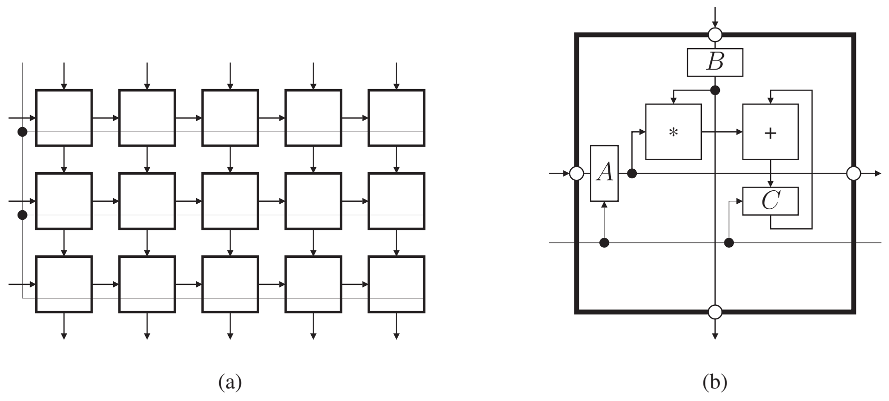


Figure 12.9. Resetting registers via global control. **(a)** Array structure. **(b)** Cell structure.

cell with a respective spacing of one timestep, because the fourth entry of π equals 1.

Finally, we calculate the average *utilisation*—with or without interleaving—for the concrete problem parameters $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$. For a single matrix product, we have to perform $N_1 \cdot N_2 \cdot N_3 = 60$ calculations, considering a multiplication and a corresponding addition as a compound operation, i.e., counting both together as only one calculation; input/output operations are not counted at all. The systolic array has 36 cells.

Without interleaving, our systolic array altogether takes 16 timesteps for calculating a single matrix product, resulting in an average utilisation of $60/(16 \cdot 36) \approx 0.104$ calculations per timestep and cell. When applying the described interleaving technique, the calculation of all three matrix products needs only two timesteps more, i.e., 18 timesteps altogether. But the number of calculations performed thereby has tripled, so we get an average utilisation of the cells amounting to $3 \cdot 60/(18 \cdot 36) \approx 0.278$ calculations per timestep and cell. Thus, by interleaving, we were able to improve the utilisation of the cells to 267 per cent!

Exercises

12.3-1 From equation (12.31), formally derive the spatial difference vectors of matrices B and C for the input/output scheme shown in Figure 12.6.

12.3-2 Augmenting Figure 12.6, draw an extended input/output scheme that forces both operands of all spurious multiplications to zero.

12.3-3 Apply the techniques presented in section 12.3 to the systolic array from Figure 12.1.

12.3-4★ Proof the properties claimed in section 12.3.7 for the special space-time transformation (12.33) with respect to system (12.32).

12.4. Control

So far we have assumed that each cell of a systolic array behaves in completely the same way during every timestep. Admittedly there are some relevant examples of such systolic arrays. However, in general the cells successively have to work in several ***operation modes***, switched to by some control mechanism. In the sequel, we study some typical situations for exerting control.

12.4.1. Cells without control

The cell from Figure 12.3(b) contains the registers A, B, and C, that—when activated by the global clock signal—accept the data applied to their inputs and then reliably reproduce these values at their outputs for one clock cycle. Apart from this system-wide activity, the function calculated by the cell is invariant for all timesteps: a *fused multiply-add* operation is applied to the three input operands A, B, and C, with result passed to a neighbouring cell; during the same cycle, the operands A and B are also forwarded to two other neighbouring cells. So in this case, the cell needs *no control* at all.

The *initial values* $c(i, j, 0)$ for the execution of the generic sum operator—which could also be different from zero here—are provided to the systolic array via the *input streams*, see Figure 12.7; the *final results* $c(i, j, N_3)$ continue to flow into the same direction up to the boundary of the array. Therefore, the input/output activities for the cell from Figure 12.3(b) constitute an intrinsic part of the normal cell function. The price to pay for this extremely simple cell function without any control is a restriction in all three dimensions of the matrices: on a systolic array like that from Figure 12.3, with *fixed* array parameters N_1, N_2, N_3, an $M_1 \times M_3$ matrix A can only be multiplied by an $M_3 \times M_2$ matrix B if the relations $M_1 \le N_1$, $M_2 \le N_2$, and $M_3 \le N_3$ hold.

12.4.2. Global control

In this respect, constraints for the array from Figure 12.1 are not so restrictive: though the problem parameters M_1 and M_2 also are bounded by $M_1 \le N_1$ and $M_2 \le N_2$, there is no constraint for M_3. Problem parameters unconstrained in spite of fixed array parameters can only emerge in time but not in space, thus mandating the use of *stationary variables*.

Before a new calculation can start, each register assigned to a stationary variable has to be *reset* to an initial state independent from the previously performed calculations. For instance, concerning the systolic cell from Figure 12.3(b), this should be the case for register C. By a *global* signal similar to the clock, register C can be cleared in all cells at the same time, i.e., reset to a zero value. To prevent a corruption of the reset by the current values of A or B, at least one of the registers A or B must be *cleared* at the same time, too. Figure 12.9 shows an array structure and a cell structure implementing this idea.

12.4.3. Local control

Unfortunately, for the matrix product the principle of the global control is not sufficient without further measures. Since the systolic array presented in Figure 12.1 even lacks another essential property: the results c_{ij} are not passed to the boundary but stay in the cells.

At first sight, it seems quite simple to forward the results to the boundary: when the calculation of an item c_{ij} is finished, the links from cell (i, j) to the neighbouring cells $(i, j + 1)$ and $(i + 1, j)$ are no longer needed to forward items of the matrices A and B. These links can be reused then for any other purpose. For example, we could pass all items of C

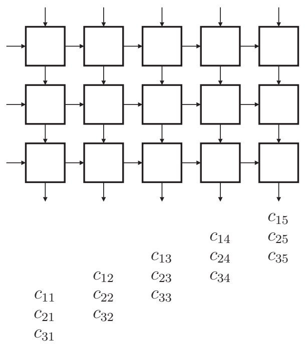


Figure 12.10. Output scheme with delayed output of results.

through the downward-directed links to the lower border of the systolic array.

