Evolutionary Computing Solutions for the de Bruijn Torus Problem

Master's Thesis

by

J. B. Kapinya

Supervisor:	Prof. Dr. A. E. Eiben
Co-Supervisor:	Prof. Dr. Antal Iványi
Second Reader:	Dr. Elena Marchiori

VRIJE UNIVERSITEIT

AMSTERDAM

2004

Author

Judit Kapinya

Faculty of Sciences
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam

juditk@elte.hu

Supervisor

Prof. Dr. A. E. Eiben
Professor

Faculty of Sciences
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam

gusz@cs.vu.nl

Co-Supervisor

Prof. Dr. Antal Iványi
Professor

Eötvös Loránd University, Budapest
Department of Computer Algebra
Pázmány Péter sétány 1/c
1117 Budapest

tony@inf.elte.hu

Second Reader

Dr. Elena Marchiori
Assistant Professor

Faculty of Sciences
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam

elena@cs.vu.nl

Contents

1 Introduction

My thesis is based on a mathematical complexity problem, namely the investigation of perfect maps, especially perfect tori. The central issue is the existence of such maps with certain parameters. This issue can be approached by two points of view:

theoretical: Many of construction methods were proposed and many theorems and proofs were produced, as well. These results apply only to certain sets of perfect maps. The conjecture is that such maps exist for every parameter set, but the general proof remains to be seen.

practical: The other approach is to search for certain maps with practical algorithms. The certainty of existence and the possibility for investigating the structure of an existing map may lead nearer to the final solution.

My research is related to the practical approach, but it diverges from the previous attempts: I am going to test the power of evolutionary computing (EC) applied to this complexity problem. There are more arguments for applying EC algorithms. The search space is very large even in case of maps with small parameters, it is out of reach of computer search. In the worst case the time complexity of a branch and bound search algorithm – even if equipped with smart heuristics – is the same as complete enumeration, so it took millions of years to reconnoitre the search space with the recent computational capacity. The EC algorithm is a quite different approach, it provides a means of coping with large and discontinuous search spaces, and furthermore the problem fits into the area of evolutionary computing: the maps have straightforward representations and we can easily define adequate fitness functions, as well.

About the Problem to Solve

The precise definitions of perfect maps can be found in the next section, but the essence in a nutshell is the following. *Perfect Maps* (or *de Bruijn Tori*) are two-dimensional arrays in which every possible rectangular sub-array (of fixed size) occurs precisely once.

The problem from the perspective of system analysis

From the perspective of system analysis there are three main components of a system: inputs, outputs and the model that processes the inputs and returns the corresponding outputs. The book of Eiben and Smith [52] classes the possible problems among three categories: optimization, modeling and simulation problems.

Our mathematical problem is an optimization problem (Figure 1-1):

i) *input* The candidate perfect maps, namely the elements of the search space.

ii) *model* The model is known, that is we know the way to decide whether a map is a perfect one or not.

iii) output The qualification of the input. In case of a "traditional" search algorithm it is "yes" or "no". In case of genetic algorithm it stands for the fitness values.

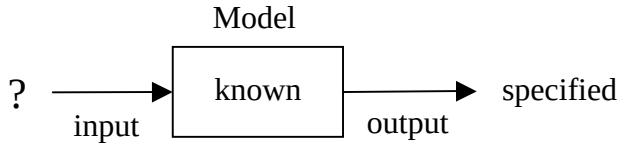


Figure 1-1 Optimization problem

Research Objective

The main goal of my thesis is to demonstrate that evolutionary computing is a useful tool in the investigation of this complexity problem and should be considered before declaring the computational limits of the resolvability. It is a tool that provides the computer with some kind of intelligence hence the computing capacity can be used in a quite different way than before.

Apart from proving the aptitude of EC, there are two more important issues I aim at: to gather all the available information about the foregoing results and to verify them, as well as find such maps whose existence was only a conjecture so far. It is a great challenge, because neither theoretical proofs shore up their existence nor practical attempts, namely no one could construct or simply search for such mathematical objects due to the size of the search space. The detailed survey on my research can be found in section 4.

The Structure of My Thesis

The next section is dedicated to the complexity problem. I tried to compile a concise well-structured survey ("state-of-the-art") on the problem based on the corresponding papers. There were more issues I considered about this survey. First, the literature regarding the problem is quite large and complex. There are many papers devoted to special practical applications (robot self location, pseudorandom arrays, etc.) or features (e.g. decodable maps), while others are interested in construction methods. The really important issues in terms of my thesis are the results concerning the existence question (necessary and sufficient conditions) of the maps, hence I collected and structured only this kind of information. The notations were quite diverse and sometimes confusing, so I aimed at using a uniform and consistent notation based on the papers of Hulbert and Isaak.

Considering the size of the problem it is very tempting to have an eye to the possibility of parallelization, especially if the appropriate hardware is to hand. In the third section I collected the alternative ways to parallelizing an evolutionary algorithm (section 3.1) and I gave a short description of the parallel testing environment, the *DAS-2* (section 3.2).

The fourth section contains my research. It is divided into two main parts: the first one (section 4.1) contains the concise specification of the implemented algorithms (both the reference and the genetic algorithm), and the second one is devoted to the outcomes of the experiments and some theoretical results (section 4.2).

I implemented several algorithms during my work, the list of them can be found in Appendix B. The principles of the certain algorithms are detailed in the corresponding sections and the documentation of the most important application can be found in Appendix A.

2 Theoretical Survey on the Mathematical Problem

2.1 One-Dimensional Case

2.1.1 Perfect Sequences (de Bruijn Cycles)

In one dimension the aperiodic and periodic cases are not clearly distinguished in the literature, because they are barely different from each other and the conversion is trivial between them. The phrases *de Bruijn Cycle* and *de Bruijn Sequence* are equally in use to stand for the periodic case, where the sequence is considered to be wrapped round on itself. This corresponds to writing it on the outside of a cylinder.

DEFINITION 1 *A $(k^n;n)_k$ - de Bruijn Cycle is a cyclic k-ary sequence of length k^n with the property that every k-ary n-tuple appears exactly once contiguously on the cycle. The parameter n is often called the span of the sequences.*

[0 0 0 1 0 1 1 1]

Figure 2-1 A $(8;3)_2$ - de Bruijn Cycle

REMARK 2 *In a cycle there are two directions and they need to be considered as different in spite of the fact that they represent the same cycle. For example in Figure 2-2 we have two cycles:* [0 0 0 1 0 1 1 1] *(found clockwise) and* [0 0 0 1 1 1 0 1] *(found counter-clockwise).*

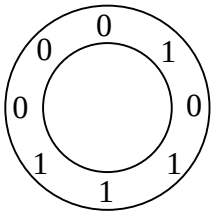


Figure 2-2

THEOREM 3 *A $(k^n;n)_k$ - de Bruijn Cycle exists for every k and n ($k \geq 2$ and $n \geq 1$).*

Such cycles were first discovered in 1894 by Flye-Sainte Marie [1], and rediscovered in 1946 by de Bruijn [2] and Good [3]. An excellent survey on the topic by Fredricksen can be found in [4].

De Bruijn Cycles have applications in the study of position-detection [5, 6, 7, 8, 9, 10, 11, 12], pseudorandom numbers, cryptography, nonlinear shift registers and coding theory, and a vast literature exists [13, 14, 15, 16, 17].

2.1.2 The Decoding Problem

The decoding problem is to discover the position any specified *n*-tuple within a particular sequence. In spite of its importance [5, 8, 30] it has been much less well studied than the construction problem.

A summary of the previous work and two new methods for construction of de Bruijn Cycles (which have the advantage that they can be decoded very efficiently) can be found in [18].

2.1.3 Infinite Perfect Sequences

2.1.3.1 Superperfect Sequences

DEFINITION 1 *A* $(k^n;n)_k$ *- Superperfect Sequence is a Perfect Sequence whose* k^n *length prefixes are* $(k^n;n)_k$ *- de Bruijn Cycles for* $n=1,2,\ldots.$

In 1984 N. Vörös [19] gave a sufficient condition for the existence of such sequences.

2.1.3.2 Growing Sequences

DEFINITION 1 *A* $(k^n;n)_k$ *- Growing Sequence is a Perfect Sequence whose* k^n *length prefixes are* $(k^n;n)_k$ *- de Bruijn Cycles for* $k=1,2,\ldots.$

DEFINITION 2 *Let* $\bar{k}=\langle k_1k_2\ldots\rangle$ *be a strictly increasing sequence of positive integers. An* $(\bar{k}^n;n)_{\bar{k}}$ *- Growing Sequence is a Perfect Sequence whose* k_i^n *length prefixes are* $(k_i^n;n)_{k_i}$ *- de Bruijn Cycles for* $i=1,2,\ldots.$

REMARK 3 *This is the one-dimensional equivalent of a more general definition that can be found in Section 2.3.1.2.1.*

Hurlbert and Isaak [45] in 1994 constructed a Growing Sequence for the case when $\bar{k}$ is the sequence of the even number. Then years later, Horváth and Iványi [21] proved the following

LEMMA 4 *If* $n\geq 1$ *and* $k\geq 1$ *then any* $(k^n;n)_k$ *- de Bruijn Cycle can be continued in order to get a* $((k+1)^n;n)_{k+1}$ *- de Bruijn Cycle.*

This lemma yields [21] the following

THEOREM 5 *If* $n\geq 1$ *and* $\bar{k}=\langle k_1k_2\ldots\rangle$ *with* $k_i=i$*, then exists a* $(\bar{k}^n;n)_{\bar{k}}$ *- Growing Sequence.*

The most general result (see section 2.3.1.2.1) can be found in [20].

2.1.3.3 Alternating Sequences

Alternating Sequences are hybrids of the previously mentioned two kind of infinite sequences. The proof of their existence can be found in [21].

DEFINITION 1 *An Alternating Sequence is a Perfect Sequence whose i^i length prefixes are $(i^i;i)_i$ - de Bruijn Cycles and $(i+1)^i$ length prefixes are $((i+1)^i;i)_{i+1}$ - de Bruijn Cycles for $i=1,2,\ldots$.*

2.1.4 Perfect Factors (Equivalence-Class de Bruijn Cycles)

Perfect Factors are related objects introduced by Etzion [22] and later by Hurlbert and Isaak [40] as *Equivalence-Class de Bruijn Cycles*. Perfect Factors have proved useful in constructions for de Bruijn Tori and have been extensively studied in [23, 24, 25, 26].

DEFINITION 1 *An $(R;n;T)_k$ - Perfect Factor is a set of $T=k^n/R$ k-ary, period R sequences in which every k-ary n-tuple occurs exactly once as a subsequence. The parameter n is often called the span of the sequences.*

$$\left\{\begin{matrix}[0 & 0 & 0 & 1 & 2 & 2 & 1 & 2 & 1], \\ [1 & 1 & 1 & 2 & 0 & 0 & 2 & 0 & 2], \\ [2 & 2 & 2 & 0 & 1 & 1 & 0 & 1 & 0]\end{matrix}\right\}$$

Figure 2-3 A $(9;3;3)_3$ - Perfect Factor

REMARK 2 *Perfect Factors are generalizations of the classical de Bruijn Cycles: a de Bruijn Cycle is a Perfect Factor with $T=1$, that is a $(k^n;n;1)_k$ - Perfect Factor.*

The following necessary conditions for the existence of a Perfect Factor were formulated in [25].