But it turns out that leading through results from the upper cells is hampered by ongoing calculations in the lower parts of the array. If the result c_{ij}, finished in timestep $i + j + N_3$, would be passed to cell $(i+1, j)$ in the next timestep, a conflict would be introduced between two values: since only one value per timestep can be sent from cell $(i + 1, j)$ via the lower port, we would be forced to keep either c_{ij} or $c_{i+1\,j}$, the result currently finished in cell $(i + 1, j)$. This effect would spread over all cells down.

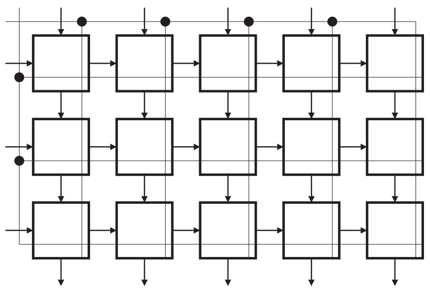
To fix the problem, we could ***slow down*** the forwarding of items c_{ij}. If it would take two timesteps for c_{ij} to pass a cell, no collisions could occur. Then, the results stage a procession through the same link, each separated from the next by one timestep. From the lower boundary cell of a column, the host computer first receives the result of the bottom row, then that of the penultimate row; this procedure continues until eventually we see the result of the top row. Thus we get the output scheme shown in Figure 12.10.

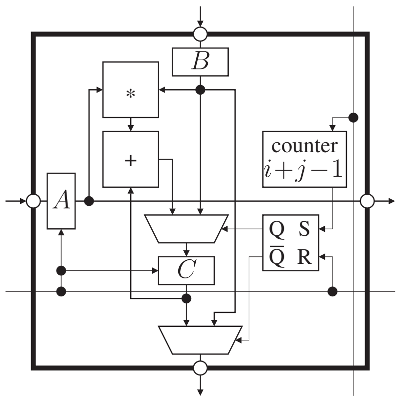
How can a cell recognise when to change from forwarding items of matrix B to passing items of matrix C through the lower port? We can solve this task by an automaton combining global control with local control in the cell:

If we send a global signal to all cells at exactly the moment when the last items of A and B are input to cell (1, 1), each cell can start a countdown process: in each successive timestep, we decrement a counter initially set to the number of the remaining calculation steps. Thereby cell (i, j) still has to perform $i + j - 1$ calculations before changing to *propagation mode*. Later, the already mentioned global reset signal switches the cell back to *calculation mode*.

Figure 12.11 presents a systolic array implementing this local/global principle. Basically, the array structure and the communication topology have been preserved. But each cell can run in one of two states now, switched by a *control logic*:

1. In *calculation mode*, as before, the result of the addition is written to register C. At the same time, the value in register B—i.e., the operand used for the multiplication—is forwarded through the lower port of the cell.
2. In *propagation mode*, registers B and C are connected in series. In this mode, the only function of the cell is to guide each value received at the upper port down to the lower port, thereby enforcing a *delay* of two timesteps.


(a)



(b)

Figure 12.11. Combined local/global control. **(a)** Array structure. **(b)** Cell structure.

The first value output from cell (i, j) in *propagation mode* is the currently calculated value c_{ij}, stored in register C. All further output values are results forwarded from cells above. A formal description of the algorithm implemented in Figure 12.11 is given by the assignment-free system (12.34).

$$
\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k) = a_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, \\
b(i,j,k) = b_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0. \\
\textit{calculations and forwarding} & \\
a(i,j,k) = a(i,j-1,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
b(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3. \\
\quad + a(i,j-1,k) \cdot b(i-1,j,k) & \\
\textit{propagation} & \\
b(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i + N_3, \\
c(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i - 1 + N_3, \\
\textit{output operations} & \\
c_{1+N_1+N_3-k,j} = b(i,j,k) & i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3 \ .
\end{array}
\tag{12.34}
$$

It rests to explain how the control signals in a cell are generated in this model. As a prerequisite, the cell must contain a ***state flip-flop*** indicating the current *operation mode*.

The output of this flip-flop is connected to the control inputs of both multiplexors, see Figure 12.11(b). The global reset signal clears the state flip-flop, as well as the registers A and C: the cell now works in *calculation mode*.

The global ready signal starts the countdown in all cells, so in every timestep the counter is diminished by 1. The counter is initially set to the precalculated value $i + j - 1$, dependent on the position of the cell. When the counter reaches zero, the flip-flop is set: the cell switches to *propagation mode*.

If desisting from a direct reset of the register C, the last value passed, before the reset, from register B to register C of a cell can be used as a freely decidable initial value for the next dot product to evaluate in the cell. We then even calculate, as already in the systolic array from Figure 12.3, the more general problem

$$C = A \cdot B + D \ , \tag{12.35}$$

detailed by the following equation system:

$$\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k) = a_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, \\
b(i,j,k) = b_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = d_{ij} & 1 \le i \le N_1, 1 \le j \le N_2, k = 0 \ . \\
\textit{calculations and forwarding} & \\
a(i,j,k) = a(i,j-1,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
b(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ . \\
\quad + a(i,j-1,k) \cdot b(i-1,j,k) & \\
\textit{propagation} & \\
b(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i + N_3, \\
c(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i - 1 + N_3 \ . \\
\textit{output operations} & \\
c_{1+N_1+N_3-k,j} = b(i,j,k) & i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3 \ .
\end{array} \tag{12.36}$$

12.4.4. Distributed control

The method sketched in Figure 12.11 still has the following drawbacks:

1. The systolic array uses global control signals, requiring a high technical accuracy.
2. Each cell needs a counter with counting register, introducing a considerable hardware expense.
3. The initial value of the counter varies between the cells. Thus, each cell must be individually designed and implemented.

4. The input data of any successive problem must wait outside the cells until all results from the current problem have left the systolic array.

These disadvantages can be avoided, if *control signals* are *propagated like data*—meaning a ***distributed control***. Therefore, we preserve the connections of the registers B and C with the multiplexors from Figure 12.11(b), but do not generate any control signals in the cells; also, there will be no global reset signal. Instead, a cell receives the necessary control signal from one of the neighbours, stores it in a new one-bit register S, and appropriately forwards it to further neighbouring cells. The primary control signals are generated by the *host computer*, and infused into the systolic array by *boundary cells*, only. Figure 12.12(a) shows the required array structure, Figure 12.12(b) the modified cell structure.

Switching to the *propagation mode* occurs successively down one cell in a column, always delayed by one timestep. The delay introduced by register S is therefore sufficient.

Reset to the *calculation mode* is performed via the same control wire, and thus also happens with a delay of one timestep per cell. But since the results c_{ij} sink down at half speed, only, we have to wait sufficiently long with the reset: if a cell is switched to *calculation mode* in timestep t, it goes to *propagation mode* in timestep $t + N_3$, and is reset back to *calculation mode* in timestep $t + N_1 + N_3$.

So we learned that in a systolic array, distributed control induces a different macroscopic timing behaviour than local/global control. Whereas the systolic array from Figure 12.12 can start the calculation of a new problem (12.35) every $N_1 + N_3$ timesteps, the systolic array from Figure 12.11 must wait for $2 \cdot N_1 + N_2 + N_3 - 2$ timesteps. The time difference $N_1 + N_3$ resp. $2 \cdot N_1 + N_2 + N_3 - 2$ is called the ***period***, its reciprocal being the ***throughput***.

System (12.37) formally describes the relations between distributed control and calculations. We thereby assume an infinite, densely packed sequence of matrix product problems, the additional iteration variable l being unbounded. The equation headed *variables with alias* describes but pure identity relations.

$$
\begin{array}{ll}
\textit{control} & \\
s(i,j,k,l) = 0 & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
s(i,j,k,l) = 1 & i = 0, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3, \\
s(i,j,k,l) = s(i-1,j,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ . \\
& \\
\textit{input operations} & \\
a(i,j,k,l) = a^l_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, \\
b(i,j,k,l) = b^l_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
b(i,j,k,l) = d^{l+1}_{N_1+N_3+1-k,j} & i = 0, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ . \\
& \\
\textit{variables with alias} & \\
c(i,j,k,l) = c(i,j,N_1+N_3,l-1) & 1 \le i \le N_1, 1 \le j \le N_2, k = 0\ . \\
& \\
\textit{calculations and forwarding} & \\
a(i,j,k,l) = a(i,j-1,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ , \\
b(i,j,k,l) = \begin{cases} b(i-1,j,k,l) : s(i-1,j,k,l) = 0 \\ c(i,j,k-1,l) : s(i-1,j,k,l) = 1 \end{cases} & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ , \\
c(i,j,k,l) = \begin{cases} c(i,j,k-1,l) \\ \ + a(i,j-1,k,l) \\ \ \cdot b(i-1,j,k,l) : s(i-1,j,k,l) = 0 \\ b(i-1,j,k,l) \quad : s(i-1,j,k,l) = 1 \end{cases} & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ . \\
& \\
\textit{output operations} & \\
c^l_{1+N_1+N_3-k,j} = b(i,j,k,l) & i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ .
\end{array}
\tag{12.37}
$$

Formula (12.38) shows the corresponding space-time matrix. Note that one entry of T is not constant but depends on the *problem parameters*:

$$
T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & N_1 + N_3 \end{pmatrix}
\tag{12.38}
$$

Interestingly, also the cells in a row switch one timestep later when moving one position to the right. Sacrificing some regularity, we could use this circumstance to relieve the *host computer* by applying control to the systolic array at cell $(1, 1)$, only. We therefore would have to change the control scheme in the following way:

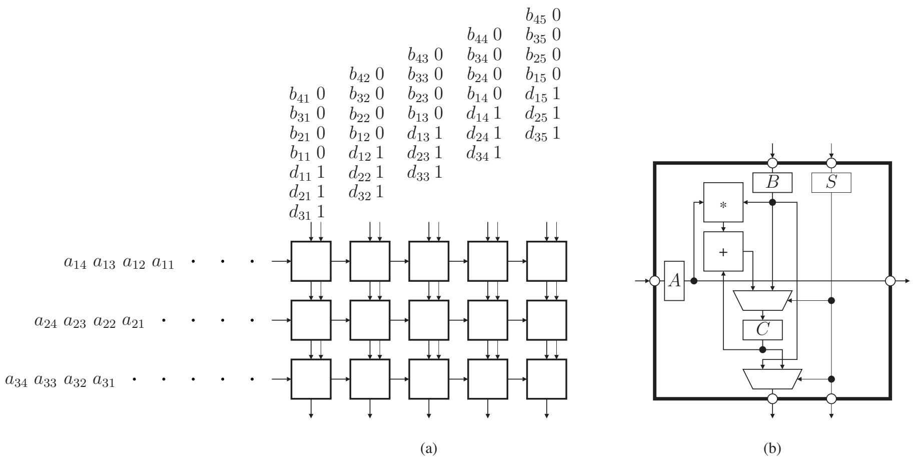


Figure 12.12. Matrix product on a rectangular systolic array, with output of results and distributed control. **(a)** Array structure. **(b)** Cell structure.

$$
\begin{aligned}
&\textit{control} && \\
&s(i,j,k,l) = 0 && i = 1, j = 0, 1 \le k \le N_3\ , \\
&s(i,j,k,l) = 1 && i = 1, j = 0, 1 + N_3 \le k \le N_1 + N_3\ , \\
&s(i,j,k,l) = s(i-1,j,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ , \\
&\dots && \\
&\textit{variables with alias} && \\
&s(i,j,k,l) = s(i+1,j-1,k,l) && i = 0, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ , \\
&\dots &&
\end{aligned}
\tag{12.39}
$$

Figure 12.13 shows the result of this modification. We now need cells of two kinds: cells on the upper border of the systolic array must be like that in Figure 12.13(b); all other cells would be as before, see Figure 12.13(c). Moreover, the *communication topology* on the upper border of the systolic array would be slightly different from that in the regular area.

12.4.5. The cell program as a local view

The chosen *space-time transformation* widely determines the architecture of the systolic array. Mapping recurrence equations to space-time coordinates yields an explicit view to the *geometric properties* of the systolic array, but gives no real insight into the *function of the cells*. In contrast, the processes performed inside a cell can be directly expressed by a ***cell program***. This approach is particularly of interest if dealing with a *programmable systolic array*, consisting of cells indeed controlled by a repetitive program.