LEMMA 3 *Suppose A is an $(R;n;T)_k$ - Perfect Factor. Then*

i) $R \mid k^n$, *and*

ii) $n < R \le k^n$ *or* $(R=n=1)$.

CONJECTURE 4 *The conditions of Lemma 3 are sufficient for the existence of an $(R;n;T)_k$ - Perfect Factor.*

Etzion [22] has shown that the conjecture holds in the binary case. This was extended to cases where *k* is a prime power by Paterson [26]. Mitchell and Paterson [27] have shown the sufficiency for the case when $n=R-1$.

In the view of the first condition in Lemma 3 we can assume that the prime factorizations of *k* and *R* are:

$$k=\prod_{i=1}^{n} p_i^{k_i} \text{ and } R=\prod_{i=1}^{n} p_i^{r_i} \text{ where } 0 \le r_i \le k_i n \text{ for each } i.$$

It was also proved in [26] that the conditions of Lemma 3 are sufficient when $p_i^{s_i} > n$ for every index *i*. In [23] this result has been improved to establish the sufficiency of the conditions of Lemma 3 whenever $p_i^{s_i} > n$ for at least one index *i*.

Mitchell has shown that the conjecture holds if $n < 5$ [24] and that Perfect Factors exist for all triples $(R,6,k^6/R)_k$ satisfying the conditions of Lemma 3 with some possible exceptions [23]. He obtained Perfect Factors for some of those exceptions in [25].

This result was extended to $n < 7$ in [27]. Regarding the cases $n = 7$ and 8, unresolved parameter sets and remarks can be found in [27].

Mitchell [24] has shown that the following Perfect Factors exists:

i) $(6,3,6^2 d^3)_{6d}$ - Perfect Factor $(d \geq 1)$,
ii) $(10,3,10^2 d^3)_{10d}$ - Perfect Factor $(d \geq 1)$ and
iii) $(30,3,30^2 d^3)_{30d}$ - Perfect Factor $(d \geq 1)$.

2.1.5 Perfect Multi-Factors

Mitchell introduced two auxiliary classes of combinatorial objects: *Perfect Multi-Factors* [23] and *Generalized Perfect Factors* [24] (see section 2.1.6), which can be combined in various ways to yield Perfect Factors.

DEFINITION 1 *Suppose R, m, n and k are positive integers satisfying* $R \mid k^n$ *and* $k \geq 2$. *An* $(R;n;T;m)_k$ *- Perfect Multi-Factor is a set of* $T = k^n / R$ *k-ary, period Rm sequences with the property that for every k-ary n-tuple* τ*, and for every integer j in the range* $0 \leq j < m$, τ *occurs at a position* $p \equiv j \pmod m$ *in one of these sequences.*

$$\left\{\begin{matrix}[0 & 0 & 0 & 0 & 1 & 1 & 1 & 1], \\ [1 & 0 & 0 & 0 & 0 & 1 & 1 & 1], \\ [0 & 1 & 0 & 1 & 1 & 0 & 1 & 0], \\ [0 & 0 & 1 & 0 & 1 & 1 & 0 & 1]\end{matrix}\right\}$$

Figure 2-4 A $(4;4;4;2)_2$ - Perfect Multi-Factor

REMARK 2 An $(R;n;T;1)_k$ - Perfect Multi-Factor is precisely equivalent to an $(R;n;T)_k$ - Perfect Factor.

The following necessary conditions for the existence of a Perfect Multi-Factor were formulated in [23].

LEMMA 3 *Suppose A is an* $(R;n;T;m)_k$ *- Perfect Multi-Factor. Then*

i) $R \mid k^n$, *and*
ii) $n < Rm$ *or* $(n = m$ *and* $R = 1)$.

It has been shown [23] that the above necessary conditions are sufficient if $m \geq n$.

2.1.6 Generalized Perfect Factors

DEFINITION 1 *Suppose R, m, n and k are positive integers satisfying* $R \mid k^n$ *and* $k \geq 2$. *An* $(R;n;T;m)_k$ *- Generalized Perfect Factor is a set of* $T = k^n / R$ *k-ary, period* $R \cdot m$ *sequences with the property that for every k-ary n-tuple* τ*, there exists an integer j in the range* $0 \leq j < R$ *such that for every i* $(0 \leq i < m)$ τ *occurs at position* $j + iR$ *in one of these sequences.*

REMARK 2

i) *An* $(R;n;T;1)_k$ *- Generalized Perfect Factor is precisely equivalent* to an $(R;n;T)_k$ *- Perfect Factor, and*

ii) *An* $(1;n;T;m)_k$ *- Generalized Perfect Factor is precisely equivalent* to an $(1;n;T;m)_k$ *- Perfect Multi-Factor.*

The following necessary conditions for the existence of a Generalized Perfect Factor were formulated in [24].

LEMMA 3 *Suppose A is an* $(R;n;T;m)_k$ *- Generalized Perfect Factor. Then*

i) $R \mid k^n$, *and*

ii) $n < Rm$ *or* $(n = m$ *and* $R = 1)$.

These necessary conditions for the existence are not sufficient [24], but there are some (constructive) existence results for Generalized Perfect Factors in [24, 27].

2.1.7 De Bruijn Graphs

DEFINITION 1 *Let* $K = \{0,1,\ldots,k-1\}$ *be and alphabet and let* K^n *denote the set of n-tuples. A* (k,n) *- de Bruijn Graph is a graph with vertex set* K^n *and edge set* K^{n+1} *so that if* $e = < x_1 x_2 \ldots x_{n+1} > \in K^{n+1}$ *then e determines a directed edge going from the vertex* $< x_1 x_2 \ldots x_n >$ *to the vertex* $< x_2 x_3 \ldots x_{n+1} >$.

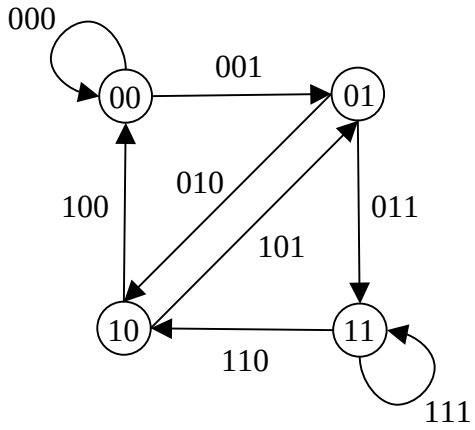


Figure 2-5 A $(2,2)$ - de Bruijn Graph

Since a (k,n) - de Bruijn Graph is connected and each vertex has *k* ingoing and *k* outgoing edges, it has an Euler path [28]. Note that an Euler path in a (k,n) - de Bruijn Graph is equivalent to a $(k^{n+1},n+1)_k$ - de Bruijn Cycle.

The number of distinct Euler paths in a de Bruijn Graph is equal to $\Delta((k-1)!)^{k^n}$, where Δ denotes the number of spanning trees of the graph [28]. Considering that the in-degree matrix contains at most $k^n(k+1)$ non-zero elements, this number can be determined in $O(k^{n+1}!)$ time. Even the best non-approximating algorithm (Gaussian elimination) needs $O(k^{3n})$ time, which is still exponential.

An easily applicable equivalent formula with the specialty that it does not require any knowledge about graph theory and can be applied in $\Theta(n+k)$ time, is given in Section 4.2.6.2.

2.2 Two-Dimensional Case

2.2.1 Aperiodic Perfect Maps

In the aperiodic case the array is deemed to be written onto a planar surface and the sub-arrays are always completely within the borders of the array.

DEFINITION 1 *An* $(R,S;m,n)_k$ *- Aperiodic Perfect Map is a k-ary* $(R\times S)$ *toroidal array with the property that every k-ary* $(m\times n)$ *array occurs exactly once in the set of* $(m\times n)$ *aperiodic sub-arrays. The pair* (*m, n*) *is often called the window of the map.*

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Figure 2-6 A $(3,9;2,2)_2$ - Aperiodic Perfect Map

LEMMA 2 *If A is a k-ary* $(R,S;m,n)_k$ *- Aperiodic Perfect Map then*

i) $R \geq m \geq 1$,

ii) $S \geq n \geq 1$, *and*

iii) $(R-m+1)(S-n+1) = k^{mn}$

In [41], C. J. Mitchell proved the binary case of the following

CONJECTURE 3 *The necessary conditions of Lemma 2 on R, S, m, n are sufficient for the existence of a k-ary* $(R,S;m,n)_k$ *- Aperiodic Perfect Map.*

2.2.2 Periodic Perfect Maps (or de Bruijn Tori)

In the periodic case the array is considered to be wrapped round on itself. This corresponds to writing the array onto a torus. Sub-arrays then exist starting at any point in the array, which no longer has any "edges".

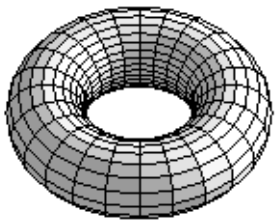


Figure 2-7 A torus

These periodic structures can be transformed very simply into corresponding Aperiodic (see section 2.2.1) and Semi-Periodic (see section 2.2.3) Perfect Maps. However, Aperiodic and Semi-Periodic Perfect Maps can exist for parameter sets for which the corresponding Periodic Perfect Maps cannot [41].

DEFINITION 1 *An* $(R,S;m,n)_k$ *- de Bruijn Torus (or Periodic Perfect Map) is a k-ary* $(R \times S)$ *toroidal array with the property that every k-ary* $(m \times n)$ *array occurs exactly once as a periodic sub-array of the array. The pair* (m,n) *is often called the window or order, and* (R,S) *the period of the torus.*

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Figure 2-8 A $(4,4;2,2)_2$ - de Bruijn Torus

REMARK 2 *The* $(R,1;m,1)_k$ *- de Bruijn Tori are de Bruijn Cycles.*

De Bruijn Tori have interesting applications in robot self-location [29, 30], pseudorandom arrays [22, 31, 32, 33], and the design of mask configurations for spectrometers [34]. (For an interesting variation on this theme see [35]). Even cloth patterns have used these designs, long before their mathematical properties were discovered [36].

In 1984, Ma [37] proved the binary case of the following 1988 theorem of Cock [38] (see also [39]).

THEOREM 3 *For all m, n and k (except $n = 2$ if k even) there is a $(k^r, k^s; m, n)_k$ - de Bruijn Torus with $r = m$ and $s = m(n-1)$.*

2.2.2.1 The Necessary Conditions of the Existence

The necessary conditions of the following Lemma were mentioned by Hurlbert and Isaak in [40] and by Mitchell in [41].

LEMMA 4 *If A is a k-ary $(R, S; m, n)_k$ - de Bruijn Torus then*

i) $R > m \geq 1$ *or* $R = m = 1$,
ii) $S > n \geq 1$ *or* $S = n = 1$, *and*
iii) $RS = k^{mn}$

2.2.2.2 The Sufficient Conditions of the Existence

Paterson [42] showed that in the binary case the necessary conditions of the Lemma 4 are in fact sufficient for the existence of de Bruijn Tori. In [43] he extended his work to alphabets of prime-power size.

CONJECTURE 5 *If R, S, m, n and k satisfy*

i) $R > m$,
ii) $S > n$, *and*
iii) $RS = k^{mn}$

then there is an $(R, S; m, n)_k$ - de Bruijn Torus.