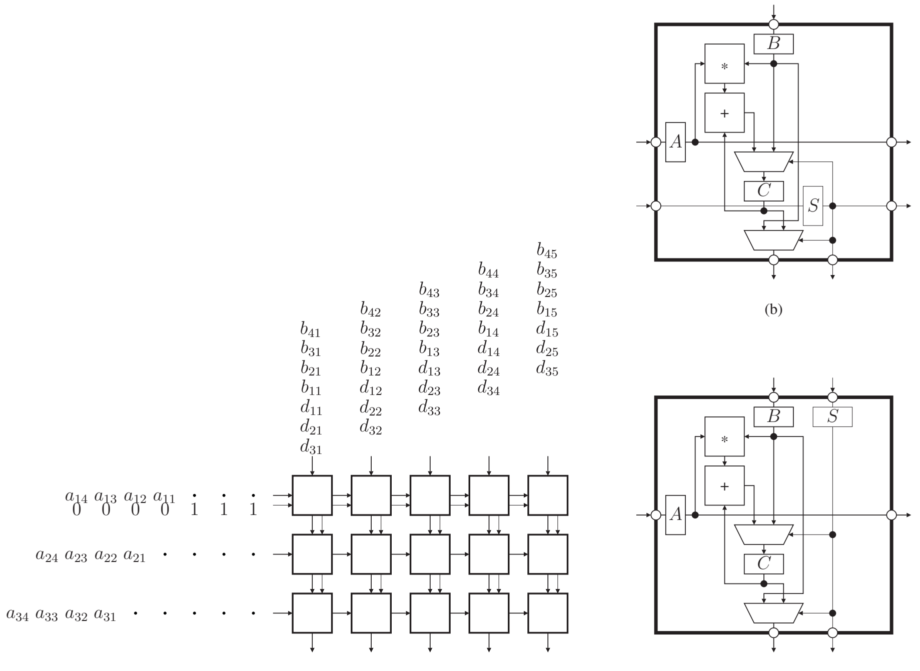


Figure 12.13. Matrix product on a rectangular systolic array, with output of results and distributed control. **(a)** Array structure. **(b)** Cell on the upper border. **(c)** Regular cell.

Like the ***global view***, i.e., the *structure* of the systolic array, the ***local view*** given by a *cell program* in fact is already fixed by the space-time transformation. But, this local view is only induced implicitly here, and thus, by a further mathematical transformation, an explicit representation must be extracted, suitable as a cell program.

In general, we denote instances of program variables with the aid of ***index expressions***, that refer to iteration variables. Take, for instance, the equation

$$c(i,j,k) = c(i,j,k-1) + a(i,j-1,k) \cdot b(i-1,j,k) \qquad 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3$$

from system (12.3). The instance $c(i,j,k-1)$ of the program variable c is specified using the index expressions i, j, and $k-1$, which can be regarded as functions of the iteration variables i,j,k.

As we have noticed, the set of iteration vectors (i,j,k) from the quantification becomes a set of space-time coordinates (x,y,t) when applying a space-time transformation (12.12) with transformation matrix T from (12.14),

$$\begin{pmatrix} x \\ y \\ t \end{pmatrix} = T \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} . \tag{12.40}$$

Since each cell is denoted by space coordinates (x,y), and the cell program must refer to the current time t, the iteration variables i, j, k in the index expressions for the program variables are not suitable, and must be translated into the new coordinates x, y, t. Therefore, using the inverse of the space-time transformation from (12.40), we express the iteration variables i, j, k as functions of the space-time coordinates (x, y, t),

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \\ t \end{pmatrix} = \frac{1}{3} \cdot \begin{pmatrix} -1 & -2 & 1 \\ -1 & 1 & 1 \\ 2 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ t \end{pmatrix} . \tag{12.41}$$

The existence of such an inverse transformation is guaranteed if the space-time transformation is injective on the domain—and that it should always be: if not, some instances must be calculated by a cell in the same timestep. In the example, reversibility is guaranteed by the square, non singular matrix T, even without referral to the domain. With respect to the time vector π and any projection vector u, the property $\pi \cdot u \neq 0$ is sufficient.

Replacing iteration variables by space-time coordinates, which might be interpreted as a ***transformation of the domain***, frequently yields very unpleasant index expressions. Here, for example, from $c(i, j, k-1)$ we get

$$c((-x - 2 \cdot y + t)/3, (-x + y + t)/3, (2 \cdot x + y + t)/3) qkoz .$$

But, by a successive ***transformation of the index sets***, we can relabel the instances of the program variables such that the reference to cell and time appears more evident. In particular, it seems worthwhile to transform the equation system back into ***output normal form***, i.e., to denote the results calculated during timestep t in cell (x, y) by instances (x, y, t) of the program variables. We best gain a real understanding of this approach via an abstract mathematical formalism, that we can fit to our special situation.

Therefore, let

$$r(\psi_r(v)) = \mathcal{F}(\ldots, s(\psi_s(v)), \ldots) \qquad v \in S \tag{12.42}$$

be a quantified equation over a domain S, with program variables r and s. The ***index functions*** ψ_r and ψ_s generate the instances of the program variables as tuples of index expressions.

By transforming the domain with a function φ that is injective on S, equation (12.42) becomes

$$r(\psi_r(\varphi^{-1}(e))) = \mathcal{F}(\ldots, s(\psi_s(\varphi^{-1}(e))), \ldots) \qquad e \in \varphi(S), \tag{12.43}$$

where φ^{-1} is a function that constitutes an inverse of φ on $\varphi(S)$. The new index functions are $\psi_r \circ \varphi^{-1}$ and $\psi_s \circ \varphi^{-1}$.

Transformations of index sets don't touch the domain; they can be applied to each program variable separately, since only the instances of this program variable are renamed, and in a consistent way. With such renamings ϑ_r and ϑ_s, equation (12.43) becomes

$$r(\vartheta_r(\psi_r(\varphi^{-1}(e)))) = \mathcal{F}(\ldots, s(\vartheta_s(\psi_s(\varphi^{-1}(e)))), \ldots) \qquad e \in \varphi(S) . \tag{12.44}$$

If output normal form is desired, $\vartheta_r \circ \psi_r \circ \varphi^{-1}$ has to be the identity.

In the most simple case (as for our example), ψ_r is the identity, and ψ_s is an affine transformation of the form $\psi_s(v) = v - d$, with constant d—the already known *dependence vector*. ψ_r then can be represented in the same way, with $d = \vec{0}$. Transformation of the domains happens by the space-time transformation $\varphi(v) = T \cdot v$, with an invertible matrix T.

For all index transformations, we choose the same $\vartheta = \varphi$. Thus equation (12.44) becomes

$$r(e) = \mathcal{F}(\dots, s(e - T \cdot d), \dots) \qquad e \in T(S) . \tag{12.45}$$

For the generation of a *cell program*, we have to know the following information for every timestep: the operation to perform, the source of the data, and the destination of the results—known from assembler programs as `opc`, `src`, `dst`.

The operation to perform (`opc`) follows directly from the function $\mathcal{F}$. For a cell with control, we must also find the timesteps when to perform this individual function $\mathcal{F}$. The set of these timesteps, as a function of the space coordinates, can be determined by projecting the set $T(S)$ onto the time axis; for general polyhedric S with the aid of a *Fourier-Motzkin elimination*, for example.

In system (12.45), we get a new dependence vector $T \cdot d$, consisting of two components: a (vectorial) spatial part, and a (scalar) timely part. The *spatial* part Δz, as a difference vector, specifies *which* neighbouring cell has calculated the operand. We directly can translate this information, concerning the input of operands to cell z, into a port specifier with port position $-\Delta z$, serving as the `src` operand of the instruction. In the same way, the cell calculating the operand, with position $z - \Delta z$, must write this value to a port with port position Δz, used as the `dst` operand in the instruction.

The *timely* part of $T \cdot d$ specifies, as a time difference Δt, *when* the calculation of the operand has been performed. If $\Delta t = 1$, this information is irrelevant, because the reading cell z always gets the output of the immediately preceding timestep from neighbouring cells. However, for $\Delta t > 1$, the value must be buffered for $\Delta t - 1$ timesteps, either by the *producer* cell $z - \Delta z$, or by the *consumer* cell z—or by both, sharing the burden. This need can be realised in the cell program, for example, with $\Delta t - 1$ copy instructions executed by the producer cell $z - \Delta z$, preserving the value of the operand until its final output from the cell by passing it through $\Delta t - 1$ registers.

Applying this method to system (12.37), with transformation matrix T as in (12.38), yields

$$\begin{aligned}
s(x,y,t) &= s(x-1,y,t-1) \\
a(x,y,t) &= a(x,y-1,t-1) \\
b(x,y,t) &= \begin{cases} b(x-1,y,t-1) & : s(x-1,y,t-1) = 0 \\ c(x,y,t-1) & : s(x-1,y,t-1) = 1 \end{cases} \\
c(x,y,t) &= \begin{cases} c(x,y,t-1) \\ \;+ a(x,y-1,t-1) \\ \;\cdot b(x-1,y,t-1) & : s(x-1,y,t-1) = 0 \\ b(x-1,y,t-1) & : s(x-1,y,t-1) = 1 \end{cases}
\end{aligned} \tag{12.46}$$

The iteration variable l, being relevant only for the input/output scheme, can be set to a fixed value prior to the transformation. The corresponding cell program for the systolic array from Figure 12.12, performed once in every timestep, reads as follows:

Cell-Program

```
 1  S ← C(−1, 0)(0)
 2  A ← C(0, −1)
 3  B ← C(−1, 0)(1 : N)
 4  C(1, 0)(0) ← S
 5  C(0, 1) ← A
 6  if S = 1
 7    then C(1, 0)(1 : N) ← C
 8         C ← B
 9    else C(1, 0)(1 : N) ← B
10         C ← C + A · B
```

The port specifiers stand for local input/output to/from the cell. For each, a pair of qualifiers is derived from the geometric position of the ports relative to the centre of the cell. Port C$(0, -1)$ is situated on the left border of the cell, C$(0, 1)$ on the right border; C$(-1, 0)$ is above the centre, C$(1, 0)$ below. Each port specifier can be augmented by a bit range: C$(-1, 0)(0)$ stands for bit 0 of the port, only; C$(-1, 0)(1 : N)$ denotes the bits 1 to N. The designations A, B, ... without port qualifiers stand for registers of the cell.

By application of matrix T from (12.13) to system (12.36), we get

$$\begin{aligned}
a(x, y, t) &= a(x, y-1, t-1) && 1 + x + y \le t \le x + y + N_3 \;, \\
b(x, y, t) &= b(x-1, y, t-1) && 1 + x + y \le t \le x + y + N_3 \;, \\
c(x, y, t) &= c(x, y, t-1) && 1 + x + y \le t \le x + y + N_3 \;, \\
&\quad + a(x, y-1, t-1) \cdot b(x-1, y, t-1) && \\
b(x, y, t) &= c(x, y, t-1) && x + y + 1 + N_3 \le t \le 2 \cdot x + y + N_3 \;, \\
c(x, y, t) &= b(x-1, y, t-1) && x + y + 1 + N_3 \le t \le 2 \cdot x + y - 1 + N_3 \;.
\end{aligned} \tag{12.47}$$

Now the advantages of distributed control become obvious. The cell program for (12.46) can be written with referral to the respective timestep t, only. And thus, we need no reaction to global control signals, no counting register, no counting operations, and no coding of the local cell coordinates.

Exercises

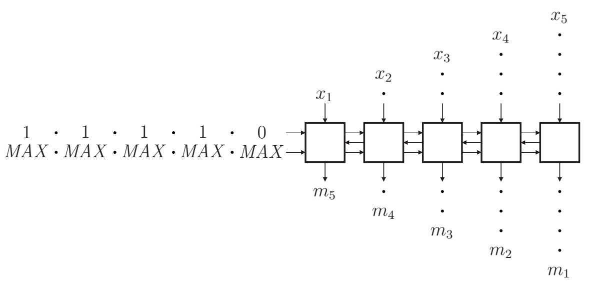
12.4-1 Specify appropriate input/output schemes for performing, on the systolic arrays presented in Figures 12.11 and 12.12, two evaluations of system (12.36) that follow each other closest in time.

12.4-2 How could we change the systolic array from Figure 12.12, to efficiently support the calculation of matrix products with parameters $M_1 < N_1$ or $M_2 < N_2$?

12.4-3 Write a cell program for the systolic array from Figure 12.3.

12.4-4★ Which throughput allows the systolic array from Figure 12.3 for the assumed values of N_1, N_2, N_3? Which for general N_1, N_2, N_3?