Hurlbert and Isaak [44] produced tori for which the period is not a power of *k*:

THEOREM 6 *Let k have prime factorization $\prod p_i^{\alpha_i}$ and let $q = k \prod p_i^{\lfloor \log_{p_i} m \rfloor}$. Then for all m, n there is a $(q, k^{mn}/q; m, n)_k$ - de Bruijn Torus.*

In [45] they proved a sub-case (see Theorem 1 in Section 2.2.2.4) with the hope that it will help to extend this result.

The most progress toward the previous conjecture by Paterson [46] is the following

THEOREM 7 *Suppose k, R and S have prime factorizations as follows:*

$k = \prod_{i=1}^{n} p_i^{k_i}$, $R = \prod_{i=1}^{n} p_i^{r_i}$ *and* $S = \prod_{i=1}^{n} p_i^{s_i}$ *for some* $0 \le r_i \le k_i mn$ *where* $s_i = k_i mn - r_i$, $R > m$ *and* $S > n$. *And that for some i we have* $p_i^{r_i} > m$ *and* $p_i^{s_i} > n$. *Then there exists an* $(R, S; m, n)_k$ *- de Bruijn Torus.*

This prompted Hurlbert, Mitchell and Paterson [47] to examine the parameter sets where $p_i^{r_i} \le m$ for some indices and $p_i^{s_i} \le n$ for other indices in the case where $m = n = 2$. They developed new construction methods for some sub-cases (see Theorem 2 and Theorem 3 in Section 2.2.2.4) and with the combination of those cases obtained the following

THEOREM 8 *The necessary conditions of Lemma 4 are sufficient for the existence of an* $(R, S; 2, 2)_k$ *- de Bruijn Torus.*

2.2.2.3 A Special Case: de Bruijn Square

In 1992 Chung, Diaconis and Graham [48] asked whether it is possible that "square" tori exist for even *n*. That is, can it be that $R = S$ and $m = n$? This question was resolved for the binary case by Fan, Fan, Ma and Siu [49], who proved

THEOREM 1 *There exist a* $(2^r, 2^r; n, n)_2$ *- de Bruijn torus if and only if n is even (of course,* $r = n^2/2$*).*

Hurlbert and Isaak [40] settled the question for general *k* with the following

THEOREM 2 *Except in the case that k is an even square and* $n = 3, 5, 7$ *or* 9*, there is an* $(R, R; n, n)_k$ *- de Bruijn Torus if and only if n is even or k is a perfect square.*

In [46] Paterson made up for the mission cases ($n = 3, 5, 7$ *and* 9), so previous theorem reads as follows.

THEOREM 3 *There is an* $(R, R; n, n)_k$ *- de Bruijn Torus if and only if n is even or k is a perfect square.*

2.2.2.4 Some Constructions for Sub-Cases

THEOREM 1 *For all s and t there is a* $(4st^2, 4s^3t^2; 2, 2)_{2st}$ *- de Bruijn Torus.*

THEOREM 2 *Suppose* $m > n \ge 2$. *Then there exists an* $(m^4, n^4; 2, 2)_{mn}$ *- de Bruijn Torus.*

THEOREM 3 *Suppose* $n > 2$ *is odd. Then for every* $k \ge 1$, *there exists a* $(2n^4, 2^{4k-1}; 2, 2)_{2^k n}$ *- de Bruijn Torus.*

2.2.3 Semi-Periodic Perfect Maps

In the semi-periodic case the array is considered as periodic in one dimension and aperiodic in the other. This corresponds to writing the array onto the outside of a cylinder.

DEFINITION 1 *An* $(R,S;m,n)_k$ *- Semi-Periodic Perfect Map is a k-ary* $(R\times S)$ *toroidal array with the property that every k-ary* $(m\times n)$ *array occurs exactly once in the set of* $(m\times n)$ *semi-periodic sub-arrays. The pair* (m,n) *is often called the window of the map.*

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2-9 A $(3,8;2,2)_2$ - Semi-Periodic Perfect Map

LEMMA 2 *If A is a k-ary* $(R,S;m,n)_k$ *- Semi-Periodic Perfect Map then*

i) $R \geq m \geq 1$,

ii) $S > n \geq 1$ *or* $S = n = 1$, *and*

iii) $(R-m+1)S = k^{mn}$

In [41], C. J. Mitchell proved the binary case of the following

CONJECTURE 3 *The necessary conditions of Lemma 2 on R, S, m, n are sufficient for the existence of a k-ary* $(R,S;m,n)_k$ *- Semi-Periodic Perfect Map.*

2.2.4 The Decoding Problem

As already mentioned with reference to the one-dimensional case in Section 2.1.2, Perfect Maps play a significant role in many applications, especially in position location [30, 50]. Decoding means a method for computing the position of a given sub-array within a Perfect Map. In [50] we can found methods for constructing Perfect Maps, which can be decoded efficiently. Some remark on the efficiency of other methods can be found in [41].

2.2.5 Infinite Perfect Maps

The definitions of the two-dimensional *Growing Perfect Maps* and *Alternating Perfect Maps* can be easily generalized from their one-dimensional equivalent (see section 2.1.3). For the most general definition see section 2.3.2.

The following Infinite Perfect Maps can be considered as two-dimensional interpretations of the Superperfect Sequences.

2.2.5.1 Increasing Perfect Maps

DEFINITION 1 *An* $(R, S(x); m, n(x))_k$ *- Increasing Perfect Map is a Perfect Map with the property that every prefix of the map is a* $(R, S(x); m, n(x))_k$ *- de Bruijn Torus, where* $n(x) = x$ *and* $S(x) = k^{mx} / R$ *for* $x = 1,2,\ldots.$

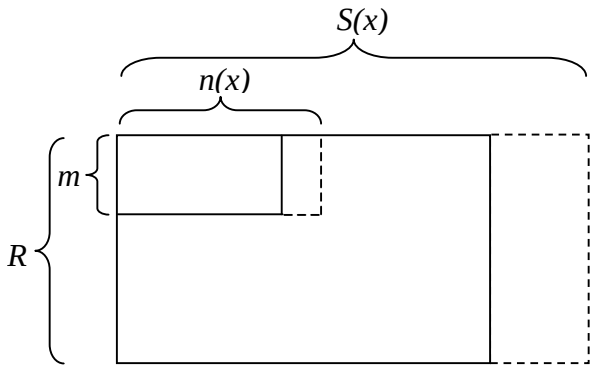


Figure 2-10 Sketch of an Increasing Perfect Map

2.2.5.2 Expanding Perfect Maps

DEFINITION 1 *An* $(R(c,x), S(c,x); m(x), n(c,x))_k$ *- Expanding Perfect Map is a Perfect Map with the property that every prefix of the map is a* $(R(c,x), S(c,x); m(x), n(c,x))_k$ *- de Bruijn Torus (*$c \geq 0$*), where* $m(x) = x$, $n(c,x) = c + x$, $S(c,x) = k^{m(x-1)n(c,x)} / R(c, x-1)$ *and* $R(c,x) = k^{m(x)n(c,x)} / S(c,x)$ *for* $x = 1,2,\ldots.$

REMARK 2 *Expanding consists of two consecutive steps: first increasing the Perfect Map in one direction, then increasing it in the other direction.*

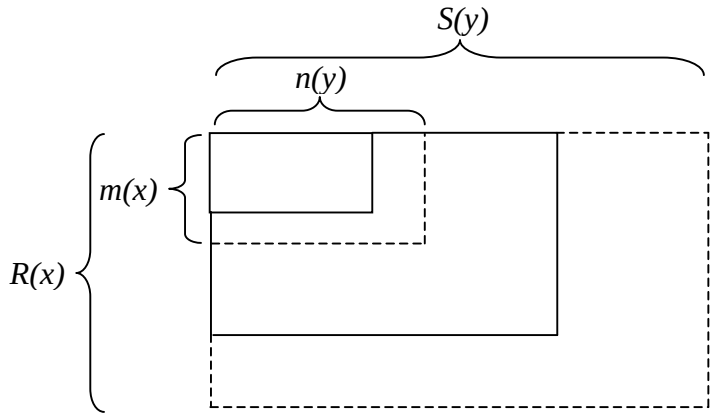


Figure 2-11 Sketch of an Expanding Perfect Map

2.2.6 Perfect Factors

DEFINITION 1 *An* $(R,S;u,v;T)_k$ *- Perfect Factor is a set of* T $R\times S$ *periodic arrays, with symbols drawn from a set of size k, having the property that every possible* $u\times v$ *array occurs exactly once as a periodic sub-array in precisely one of the arrays.*

REMARK 2 *An* $(R,S;u,v;1)_k$ *- Perfect Factor is simply an* $(R,S;u,v)_k$ *- de Bruijn Torus.*

Hurlbert, Mitchell and Paterson [47] obtained a complete answer for the necessary and sufficient conditions of the existence in the case where *k* is a prime-power:

THEOREM 3 *Let p be a prime and k, r, s and t be integers. The conditions that* $p^r, p^s > 2$ *and* $r+s+t=4k$ *are necessary and sufficient for the existence of a* $(p^r, p^s;2,2;p^t)_{p^k}$ *- Perfect Factor.*

2.3 Higher dimensions

2.3.1 De Bruijn *d*-Tori

DEFINITION 1 *Let* $\overline{R}=(r_1,...,r_d)$ *and* $\overline{n}=(n_1,...,n_d)$ *with* $r_i > n_i$ *and* $\prod r_i = k^{\prod n_i}$. *We call a d-dimensional toroidal k-ary block an* $(\overline{R};\overline{n})_k^d$ *- de Bruijn Torus if it has dimensions* $r_1\times\cdots\times r_d$ *and every k-ary* $n_1\times\cdots\times n_d$ *block appears exactly once contiguously in the d-dimensional torus.*

DEFINITION 2 *A fundamental block of an* $(\overline{R};\overline{n})_k^d$ *- de Bruijn Torus is an array consisting of* r_i *consecutive rows in the* i^{th} *dimension for* $i=1, 2, \ldots, d$. *Repeating such a block produces the torus.*

REMARK 3 *A matrix appears uniquely in an infinite periodic array if it appears uniquely in a fundamental block.*

One then has the following theorem, mentioned in [38] and proved in [44].

THEOREM 4 *For all* $\overline{n}$, d *and* k *there is an* $\overline{R}$ *so that there is an* $(\overline{R};\overline{n})_k^d$ *- de Bruijn Torus (except that* $n_i=2$ *for at most one index* i *when* k *is even) with the following properties:*

$$r_1 = k^{n_1} \text{ and } r_j = \left(\prod_{i=1}^{j-1} r_i\right)^{n_j-1} = k^{(n_j-1)\prod_{i=1}^{j-1} n_i}$$

REMARK 5 *So Cock's technique* [38] *easily generalizes to higher dimensions, but unfortunately, each new dimension has size exponential in the previous.*

CONJECTURE 6 *If k,* $\overline{R}$ *and* $\overline{n}$ *satisfy*

i) $r_i > n_i$ *or* $r_i = n_i = 1$ *for all* $1 \le i \le d$ *and*

ii) $\prod_{i=1}^{d} r_i = k^{\prod_{i=1}^{d} n_i}$

then there is an $(\overline{R};\overline{n})_k^d$ *- de Bruijn Torus.*

2.3.1.1 A Special Case: de Bruijn *d*-Cubes

Hurlbert and Isaak [40] assumed that Conjecture 6 is true for $n_1 = \cdots = n_d = n$ and $r_1 = \cdots = r_d = k^{n^d/d}$, that is *de Bruijn d-Cubes*. In [20] Horváth and Iványi constructed the smallest possible (a $256 \times 256 \times 256$ sized *8*-ary) 3-Cube.