12.4-5★ Modify the systolic array from Figure 12.1 such that the results stored in stationary variables are output through additional links directed half right down, i.e., from cell (i, j) to cell $(i+1, j+1)$. Develop an assignment-free equation system functionally equivalent to system (12.36), that is compatible with the extended structure. How looks the resulting



x_1 x_2 x_3 x_4 x_5
1 · 1 · 1 · 1 · 0
$MAX \cdot MAX \cdot MAX \cdot MAX \cdot MAX$
m_5 m_4 m_3 m_2 m_1
(a)

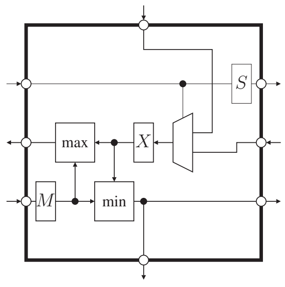


Figure 12.14. *Bubble sort* algorithm on a linear systolic array. **(a)** Array structure with input/output scheme. **(b)** Cell structure.

input/output scheme? Which period is obtained?

12.5. Linear systolic arrays

Explanations in the sections above heavily focused on *two-dimensional* systolic arrays, but in principle also apply to *one-dimensional* systolic arrays, called ***linear systolic arrays*** in the sequel. The most relevant difference between both kinds concerns the *boundary* of the systolic array. Linear systolic arrays can be regarded as consisting of *boundary cells*, only; under this assumption, input from and output to the *host computer* needs no special concern. However, the geometry of a linear systolic array provides one full dimension as well as one fictitious dimension, and thus communication along the full-dimensional axis may involve similar questions as in section 12.3.5. Eventually, the boundary of the linear systolic array can also be defined in a radically different way, namely to consist of both ***end cells***, only.

12.5.1. Matrix-vector product

If we set one of the problem parameters N_1 or N_2 to value 1 for a systolic array as that from Figure 12.1, the matrix product means to multiply a matrix by a vector, from left or right. The two-dimensional systolic array then ***degenerates*** to a one-dimensional systolic array. The vector by which to multiply is provided as an input data stream through an end cell of the linear systolic array. The matrix items are input to the array simultaneously, using the complete broadside.

As for full matrix product, results emerge stationary. But now, they either can be drained along the array to one of the end cells, or they are sent directly from the producer cells to the *host computer*. Both methods result in different control mechanisms, time schemes, and running time.

Now, would it be possible to provide *all* inputs via end cells? The answer is negative if the running time should be of complexity $\Theta(N)$. Matrix A contains $\Theta(N^2)$ items, thus there are $\Theta(N)$ items per timestep to read. But the number of items receivable through an end cell during one timestep is bounded. Thus, the ***input/output data rate***—of order $\Theta(N)$, here—may already constrain the possible design space.

12.5.2. Sorting algorithms

For sorting, the task is to bring the elements from a set $\{x_1, \ldots, x_N\}$, subset of a totally ordered basic set G, into an ascending order $\{m_i\}_{i=1,\ldots,N}$ where $m_i \leq m_k$ for $i < k$. A solution to this problem is described by the following assignment-free equation system, where MAX denotes the maximum in G:

$$\begin{array}{ll}
\textit{input operations} & \\
x(i,j) = x_i & 1 \leq i \leq N, j = 0, \\
m(i,j) = MAX & 1 \leq j \leq N, i = j-1 \;. \\
& \\
\textit{calculations} & \\
m(i,j) = \min\{x(i,j-1), m(i-1,j)\} & 1 \leq i \leq N, 1 \leq j \leq i \;, \\
x(i,j) = \max\{x(i,j-1), m(i-1,j)\} & 1 \leq i \leq N, 1 \leq j \leq i \;. \\
& \\
\textit{output operations} & \\
m(i,j) = m_j & 1 \leq j \leq N, i = N \;.
\end{array} \tag{12.48}$$

By completing a projection along direction $u = (1,1)$ to a space-time transformation

$$\begin{pmatrix} x \\ t \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}, \tag{12.49}$$

we get the linear systolic array from Figure 12.14, as an implementation of the *bubble sort* algorithm.

Correspondingly, the space-time matrix

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{12.50}$$

would induce another linear systolic array, that implements *insertion sort*. Eventually, the space-time matrix

$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{12.51}$$

would lead to still another linear systolic array, this one for *selection sort*.

For the sorting problem, we have $\Theta(N)$ input items, $\Theta(N)$ output items, and $\Theta(N)$ timesteps. This results in an input/output data rate of order $\Theta(1)$. In contrast to the matrix-vector product from section 12.5.1, the sorting problem with any prescribed input/output data rate in principle allows to perform the communication exclusively through the end cells of a linear systolic array.

Note that, in all three variants of sorting described so far, direct input is necessary to all cells: the values to order for bubble sort, the constant values *MAX* for insertion sort, and both for selection sort. However, instead of inputting the constants, the cells could generate them, or read them from a local memory.

All three variants require a cell control: insertion sort and selection sort use stationary variables; bubble sort has to switch between the processing of input data and the output of calculated values.

12.5.3. Lower triangular linear equation systems

System (12.52) below describes a *localised* algorithm for solving the linear equation system $A \cdot x = b$, where the $N \times N$ matrix A is a lower triangular matrix.

$$\begin{array}{ll}
\textit{input operations} & \\
a(i,j) = a_{i,j+1} & 1 \le i \le N, 0 \le j \le i-1\ , \\
u(i,j) = b_i & 1 \le i \le N, j = 0\ . \\
& \\
\textit{calculations and forwarding} & \\
u(i,j) = u(i,j-1) - a(i,j-1) \cdot x(i-1,j) & 2 \le i \le N, 1 \le j \le i-1\ , \\
x(i,j) = u(i,j-1)/a(i,j-1) & 1 \le i \le N, j = i\ , \\
x(i,j) = x(i-1,j) & 2 \le i \le N-1, 1 \le j \le i-1\ . \\
& \\
\textit{output operations} & \\
x_i = x(i,j) & 1 \le i \le N, j = i\ .
\end{array} \tag{12.52}$$

All previous examples had in common that, apart from copy operations, the same kind of calculation had to be performed for each domain point: fused multiply/add for the matrix algorithms, minimum and maximum for the sorting algorithms. In contrast, system (12.52) contains some domain points where multiply and subtract is required, as well as some others needing division.

When projecting system (12.52) to a linear systolic array, depending on the chosen *projection direction* we get fixed or varying cell functions. Peculiar for projecting along $u = (1,1)$, we see a single cell with divider; all other cells need a multiply/subtract unit. Projection along $u = (1,0)$ or $u = (0,1)$ yields identical cells, all containing a divider as well as a multiply/subtract unit. Projection vector $u = (1,-1)$ results in a linear systolic array with three different cell types: both end cells need a divider, only; all other cells contain a multiply/subtract unit, with or without divider, alternatingly. Thus, a certain projection can introduce ***inhomogeneities*** into a systolic array—that may be desirable, or not.

Exercises

12.5-1 For both variants of matrix-vector product as in section 12.5.1—output of the results by an end cell versus communication by all cells—specify a suitable array structure with input/output scheme and cell structure, including the necessary control mechanisms.

12.5-2 Study the effects of further projection directions on system (12.52).

12.5-3 Construct systolic arrays implementing insertion sort and selection sort, as mentioned in section 12.5.2. Also draw the corresponding cell structures.

12.5-4★ The systolic array for bubble sort from Figure 12.14 could be operated without control by cleverly organising the input streams. Can you find the trick?

12.5-5★ What purpose serves the value *MAX* in system (12.48)? How system (12.48) could be formulated without this constant value? Which consequences this would incur for the systolic arrays described?

Problems

12-1. Band matrix algorithms

In sections 12.1, 12.2, 12.5.1, and 12.5.3, we always assumed ***full*** input matrices, i.e., each matrix item a_{ij} used could be nonzero in principle. (Though in a lower triangular matrix, items above the main diagonal are all zero. Note, however, that these items are not inputs to any of the algorithms described.)

In contrast, practical problems frequently involve ***band matrices***, (see)cf. Kung/Leiserson [35]. In such a matrix, most diagonals are zero, left alone a small band around the main diagonal. Formally, we have $a_{ij} = 0$ for all i, j with $i - j \geq K$ or $j - i \geq L$, where K and L are positive integers. The ***band width***, i.e., the number of diagonals where nonzero items may appear, here amounts to $K + L - 1$.

Now the question arises whether we could profit from the band structure in one or more input matrices to optimise the systolic calculation. One opportunity would be to delete cells doing no useful work. Other benefits could be shorter input/output data streams, reduced running time, or higher throughput.

Study all systolic arrays presented in this chapter for improvements with respect to these criteria.

Chapter notes

The term *systolic array* has been coined by Kung and Leiserson in their seminal paper [35].

Karp, Miller, and Winograd did some pioneering work [32] for *uniform recurrence equations*.

Essential stimuli for a theory on the systematic design of systolic arrays have been Rao's PhD dissertation [48] and the work of Quinton [47].

The contribution of Teich and Thiele [59] shows that a formal derivation of the cell control can be achieved by methods very similar to those for a determination of the geometric array structure and the basic cell function.

The up-to-date book by Darte, Robert, and Vivien [15] joins advanced methods from compiler design and systolic array design, dealing also with the analysis of data dependences.

The monograph [63] still seems to be the most comprehensive work on systolic systems.

Each systolic array can also be modelled as a *cellular automaton*. The registers in a cell together hold the state of the cell. Thus, a *factorised* state space is adequate. Cells of different kind, for instance with varying cell functionality or position-dependent cell control, can be described with the aid of further components of the state space.

Each systolic algorithm also can be regarded as a ***PRAM algorithm*** with the same timing behaviour. Thereby, each register in a systolic cell corresponds to a PRAM memory cell, and vice versa. The EREW PRAM model is sufficient, because in every timestep exactly one systolic cell reads from this register, and then exactly one systolic cell writes to this register.

Each systolic system also is a special kind of *synchronous network* as defined by Lynch [37]. Time complexity measures agree. Communication complexity usually is no topic with systolic arrays. Restriction to input/output through boundary cells, frequently demanded for systolic arrays, also can be modelled in a synchronous network. The concept of *failures* is not required for systolic arrays.

The book [54] due to Sima, Kacsuk and Fountaine considers systolic systems in details.

Bibliography

[1] S. Albers, H. Bals. Dynamic TCP acknowledgement, penalizing long delays. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms*, pp. 47–55, 2003. 83

[2] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *Journal of the ACM*, 44:486–504, 1997. 83, 84

[3] J-P. Aubin. *Mathematical Methods of Game and Economic Theory*. North-Holland, 1979. 51

[4] B. Awerbuch, Y. Azar S. Plotkin. Throughput-competitive online routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pp. 32–40, 1993. 83

[5] Y. Azar. On-line load balancing. Lecture Notes in Computer Science, Vol. 1442. Springer-Verlag, pp. 178–195, 1998. 84

[6] B. S. Baker, J. S. Schwartz. Shelf algorithms for two dimensional packing problems. *SIAM Journal on Computing*, 12:508–525, 1983. 83

[7] A. Borodin R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998. 83

[8] L. E. J. Brouwer. Über Abbildung von Manningfaltigkeiten. *Mathematische Annalen*, pp. 97–115. 51

[9] Y. Cho, S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9(1):91–103, 1980. 84

[10] M. Chrobak, L. Larmore. An optimal algorithm for k-servers on trees. *SIAM Journal on Computing*, 20:144–148, 1991. 83

[11] M. Chrobak, L. Larmore. The server problem and on-line games. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 7, pp. 11-64. American Mathematical Society, 1992. 83

[12] M. Chrobak, H. J. Karloff, T. Payne, S. Vishwanathan. New results on the server problem. *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991. 83

[13] J. Csirik, G. Woeginger. On-line packing and covering problems. Lecture Notes in Computer Science, Vol. 1442, pp. 147–177. Springer-Verlag, 1998. 83

[14] J. Csirik, G. J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63:171–175, 1997. 84

[15] A. Darte, Y. Robert, F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser Boston, 2000. 563

[16] D. R. Dooly, S. A. Goldman, S. D. Scott. On-line analysis of the TCP acknowledgement delay problem. *Journal of the ACM*, 48:243–273, 2001. 83, 84

[17] Gy. Dósa, Y. He. Better online algorithms for scheduling with machine cost. *SIAM Journal on Computing*, 33(5):1035–1051, 2004. 84