2.3.1.2 Infinite de Bruijn *d*-Cubes

2.3.1.2.1 Growing de Bruijn d-Cubes

In [20] Horváth and Iványi proposed the following definitions and proved Theorem 3.

DEFINITION 1 *Let* $\overline{k} = \langle k_1 k_2 \ldots \rangle$ *be a strictly increasing sequence of positive integers. A* $(\overline{k}^{n^d/d};n)_{\overline{k}}^d$ *- Growing de Bruijn Cube is a de Bruijn d-Cube whose prefixes are* $(k_i^{n^d/d};n)_{k_i}^d$ *- de Bruijn Cubes for* $i = 1,2,\ldots$.

DEFINITION 2 *For* $n,k \ge 2$ *the new alphabet size* $K(k,n)$ *is*

$$K(k,n) = \begin{cases} k, & \text{if any prime divides } k, \\ kq, & \text{otherwise,} \end{cases}$$

where q is the product of prime divisors of n not dividing k.

THEOREM 3 *If* $d \ge 1$, $n \ge 2$, $k \ge 2$ *and* $k_i = N^{\frac{di}{\gcd(d,n^d)}}$ *for* $i = 1,2,\ldots$ *then exists a* $(\overline{k}^{n^d/d};n)_{\overline{k}}^d$ *- Growing de Bruijn Cube.*

2.3.2 Infinite de Bruijn *d*-Tori

2.3.2.1 Increasing de Bruijn *d*-Tori

DEFINITION 1 *An* $(\overline{R}(x);\overline{n}(x))_k^d$ *- Increasing de Bruijn Torus is a de Bruijn d-Torus with the property that every* $\overline{R}(x)$ *sized prefix of the torus is an* $(\overline{R}(x);\overline{n}(x))_k^d$ *- de Bruijn Torus, where* $\overline{n}(x) = \langle n_1, n_2, \ldots, n_{d-1}, x \rangle$ *and* $\overline{R}(x) = \langle r_1, r_2, \ldots, r_{d-1}, n^{f_1 f_2 \cdots f_{d-1} x} / r_1 r_2 \ldots r_{d-1} \rangle$ *for* $x = 1,2,\ldots$.

2.3.2.2 Expanding de Bruijn *d*-Tori

DEFINITION 1 *An* $(\bar{R}(x);\bar{n}(x))_k^d$ *- Expanding de Bruijn Torus is a de Bruijn d-Torus with the property that every prefix of the torus is an* $(\bar{R}(x);\bar{n}(x))_k^d$ *- de Bruijn Torus, where* $n_i(x) = c_i + x$ *(*$c_1 = 0$*,* $c_i \geq 0$ *for* $i = 2,3,\ldots$*) and*

$$r_i(x) = k^{\prod_{j=1}^{i} n_j(x) \prod_{j=i+1}^{d} n_j(x-1)} / \prod_{j=1}^{i} r_j(x) \prod_{j=i+1}^{d} r_j(x-1) \text{ for } x = 1,2,\ldots.$$

2.3.2.3 Developing de Bruijn *d*-Tori

DEFINITION 1 *Let* $\bar{n} = \langle n_1 n_2 \ldots \rangle$ *be a sequence of positive integers. An* $(\bar{R};\bar{n})_k^d$ *- Developing de Bruijn Torus is a de Bruijn d-Torus with the property that every i-dimensional prefix of the torus is an* $(\bar{R};\bar{n})_k^i$ *- de Bruijn Torus, where* $r_j = k^{\prod n_l} / \prod_{l \neq j} r_l$ *for* $j = 1,2,\ldots,i$*.*

2.3.2.4 Growing de Bruijn *d*-Tori

DEFINITION 1 *Let* $\bar{k} = \langle k_1 k_2 \ldots \rangle$ *be a strictly increasing sequence of positive integers. An* $(\bar{R}(\bar{k});\bar{n})_{\bar{k}}^d$ *- Growing de Bruijn Torus is a de Bruijn d-Torus with the property that every prefix of the torus is an* $(\bar{R}(k_i);\bar{n})_{k_i}^d$ *- de Bruijn Torus, where* $r_j(k_i) = k_i^{\prod n_k} / \prod_{k \neq j} r_k$ $(j = 1,\ldots,d)$ *for* $i = 1,2,\ldots.$

2.3.2.5 Alternating de Bruijn *d*-Tori

DEFINITION 1 *An* $(\bar{R};\bar{n})_{\bar{k}}^d$ *- Alternating de Bruijn Torus is a de Bruijn d-Torus with the property that every* i^i *sized prefix of the torus is an* $(\bar{R};\bar{n})_i^d$ *- de Bruijn Torus with* $\prod n_j = i$*, and every* $(i+1)^i$ *sized prefix is an* $(\bar{R};\bar{n})_{i+1}^d$ *- de Bruijn Torus with* $\prod n_j = i+1$*, for* $i = 1,2,\ldots.$

2.3.3 Perfect Factors (de Bruijn Families)

DEFINITION 1 *A d-dimensional k-ary, order* $\bar{n}$ *Perfect Factor (or de Bruijn Family) of size t and period* $\bar{R}$ *is a family* $\{B_1,\ldots,B_t\}$ *of d-dimensional k-ary toroidal arrays, of period* $\bar{R}$ *each, with the property that for every d-dimensional k-ary matrix M of size* $\bar{n}$ *there is a unique j and a unique* $\bar{i}$ *so that M appears in* B_j *at position* $\bar{i}$*. (We will say that a particular matrix M of size* $\bar{n}$ *appears in B at a position* $\bar{i} = \langle i_1,\ldots,i_d \rangle$ *if M appears in the positions* $\bar{i}$ *through* $\bar{i} + \bar{n}$*.) We call such a Perfect Factor an* $(\bar{R};\bar{n};t)_k^d$ *- Perfect Factor.*

REMARK 2 *In the case that* $d = t = 1$*, Perfect Factors have been called de Bruijn Cycles. Perfect Factors with* $t = 1$ *and* $d > 1$ *have been called de Bruijn Tori (or Perfect Maps).*

Hurlbert and Isaak [51] obtained the following

THEOREM 3 Let $k = \prod_{i=1}^{s} p_i^{\alpha_i}$ for primes p_i and for $j \le d$ suppose that $r_j = \prod_{i=1}^{s} p_i^{\beta_{i,j}}$ with each $p_i^{\beta_{i,j}} > 2$. Further assume that for each $i \le s$ there is a permutation $\sigma_i = (\sigma_{i,1}, \cdots, \sigma_{i,d})$ of $\{1, \ldots, d\}$ so that for each $l \le d$ we have $\sum_{j=1}^{l} \beta_{i,\sigma_{i,j}} \le \alpha_i 2^l$. Then there is an $(\overline{R}; \overline{n}; t)_k^d$ - Perfect Factor, where each $n_i = 2$.

3 Tools for Parallelizing the Algorithm

3.1 The Available Evolutionary Computing Tools

The following two models were specified by Eiben and Smith [52].

3.1.1 Island Model

The principle of the Island Model is that we have multiple populations in parallel. They exist and evolve independently from one another; each one is a separate “island”. Sometimes individuals are moving from a population to another neighbouring one, this process is called migration. Its mechanism is illustrated in Figure 3-1, where we have three populations with three individuals migrating, one from island 2 to1 and two from island 3 to 2.

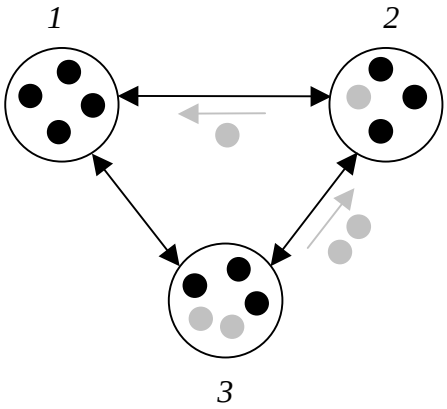


Figure 3-1 Sketch of the Island Model

Migration takes place after an *epoch*, namely a number of generations. While the populations are evolving independently from the others, they are exploring a certain part of the search space, namely they are exploiting that area. If a new individual gets into the population, it can direct the search into other (maybe fitting) directions and by this means expand the space searched so far, hence facilitating exploration.

Basic parameters and some recommendations to consider:

i) How long should be an epoch? Its length is usually fixed, but we have countless possibilities to plant it into the evolutionary mechanism and make it depend on some other parameter or feature of the populations.

ii) How many individuals to exchange? If we exchange a large number of individuals, the populations may converge to the same solution too rapidly, and we will have a lot of populations producing the same results, consuming time and capacity unnecessarily.

iii) Which individuals to exchange? The selection may carry out on the basis of fitness, or it can be simply a random choice. In the latter case it is less likely that a population will be took over by a new high-fitness migrant.

iv) How to initialize the different populations? It is not guaranteed that the different populations are exploring different regions of the search space, that's why we have to be very cautious and apply some refined heuristics during the initialization process.

It is possible to maintain different populations with different parameters, like the continents have different features in real life.

3.1.2 Diffusion Model

The principle of the Diffusion Model is that we have multiple overlapping subpopulations in parallel. The members of the populations can be considered being scattered over on a toroidal grid, and communicating only with individuals in their neighbourhood. Communication means the applicability of the recombination and selection operators in this context. This mechanism is illustrated in Figure 3-2, where the black individual in the middle communicates exclusively with the grey ones in its immediate vicinity.

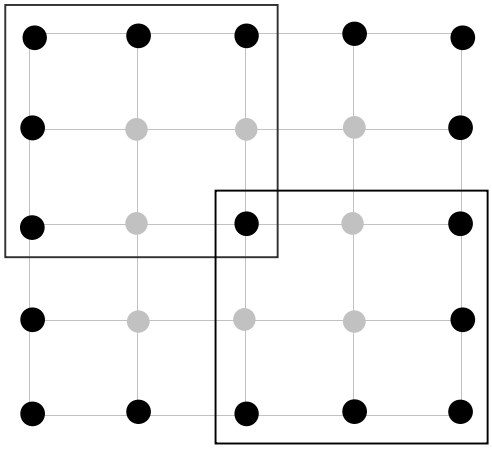


Figure 3-2 Sketch of the Diffusion Model

Basic parameters and some recommendations to consider:

i) How large should be a neighbourhood? The size of the neighbourhood is usually the same for all nodes, but we can make it depend on some feature of the individual, by so doing the populations turn into some kind of realistic community, where the individuals are making friends with each other: there are timid ones with smaller vicinity and social ones with larger vicinity.

ii) Which element to replace? Owning to the overlapping feature of the subpopulations we have to be very careful when applying the replacement operator. If both subpopulations want to replace the same individual, race conditions may occur. This situation is illustrated in Figure 3-2, where two subpopulations indicated by black frames want to replace the same individual in their intersection. One possible solution is to replace the central node of a subpopulation.

3.2 The Parallel Testing Environment

The system where I run my parallel applications is called *DAS-2 (Distributed ASCI Supercomputer 2)*. It was designed by the ***A**dvanced **S**chool for **C**omputing and **I**maging*, a cooperation between a number of Dutch Universities. The machine is built of clusters of workstations, which are interconnected by *SurfNet*, the Dutch university Internet backbone for wide-area communication. The nodes within a local cluster are connected by a *Myrinet-2000* network, a popular high-speed LAN. The system was built by *IBM* and runs the *Red Hat Linux* operating system. The clusters are located at five Dutch Universities, there are 200 nodes altogether. I use only one cluster of 72 nodes, located at the Vrije Univeristeit.

Each node contains:

- Two 1 GHz Pentium IIIs
- At least 1 GB RAM (2 GB for two "large" nodes)
- A 20 GByte local IDE disk
- A Myrinet interface card
- A Fast Ethernet interface card

Each cluster consists of a file/compile server (called *fs0* that of VU) and a number of compute nodes. Running of jobs must be done on the worker nodes via the cluster scheduling system *OpenPBS*. This system reserves the requested number of nodes for a specific duration (the default is 15 minutes). The user interface of this job manager is called *prun*, which provides a convenient way to start jobs.

For more information about the *DAS-2*, see *http://www.cs.vu.nl/das2*.

4 My research

Concise Survey

As I already mentioned in the introduction, I have got three aims: proving the aptitude of EC, verifying the foregoing results and finding such maps whose existence was only a conjecture so far.

At first I studied the one-dimensional equivalents of the tori, namely de Bruijn cycles. My aim was to observe all the features that can be useful in higher dimensions, and to use the obtained experience in course of the implementation of the higher-dimensional cases. By means of the reference algorithm (a backtrack search algorithm) I managed to devise a formula concerning the number of tokens in a cycle. This result allows of applying a representation (*permutation representation*) of the individuals in the EA that proved to be more efficient during the evolution than the straightforward one (*integer representation*) that first I had a whack at. Although it is not so relevant from the point of view of my research regarding EAs, but during my experiments I observed a mathematical relation concerning the number of de Bruijn cycles, that diverges from the known one (the number of Euler paths).

Next I started to experiment with two-dimensional tori. Unfortunately, in this case the experiments with the reference algorithm are often time-consuming and sometimes also impossible due to the size of the search space. Hence I couldn't gain sufficient data to devise a similar formula concerning the number of tokens as in case of one-dimensional cycles. Even so the results of these experiments show that there is a relationship, even if we are not able to devise it.

Hence in the two-dimensional case I was forced to use the straightforward integer representation. It worked, however, the evaluation of individuals was very costly, so that it set a limit to my experiments. This issue holds in the higher dimensional cases, as well. With the recent computational capacity it is not guaranteed that the evolution gives any result in reasonable time. I also tried to speed up the algorithm by means of parallelization. I parallelized both the reference and the genetic algorithm, but either the outcomes of the experiments were not satisfying or the matter of applicability is troublesome.

I concluded the following. I believe that the mathematical relationship regarding the number of tokens exists in higher dimensions, as well. Furthermore I believe that the evolution with a representation based on tokens could be the most efficient tool in finding perfect maps. Hence my aim in the following is to experiment with the genetic algorithm in order to get closer to the possibility of the permutation representation.

4.1 Specification of the Algorithms

4.1.1 One-dimensional Case

4.1.1.1 Reference Algorithm

A backtrack search is implemented in *DbcBackTrack.java*.

Functioning of the Algorithm

The program reads the parameters (the alphabet size and the span size) from the standard input and searches the space of all the possible candidates for de Bruijn Cycles.

The longest possible cycle that the program is able to produce, has the length of $2^{31}-1$ (the reason for this is the *integer* representation of the cycle length). The alphabet size and the span size are represented as a *byte* variable, which has a maximum value of 2^8-1. While reading the parameters from the standard input, the program gives a warning and the set of possible values if the length of the cycle would exceed the above threshold. When having the parameters, it gives the length and the number of such cycles (see section 4.2.6.2), and asks whether to find all the possible ones.

Its output (the cycles, the number of basic steps and backtracks, and the CPU time needed, respectively) is written to a file named *dbc_alphabet_span_bt.txt* where the strings “alphabet” and “span” denote the actual size of the parameters.

Specification of the Algorithm

In what follows *k* and *n* denote the size of the alphabet and the span, respectively.

Search space: The space of all the possible candidate cycles. Its size is k^{k^n} (note that k^n is the length of the cycles).

Basic step: Inserting an element of the alphabet into the cycle.

Each candidate is bound to contain the all zeros tuple, so we insert this tuple into the forepart of the candidate. This part of the candidate is fixed, there is no backtracking from this level (the levels of the search tree correspond to the positions in the candidate, so this means the $(n-1)^{\text{th}}$ level, because we have inserted *n* zeros into the candidate and the level numbering begins with zero). In other words this means that the search does not need to be executed beginning with the other possible tuples. The explanation for this heuristic is the periodic feature of the cycles, namely no matter from which position the cycle is inspected. Hence we can be sure that all the possible candidates will be found on the branch beginning with the all zeros span.

The Principle of the Functioning

To provide the proper functioning we have two arrays:

i) **tuplesInCandidate** It is a one-dimensional array, whose indices stand for the decimal values of the tuples and the elements indicate whether the corresponding tuple is used in the candidate (1 if it is used, 0 otherwise). So the tuples need not to be stored actually, there is a conversion function instead that converts a tuple into a decimal value if needed. This array guarantees that the candidate is a prefix of a de Bruijn cycle, hence the candidate needs not to be examined in every step whether it is a legal one.

ii) **triedAlready** On each level we keep a record of the elements, which we have already inspected a branch beginning with. These elements are stored in a two-dimensional array where the indices of the first dimension stand for the levels, and those of the second dimension stand for the element of the alphabet. Likewise in the case of the previously mentioned array *tuplesInCandidate*, the elements indicate whether the corresponding tuple was tried already.

In every step we choose an element from the alphabet (which was not tried yet in this level, namely it is not in the appropriate array of *triedAlready*), and try to insert it into the candidate. If the arising tuple is legal (it is not in the array *tuplesInCandidate*) then the insertion is accomplished actually and the appropriate element of *tuplesInCandidate* is set, otherwise we backtrack one level in the search tree and modify the content of *triedAlready* accordingly. The possible number of basic steps in a level equals to the alphabet size *k*, and if there is no more non-tried element, a backtracking is needed. This backtracking differs a little bit from the previously mentioned one, because it is made from a ramification of the search tree, so the elements of the array *triedAlready* concerning the actual level needs to be reset to provide the coming element the possibility of continuation. This situation is illustrated in the figure below.

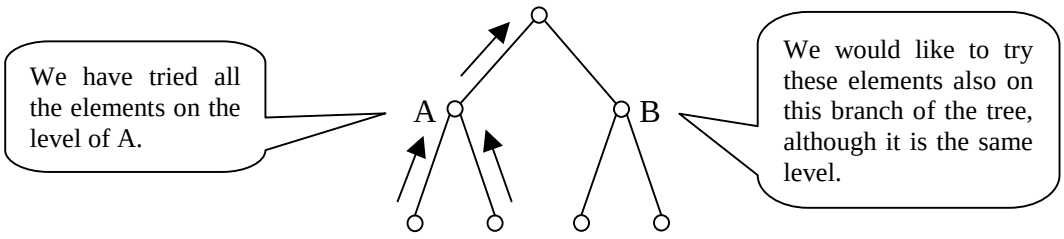


Figure 4-1 Backtracking from a ramification

Due to the two assistant arrays the insertion of the new element cannot corrupt the "perfection" and candidates on the lowest level (when the level is equal to the length of the cycle) are bound to be de Bruijn Cycles and to differ from the previously found ones.

Parallelizing the Algorithm

The parallel version of the algorithm is implemented by means of *Java RMI* (Remote Method Invocation) and *Java threads*. It is adjusted to the parallel testing environment, the *DAS-2* (see section 3.2). The program consists of two components outlined below.

i) The remote object is implemented in *DbcBackTrackRemoteObject.java*. Its task is to perform a search beginning with a particular node on a certain branch of the search tree, to that end it provides an interface with a public function called *doBackTrack()*.

ii) The main program is implemented in *DbcBackTrackRemote.java*. It divides the search tree among a given number of threads, namely every thread is provided with a node, which the search has to be performed beginning with.

The number of threads equals to the number of loaded remote objects, so there is a one-to-one correspondence between them. The task of the threads is to connect to the remote objects, invoke their *doBackTrack()* function, and return with the solution. The references to the remote objects can be retrieved by creating a file (*id*), which contains the names of the hosts they are running on. This can be done in the following way. When starting the remote objects, the output of the *prun* command has to be directed into the file:

```
>prun –v –1 ./run_java numproc DbcBackTrackRemoteObject 2> id
```

The –v flag is essential, it reports the host allocation. The –1 flag indicates that we want to run one process per node. The executable *run_java* is a special script, which sets the appropriate system properties to make running Java applications possible. The argument *numproc* stands for the number of processors.

The main program will read the information about the hosts from the *id* file, and will start a proper number of threads.

4.1.1.2 Genetic Algorithm

The first stage to build a genetic algorithm is to decide on a representation of a candidate solution to the problem. A straightforward idea is letting the phenotype and the genotype of an individual be the same, namely fixed-length combinations of the elements of the alphabet.

I made several experiments applying different operators and selection mechanisms, and the conclusion is that the algorithm based on a “tricky” representation works more efficiently (for detailed comparison see section 4.2.3.1). This is a permutation representation based on *tokens* (see section 4.2.6.1), and the components of this algorithm are outlined below.

The algorithm itself is implemented in *DbcGA.java* and the different components are implemented in separate classes (*Initialization.java*, *ParentSelection.java*, *Mutation.java*, *Recombination.java*, *Evaluation.java* and *SurvivorSelection.java*). These components provide an interface with some functions that realize various operators and mechanisms.

4.1.1.2.1 Representation

The phenotype space and the genotype space are different. Phenotypes are the possible solutions within the original problem context. Genotypes are permutations of references to different tokens. Given the alphabet and the span size, the number of tokens is particular, and each chromosome has to contain all the possible tokens. The chromosomes consist of unique elements, because even if two tokens are equal, the references to them are different. The mapping between the genotype and the phenotype is illustrated in the figure below.

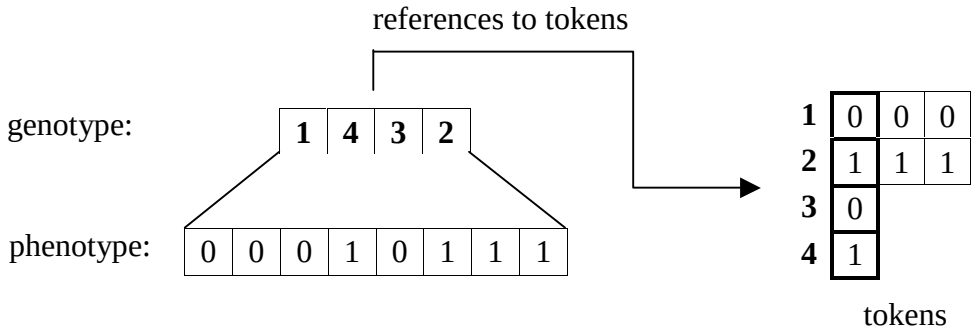


Figure 4-2 Representation of an individual

Applying this representation the search space will be all the possible permutations of the tokens. The size of this space – considering each one of the tokens as unique – is $N(k,n)!$, where $N(k,n)$ denotes the number of tokens given the alphabet size (k) and the span size (n).

4.1.1.2.2 Initialization and Termination Condition

Initialization: The population has a fixed size, and first it is filled with random permutations of the possible elements (the references to tokens).