[18] A. Fiat, Y. Rabani, Y. Ravid. Competitive k-server algorithms. *Journal of Computer and System Sciences*, 48:410–428, 1994. 83

[19] A. Fiat, G. Woeginger (szerkesztők). *Online Algorithms. The State of Art*. Springer-Verlag, 1998. 83

[20] R. Fleischer, M. Wahl. On-line scheduling revisited. *Journal of Scheduling*, 3(6):343–353, 2000. 84

[21] F. Forgó, J. Szép, F. Szidarovszky. *Introduction to the Theory of Games: Concepts, Methods and Applications*. Kluwer Academic Publishers, 1999. 51, 52

[22] R. L. Graham. Bounds for certain multiprocessor anomalies. *The Bell System Technical Journal*, 45:1563–1581, 1966. 84

[23] G. Hadley. *Nonlinear and Dynamic Programming*. Addison-Wesley, 1964. 52

[24] Cs. Imreh. Online strip packing with modifiable boxes. *Operations Research Letters*, 66:79–86, 2001. 84

[25] Cs. Imreh, J. Noga. Scheduling with machine cost. In *Proceedings of APPROX'99*, Lecture Notes in Computer Science, Vol. 1540, pp. 168–176, 1999. 84

[26] D. S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis. 83

[27] D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974. 83

[28] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, R. L. Graham. Worst-case performance-bounds for simple one-dimensional bin packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974. 83

[29] S. Kakutani. A generalization of Brouwer's fixed point theorem. *Duke Mathematical Journal*, 8:457–459, 1941. 51

[30] S. Karamardian. The nonlinear complementarity problems with applications. I, II. *Journal of Optimization Theory and Applications*, 4:87–98 and 167–181, 1969. 52

[31] A. R. Karlin, C. Kenyon, D. Randall. Dynamic TCP acknowledgement and other stories about $e/(e-1)$. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, 502–509. o., 2001. 83

[32] R. M. Karp, R. E. Miller, S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967. 563

[33] E. Koutsoupias, C. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42:971–983, 1995. 83

[34] H. W. Kuhn, A. Tucker (szerkesztők). *Contributions to the Theory of Games. II*. Princeton University Press, 1953. 51

[35] H. T. Kung, C. E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff, G. W. Stewart (szerkesztők), *Sparse Matrix Proceedings, pp. 256–282*. SIAM, 1978. 563

[36] S. Leonardi. On-line network routing. Lecture Notes in Computer Science, Vol. 1442, pp. 242–267. Springer-Verlag, 1998. 83

[37] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publisher, 2001 (fifth edition). 564

[38] M. Manasse, L. McGeoch, D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990. 83

[39] O. Mangasarian, H. Stone. Two-person zero-sum games and quadratic programming. *Journal of Mathematical Analysis and its Applications*, 9:348–355, 1964. 52

[40] B. Martos. *Nonlinear Programming Theory and Methods*. Akadémiai Kiadó, 1975. 51

[41] H. Mills. Equilibrium points of finite games. *SIAM Journal of Applied Mathematics*, 8:397–402, 1976. 51

[42] J. Nash. Noncooperative games. *Annals of Mathematics*, 54:286–295, 1951. 51

[43] J. Neumann, O. Morgenstern. *Theory of Games and Economical Behaviour*. Princeton University Press, 1947 (2. edition). 52

[44] H. Nikaido, K. Isoda. Note on noncooperative games. *Pacific Journal of Mathematics*, 5:807–815, 1955. 51

[45] K. Okuguchi. *Expectation and Stability of Oligopoly Models*. Springer, 1976. 52

[46] K. Okuguchi, F. Szidarovszky. *The Theory of Oligopoly with Multi-Product Firms*. Springer, 1999 (2. kiadás). 52

[47] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture, pp. 208–214*, 1984. 563

[48] S. K. Rao. Regular iterative algorithms and their implementations on processor arrays. Doktori értekezés, Stanford University, 1985. 563

[49] J. Robinson. An iterative method of solving a game. *Annals of Mathematics*, 154:296–301, 1951. 52

[50] J. Rosen. Existence and uniqueness of equilibrium points for concave n-person games. *Econometrica*, 33:520–534, 1965. 52

[51] J. Sgall. On-line scheduling. Lecture Notes in Computer Science, Vol. 1442, pp. 196–231. Springer-Verlag, 1998. 84

[52] H. N. Shapiro. Note on a computation method in the theory of games. *Communications on Pure and Applied Mathematics*, 11:587–593, 1958. 51

[53] D. B. Shmoys, J. Wein, D. P. Williamson. Scheduling parallel machines online. *SIAM Journal on Computing*, 24:1313–1331, 1995. 84

[54] D. Sima, T. Fountain, P. Kacsuk. *Advanced Computer Architectures: a Design Space Approach*. Addison-Wesley Publishing Company, 1998 (2. edition). 564

[55] D. Sleator R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985. 84

[56] F. Szidarovszky, C. Chiarella. Dynamic oligopolies, stability and bifurcation. *Cubo Mathemática Educational*, 3(2):267–284, 2001. 52

[57] F. Szidarovszky, S. Yakowitz. *Principles and Procedures of Numerical Analysis*. Plenum Press, 1998. 51, 52

[58] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–308, 1955. 51

[59] J. Teich, L. Thiele. Control generation in the design of processor arrays. *International Journal of VLSI and Signal Processing*, 3(2):77–92, 1991. 563

[60] A. van Vliet. An improved lower bound for on-line bin packing algorithms. *Information Processing Letters*, 43:277–284, 1992. 83

[61] A. C. C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980. 84

[62] N. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002. 83

[63] E. Zehendner. *Entwurf systolischer Systeme: Abbildung regulärer Algorithmen auf synchrone Prozessorarrays*. B. G. Teubner Verlagsgesellschaft, 1996. 563

[64] S. I. Zuhovitsky, R. A. Polyak, M. E. Primak. Concave n-person games (numerical methods). *Ékonomika i Matematicheskie Methody*, 7:888–900, 1971 (in Russian). 52

[65] A. van Vliet. *Lower and upper bounds for on-line bin packing and scheduling heuristics*. PhD thesis, Erasmus University, Rotterdam, 1995. 83

[66] A. Vestjens. *On-line machine scheduling*. PhD thesis, Eindhoven University of Technology, 1997. 84

Name index

Subject Index

Contents