Termination condition: The evolution is terminated if the program has found an appropriate cycle. This search may take quite much time, even in case of small parameters. Nonetheless, I didn't define any other termination conditions, because our aim is to find a cycle and it's up to the user to decide when to stop the search.

Note that our problem is a global optimization problem, where a "good" (near perfect, at least by reason of its fitness) but suboptimal solution cannot be satisfactory. In other words the *anytime behaviour* [52] is not granted in our case.

4.1.1.2.3 Evaluation Function

The evaluation function assigns a quality measure to genotypes. The aim is to minimize this function, it minimum value is zero. An individual with minimum fitness value is bound to be a de Bruijn cycle. This function has two components:

i) *In the phenotype space*:

At each position we inspect the chromosome whether the tuple beginning at that position is unique, so every position has an own part-fitness value. When considering a tuple, all the positions need to be examined before its beginning position. If it is unique then the part-fitness will be zero, otherwise it will stand for the rank of the tuple, namely how many times it occurred before (see the figure below). The actual fitness can be gained by summing up these part-fitness values. The zero value of this fitness indicates that all the tuples are unique, namely we have found what we were searching for.

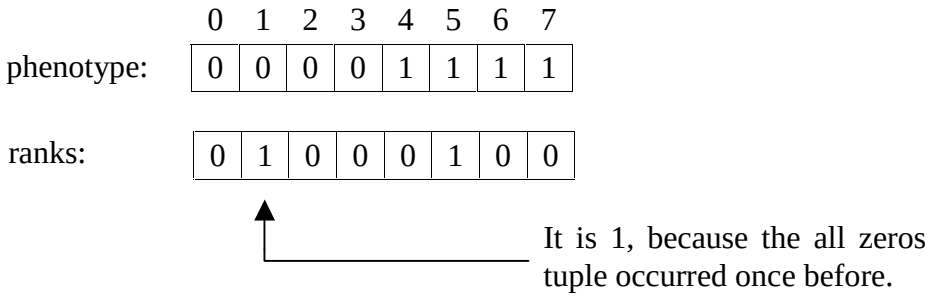


Figure 4-3 Ranks of the positions in a $(8;3)_2$ - de Bruijn Cycle

ii) *In the genotype space*:

If two tokens get next to each other, it will be a legal arrangement only if their elements are different. The reason for this is that the longest token is span-sized long, and if it gets next to any of the tokens having the same elements, then the span-sized tuple will be occur twice, hence corrupting perfection.

In a chromosome there are $N(k,n)$ fitting points, where tokens can get next to each other. We observe the number of legal connections by means of a variable: if two adjacent tokens are different, then it is increased by one. If the value of this variable equals to $N(k,n)$, then the chromosome has a legal permutation of the elements. This measure is realized by adding the difference of the number of tokens and $N(k,n)$ to the fitness value.

4.1.1.2.4 Variation Operators

The variation and selection operators I applied are commonly used for permutation representations. I based my implementations on the book of Eiben and Smith [52]. Here I give a short description of the certain operators.

4.1.1.2.4.1 Recombination

I applied an *order crossover*. It is a binary operator, namely it merges information from two parents into the offspring. In my implementations two parents always breed two children. The number of descendants is determined by the *generational gap*, which indicates the percentage

of the population that is replaced by the new individuals. This parameter is given by the user, likewise the population size, and the number of offspring is determined by their product.

The individuals are selected from the mating pool in pairs at random. It is up to the *crossover rate* whether the two candidate parents will breed a child or not. This parameter is given also by the user, and the outcome of a random drawing – compared to this rate – will be decisive: if the random value is smaller than the rate the candidates will mate, otherwise the children will be created asexually, namely the parents will be simply copied into the offspring.

The principles of the *ordered crossover*:

First, we have to choose two crossover points at random, and then copy the genes between them into the corresponding child. The remaining genes are copied into the other child according to the following three rules:

i) the copying starts from the second crossover point and wraps around at the end
ii) only the unused genes are copied
iii) the original order of the genes is preserved

4.1.1.2.4.2 Mutation

I applied two mutation operators: *swap mutation* and *inversion mutation*. The mutation is a unary operator, which takes an individual as input and alters it according to the *mutation rate*. The mutation rate is a parameter given by the user. Each individual has a probability to be mutated. If this value is smaller than the mutation rate then the individual is left unchanged, otherwise it is altered according to the semantics of the actual operator:

swap mutation: It randomly picks two genes in the individual and swaps them.

inversion mutation: It randomly selects two positions in the individual and reverses the order of the genes between those positions.

4.1.1.2.5 Selection Operators

4.1.1.2.5.1 Parent Selection

I applied three kinds of parent selection methods: *ranking selection*, *fitness proportional selection* and *tournament selection*. All the three of them apply to the population as a whole and return the mating pool. The mating pool contains the individuals that are good enough – based on their fitness – to become parents. In my implementations the mating pool has the same size as the population (μ).

The first two methods can be divided into two consecutive steps: at first they define a probability distribution that indicates the likelihood of each individual being selected for reproduction, then a selection method is applied to sample the parents from this distribution. I applied two sampling methods: *roulette wheel algorithm* and *stochastic universal sampling algorithm*. The former corresponds to spinning a one-armed roulette wheel μ times, where the sizes of the holes reflect the selection probabilities, while the latter is equivalent to making one spin of a wheel with μ equally spaced arms. I based my implementations on the corresponding pseudo codes in [52].

The principles of determining the probability distributions:

ranking selection: First, the population is ranked based on the fitness values in such a way that the worst individual has rank 0, while the best has rank μ. Next, I applied a linear mapping to assign selection probabilities to the individuals based on the following formula [52]:

$$P(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)},$$

where i stands for the actual rank and s is parameter ($1.0 < s \leq 2.0$) given by the user.

fitness proportional selection: The individuals are selected according to their fitness values, namely the probability that an individual is selected for mating is $f_i / \sum_{j=1}^{\mu} f_j$.

tournament selection: I applied two kinds of tournament selection methods: *deterministic* and *stochastic*. In both cases k individuals are selected randomly, where k stands for the tournament size. In the deterministic case always the best individual survives, otherwise there is a probability indicating the likelihood that the fittest member is selected. Both this probability and the tournament size are parameters given by the user.

In case of the stochastic version I applied a roulette wheel algorithm to select the winner. I determined the selection probabilities in the following way. The likelihood that the individual with the best fitness will be selected is p. I distributed the remaining likelihood $p - 1$ among the other $k - 1$ contestants based on their fitness.

Note that this selection can be performed with or without replacement. In my implementations the replacement doesn't make sense in case of parent selection, because the size of the population and the mating pool are equal. I applied this option only in case of survivor selection.

4.1.1.2.5.2 Survivor Selection

All the selection operators mentioned above are possible replacement schemes in case of survivor selection, as well. They can be applied with a tiny difference, namely they take the union of the population and the offspring ($\mu + \lambda$) as input and return the survivors (μ), but the mechanism of the selection is the same.

I applied two selection methods that deviate from the previous ones, because they are not probabilistic but deterministic:

best from union: The best λ members are selected to be survivors. They are selected from the union of the population and the offspring ($\mu + \lambda$).

replace worst: The worst λ members of the population are replaced by the offspring.

4.1.1.2.6 Parallelizing the Algorithm

The *Island Model* (see section 3.1.1) serves as the basis of the parallel version of the algorithm. It is implemented by means of *Java RMI* (Remote Method Invocation) and *Java threads*. It is adjusted to the parallel testing environment, the *DAS-2* (see section 3.2). The program consists of three components outlined below.

i) The Remote Object

The remote object is implemented in *DbcGARemoteObject.java*. Its task is to evolve a population, a separate "island", and it also supports the migration of the individuals. To that end it provides an interface with six public functions described below.

The function *startGA(byte alphabet, byte tupleSize, int populationSize, int epoch, int numberOfMigrants)* creates and evolves a population with the given parameters. The parameter *epoch* stands for the number of generations after individuals are exchanged. The migration needs to be synchronized, namely the exchange of individuals have to be an atomic operation.

This atomicity is realized as follows. When the migration is in due time - that is the required number of generations has evolved -, the evolution of the population is suspended until all the migration mechanisms (sending and receiving individuals) accomplishes. From the aspect of the remote object the migration consists of four consecutive steps:

i) First, it indicates that it is ready to accept requests for the selection and sending of migrants. It is realized by setting the value of the private variable *waitingForSendMigrantsThread* to true. The interface provides read access to this variable through the public function *isWaitingForSendMigrantsThread()*.

ii) It prepares the migrants by marshalling the selected individuals and their fitness values into a "package", which is implemented as a vector of length two, the first element reserved for the individuals, the second for their fitness values. The number of the individuals is determined by the parameter *numberOfMigrants*, and the selection mechanism is based on fitness, namely the ones with best fitness are selected for migration. It is important to remark that the individuals are not effectively moved to the other population, they are merely copied. If the marshalling is ready, the object notifies the thread *SendMigrantsThread* already waiting for the migrants.

iii) Then it indicates that it is ready to accept requests for the reception of migrants. It is realized by setting the value of the private variable *waitingForReceiveMigrantsThread* to true. The interface provides read access to this variable through the public function *isWaitingForReceiveMigrantsThread()*.

iv) The replacement of the individuals is settled by the thread *ReceiveMigrantsThread*, and the remote object has to wait while it accomplishes. In the course of replacement first the individuals with worst fitness are wiped out from the population, then the migrants are unmarshalled and inserted into it. It is important to remark that the

references of the migrants need to be readjusted to the local ones. If the replacement is ready, the thread notifies the object that the evolution of the population may continue.

ii) The Migration Manager

The migration of the individuals is implemented in *MigrationManager.java*. It creates the conditions of migration by providing every population with two kinds of threads, a *SendMigrantsThread*, and a *ReceiveMigrantsThread*. The contact point between these threads and the populations is realized by the *sendMigrants(int numberOfMigrants)* and the *receiveMigrants(Vector migrants)* function of the remote object, respectively. These functions perform the actual exchange of individuals and can be invoked by the threads.

It is important to note that the communication structure is a ring, namely the individuals are migrating between the neighbouring populations as illustrated in the figure below.

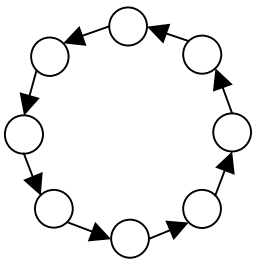


Figure 4-4 The migration between populations

As already mentioned with reference to the remote object, the migration needs to be synchronized. This synchronization was made clear on the level of individual population in the previous section. Now we inspect a higher level, where we take all the population into consideration.

The main concern is that the reception of migrants from a neighbouring population requires these migrants to be already prepared. Hence we have to apply some kind of scheduling, and it works as follows. First, we demand every population to prepare their emigrants. Transitionally, until all the populations are ready, they are stored in an array called *ellisIsland*[1]. Then, the elements of this array are delivered to the proper population. This scheduling – keeping the populations wait for each other – does not have a detrimental impact on the performance, because the populations are evolving with the same parameters, hence the time needed to produce a new generation is the same for every island.

iii) The Main Program

The main program is implemented in *DbcGARemote.java*. Its task is to start the threads that evolve the separate populations on different remote objects, and the migration manager, respectively. The references of the remote objects can be retrieved in the same way as in the case of the backtrack search algorithm (see section 4.1.1.1).

[1] Inspired by New York immigrants' quarantine in Ellis Island in the early 20th century.

4.1.2 Two-dimensional Case

4.1.2.1 Reference Algorithms

In two dimensions we distinguish three different kinds of cases regarding the periodicity of the map. The aperiodic case is implemented in *ApmBackTrack.java*, the semi-periodic case in *SpmBackTrack.java* and the periodic case in *PpmBackTrack.java*. All the three of these algorithms are backtrack search algorithms.

Functioning of the Algorithms

These algorithms are embedded in a compound software, the *Perfect Map Generator*, which provides a graphical user interface to control the algorithms. For further information about the functioning see the user documentation of the former software (Appendix A).

Specification of the Algorithms

In what follows k denotes the alphabet, (m,n) the dimension of the window and (R,S) the dimension of the map.

Search space: The space of all the possible candidate maps. Its size is $k^{k^{mn}}$ (note that k^{mn} is the area of the map, namely the number of its elements).

Basic step: Inserting an element of the alphabet into the map.

The principle of the functioning is the same as in the one-dimensional case. The candidate is a matrix this time, and the levels of the search tree correspond to the positions in the candidate, which are pairs in the form of (row, column). The conversion between the levels and these pairs is trivial (the quotient of the level and n yields the corresponding row, while the remainder yields the corresponding column).

In every step we choose a non-tried element from the alphabet, and try to fit it into the candidate. This insertion is carried out in row-major order, so first the rows of the matrix are filled up. Before we accomplish the insertion actually, the arising window should be inspected whether it is a legal one. This window is meant to have the newly inserted element in its right bottom corner (see Figure 4-5). There are cases when it is not possible to create such a window (see level 0 – 4, 8 and 12, respectively in Figure 4-5), they are treated as legal.

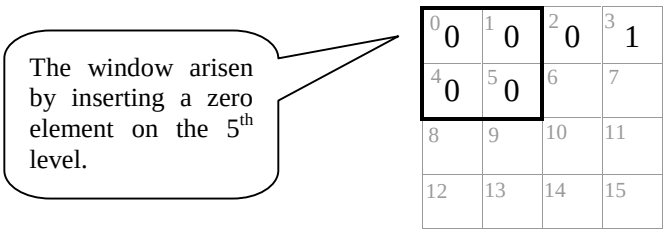


Figure 4-5 Creating an $(4,4;2,2)_2$ - Periodic Perfect Map

The only issue that needs to be treated differently in case of the three algorithms is the matter of periodicity. The two concerned case are:

i) *Semi-periodic case*: By inserting the last element into a row, not only one window is arising, but *n*. If one of them is illegal, we backtrack one level in the search tree, namely let the copy undone.

ii) *Periodic case*: Not only the last elements of the rows are concerned, but the last elements of the columns, as well. By inserting the last element into a column then *m* new windows are arising that needs to be checked.

If we are about to find all the possible perfect maps, every newly found map needs to be checked whether it differs from the previously found ones. This checking should consider the possible periodicity of the map, namely the maps arising by shifting the original map should not be considered as different.

4.1.2.2 Genetic Algorithms

4.1.2.2.1 Representation

In the one-dimensional case I applied a permutation representation based on *tokens* (see section 4.2.6.1). I examined several perfect maps, but in the two-dimensional case I didn't find an analogous relation neither in the phenotype space nor in the genotype space.

Representation: I applied an integer representation where the values are restricted to a finite set, namely the alphabet. Note that is analogous to the one-dimensional straightforward representation, where the phenotype and the genotype are the same. In this case they are not the same, however, because the two-dimensional phenotype needs to be mapped to one dimension where we can apply the existing operators; but the essence is the same, namely the genotype contains the elements of the matrix directly.

Mapping: The mapping between the phenotype space and the genotype space is illustrated in the figure below. The elements of the phenotype are stored in row-major order in the genotype. The inverse mapping (determining the position in the phenotype given the position in the genotype) is evident, as well: the quotient of the position and *n* yields the corresponding row, while the remainder yields the corresponding column.

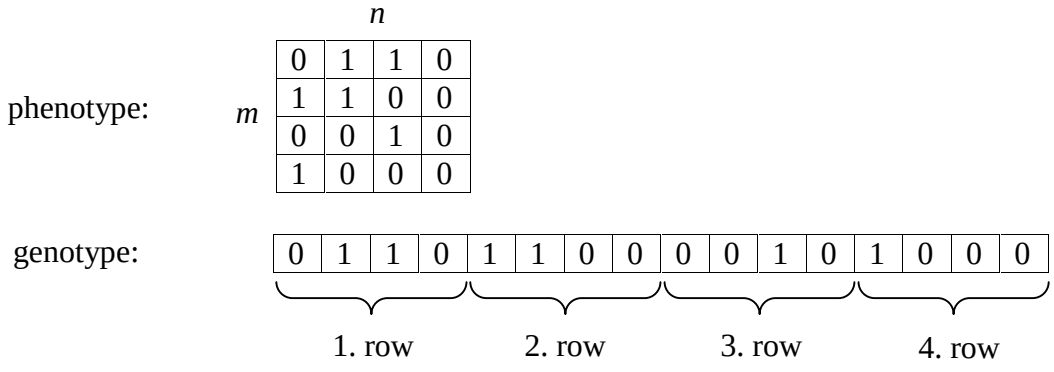


Figure 4-6 Representation of an individual

4.1.2.2.2 Initialization and Termination Condition

Initialization: The population has a fixed size, and first it is filled with random individuals (arrays of length $m \cdot n$).

Termination condition: The considerations with regard to the termination condition are the same as in the one-dimensional case (see section 4.1.1.2.2).

4.1.2.2.3 Evaluation Function

The evaluation function assigns a quality measure to genotypes. The aim is to minimize this function, it minimum value is zero. An individual with minimum fitness value is bound to be a Perfect Map. The evaluation is performed in the phenotype space and the idea is the same as in the one-dimensional case (see section 4.1.1.2.3) with the necessary modifications due to the higher dimension, namely at each position we inspect the matrix whether the window at that position is unique. The semantics of the evaluation is the same, it is not detailed here.

4.1.2.2.4 Variation Operators

I implemented more realizations of the certain operators, and I observed and compared the outcomes of the genetic algorithm applying different operators in course of the evolution. The results concerning these experiments can be found in section 4.2.3.3.

The variation and selection operators I applied are commonly used for integer representations, and I based my implementations on the book of Eiben and Smith [52]. Here I give a short description of the certain operators.

4.1.2.2.4.1 Recombination

I applied two kinds of recombination operators: *uniform crossover* and *n-point crossover*. The semantics of the recombination is the same as in the one-dimensional case (see section 4.1.1.2.4.1).

The principles of the two crossover operators:

uniform crossover: Each gene of the individual is treated independently, and a random variable will decide from which parent to inherit the certain genes.

n-point crossover: The parents are divided into sections by *n* crossover points. The sections will be copied into the children alternately. The number of crossover points is a parameter given by the user.

4.1.2.2.4.2 Mutation

I applied two kinds of mutation operators: *random resetting* and *creep mutation*. The semantics of the mutation is the same as in the one-dimensional case (see section 4.1.1.2.4.2).

The principles of the two mutation operators:

random resetting: A new value is chosen at random from the set of permissible values, namely from the alphabet.

creep mutation: A small value is added to the gene. This value has an upper bound if it is positive or a lower bound otherwise, because the sum may not to exceed the set of permissible values. Both the positive and the negative values have equal chances, this choice is implemented by means of a random variable.

4.1.2.2.5 Selection Operators

I used the same operators as in the one-dimensional case (see section 4.1.1.2.5). Note that the selection operators can be applied independently from the representation, because they take only the fitness information into account.

4.1.3 Higher Dimensions

4.1.3.1 Practical Considerations

What does a three-dimensional torus look like?

The following figure illustrates a three-dimensional torus that consists of concentric two-dimensional tori actually. The interpretation of the dimensions of this torus is the following: the first dimension stands for the number of embedded two-dimensional tori (“pipes”), the second one denotes the first dimension of the embedded tori (“circumference of the pipes”) and the third one means the second dimension of the embedded tori (“length of the pipes”), respectively.

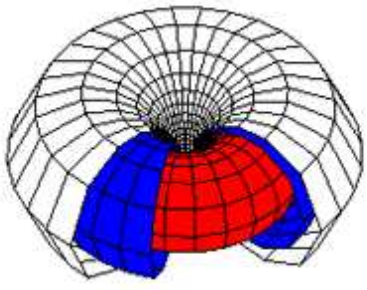


Figure 4-7 A three-dimensional torus

Theoretical results

The only theoretical result regarding the existence of a higher dimensional torus is the generalization of Cock’s technique (see Theorem 4 in section 2.3.1). The set of solutions that this theorem provides is very small, furthermore the parameters of such a torus are very special, namely each new dimension has size exponential in the previous.

With small parameters it gives reliable results. If $\bar{n}$ = <2,2,2>, k = 2 and d = 3, then the solution is $\bar{r}$ = <4,4,16>. It is the smallest possible torus in three dimensions.

Practical results

In [20] Horváth and Iványi constructed the smallest possible (a $256\times256\times256$ sized *8*-ary) 3-Cube.

Finding and verifying a higher-dimensional torus meets with obstacles on account of the limited storage capacity and the finite CPU speed.

Storage requirements of the d-Cubes:

$(256,256,256;2,2,2)_8^3$	16 MB
$(512,512,512;3,3,3)_2^3$	128 MB
$(16,16,16,16;2,2,2,2)_2^4$	64 kB
$(81,81,81,81;2,2,2,2)_3^4$	41 MB

Storage requirements of the smallest possible d-Tori:

$(2,2,64;2,2,2)_2^3$	256 bytes
$(16,16,16,16;2,2,2,2)_2^4$	64 kB
$(64,64,64,64,256;2,2,2,2,2)_2^5$	4 GB

Storage requirements of the backtrack search algorithm:

In this paragraph I try to estimate the minimum storage demand of the backtrack search algorithm. There are two objects that are essential and serve as the basis of the algorithm: the array that contains the elements tried already in a level and the candidate solution itself, respectively. I will give a lower bound by reason of these objects.

These objects will have the following form in three dimensions (r_i, $i=1\ldots n$ stand for the dimensions of the map):

```
byte[][] triedAlready = new byte[r1 * r2 * r3][alphabet];
byte[][] candidate = new byte[r1][r2][r3];
```

Storage demands (lower bound):

$(256,256,256;2,2,2)_8^3$	triedAlready: $256^3\cdot8$ bytes = 128 MB candidate: 256^3 bytes = 16 MB sum total: 144 MB
$(512,512,512;3,3,3)_2^3$	triedAlready: $512^3\cdot2$ bytes = 256 MB candidate: 512^3 bytes = 128 MB sum total: 384 MB

$(16,16,16,16;2,2,2,2)_2^4$	triedAlready: $16^4 \cdot 2$ bytes = 128 kB candidate: 16^4 bytes = 64 kB sum total: 192 kB
$(81,81,81,81;2,2,2,2)_3^4$	triedAlready: $81^4 \cdot 3$ bytes ≈ 123 MB candidate: 81^4 bytes ≈ 41 MB sum total: 164 MB

CPU time requirements of the backtrack search algorithm

Let us consider an $(\overline{R};\overline{n})_k^d$ - torus *where* $\overline{R} = (r_1,...,r_d)$ *and* $\overline{n} = (n_1,...,n_d)$. The size of the search space is $k^{\prod r_i}$, namely the search tree has $k^{\prod r_i}$ leaves:

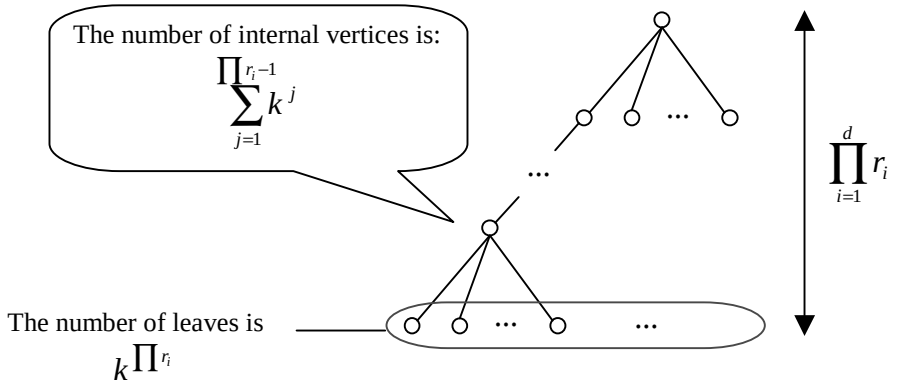


Figure 4-8 Sketch of the search tree

Let *x* denote the number of such tori (it is not known). The time complexity of the algorithm (when finding a single map):

worst case:	(if we are examining all the wrong cases before finding a good solution)	$\Omega\left(k^{\prod r_i} - x\right)$
best case:	(if the solution is on the first branch of the tree)	O(1)
average case:	(if the solutions are distributed uniformly among the leaves)	$\Omega\left(\left\lfloor \frac{k^{\prod r_i}}{x} \right\rfloor\right)$

The most meaningful measure is the number of basic steps, but it is difficult to determine the exact number of basic steps in the certain cases. A basic step is defined as inserting a new element into the candidate (see section 4.1.2.1), namely stepping to the next vertex in the search tree.

Let us consider a $(2,16;2,2)_2$ - Semi-Periodic Perfect Map. In this case the number of vertices of the search tree is ~8590 million (the root and the leaves included), and the algorithm made

164 thousand basic steps to find a map. One basic step takes 0,05 ms on a 3000 MHz CPU. Although it is not so important in terms of our aim, but interesting to mention that the algorithm made ~135 million basic steps to find all the possible maps, that is it traversed about 1,57% of the search tree.

Conclusions

On the strength of the storage and CPU requirements it is clear that the backtrack search has no chance to cope with these large search spaces. I implemented the three and four-dimensional genetic algorithms, and during the tests I applied the parameter sets and operators that turned out to be the best working in the two-dimensional case.

4.1.3.2 Three-dimensional Genetic Algorithm

The three-dimensional genetic algorithm is equivalent to the two-dimensional one, only the issue of representation differs.

The phenotype of an individual (a) is a three-dimensional solid of size $r_1 \times r_2 \times r_3$. The genotype (b) is a vector of length $r_1 \cdot r_2 \cdot r_3$. The mapping between them is the following. The chromosome has r_3 segments, which correspond to the slices of the solid (indicated by grey in Figure 4-9). That is to say every segment represents a matrix. The structure of such a segment equals to the chromosome in the two-dimensional case, namely it contains the elements of a matrix in row-major order.

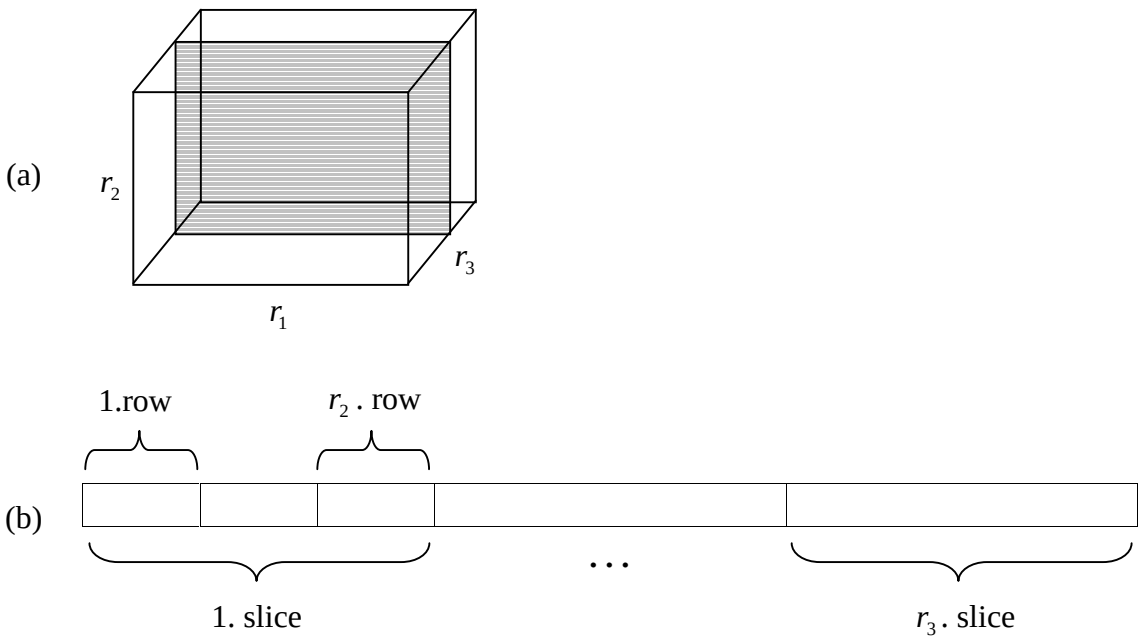


Figure 4-9 Representation of an individual

4.1.3.3 Four-dimensional Genetic Algorithm

The four-dimensional genetic algorithm is equivalent to the three-dimensional one, only the issue of representation differs. This case is a little bit intricate, because we have to imagine a four-dimensional hypercube. Any additional dimensions actually mean where to find the previous one. For example in three dimensions this means that there are r_3 slices and the third dimension indicates which slice to choose. In four dimensions there are r_4 three-dimensional solids and the fourth dimension indicates which solid to choose. The simplest way to imagine such a case is to consider the fourth dimension as series of discrete time intervals.

The phenotype of an individual (a) is series of r_4 discrete time intervals. The genotype (b) is a vector of length $r_1 \cdot r_2 \cdot r_3 \cdot r_4$. The mapping between them is the following. The chromosome has r_4 segments, which correspond to the time intervals. That is to say every segment represents a snapshot, a separate three-dimensional solid. The structure of such a segment equals to the chromosome in the three-dimensional case (see section 4.1.3.2).

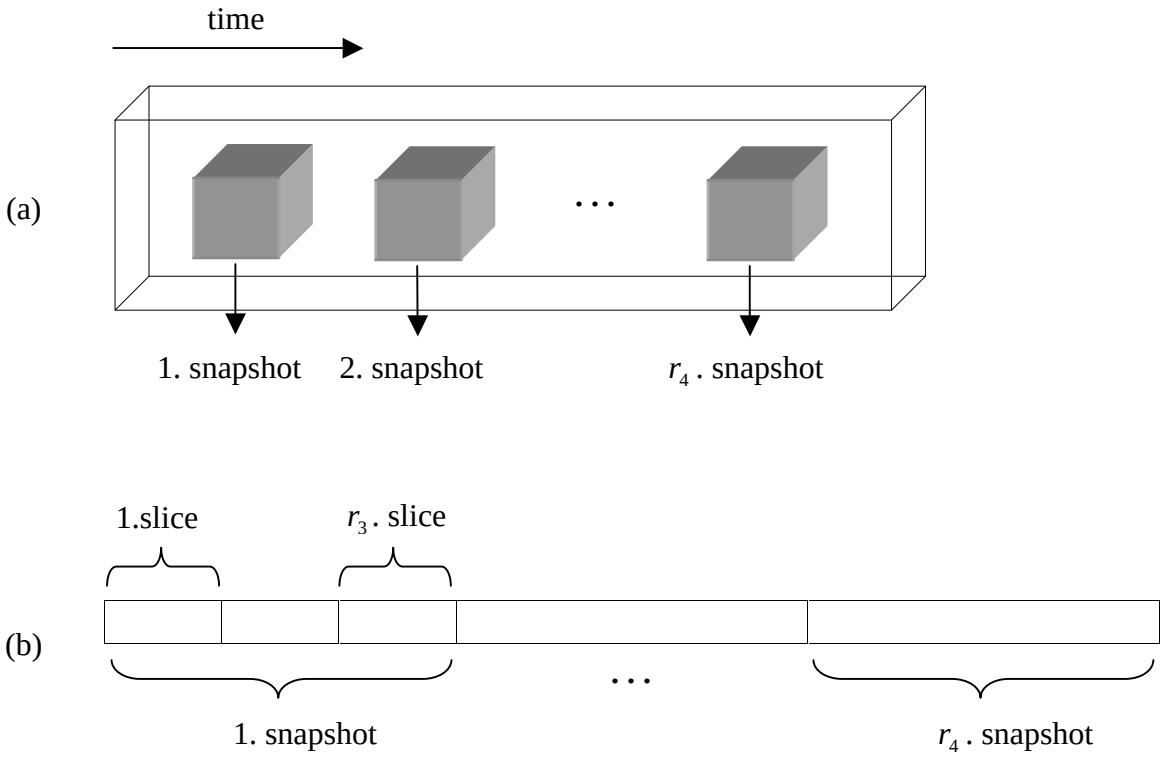


Figure 4-10 Representation of an individual

4.2 Experiments and Results

All the Excel files containing the detailed outcomes of my experiments can be found at the web address http://juditk.web.elte.hu/msc/.

I made my experiments considering the following three main issues:

i) Parameter tuning of the genetic algorithms.
ii) Comparison of the algorithms (the sequential and the parallel, and the reference- and the genetic algorithm, respectively).
iii) Searching different maps with the algorithm that proved to be the best during the experiments.

4.2.1 The performance measures

Both in case of the backtrack search and the genetic algorithm I measured the CPU time needed (in milliseconds) to find a solution. Apart from this I applied the following measures:

backtrack search algorithm: number of backtracks and basic steps (see section 4.1.1.1)
genetic algorithm: average number of evaluations to a solution (*AES*)

Although I didn't define any other termination condition besides finding the optimal map (see reasoning in section 4.1.1.2.2), yet in a few cases – to get to know the progress of the genetic algorithm – I applied a condition, namely the number of fitness evaluations. In these cases I studied the *success rate (SR)* (the percentage of runs terminating with success) and the effectiveness by means of the *mean best fitness (MBF)* (the average of the best fitness values over all runs).

Because of the stochastic nature of EAs, these performance measures are statistical, and a number of experiments need to be performed to gain sufficient experimental data. I conducted all of my experiments 100 times, hence all the values regarding the results in this thesis mean the average of 100 experiments.

4.2.2 Experiments with the Reference Algorithms

Comparison of the sequential and the parallel backtrack search algorithm

There are two cases that need to be treated differently when comparing the sequential and the parallel algorithm: whether we are about to find all the maps (a) or not (b). In both cases suppose that the solutions are distributed on the branches of the search tree uniformly.

(a) In this case the parallelization does not mean significant speed-up, since one thread has the same likelihood of finding a solution as more.

My test results shore up the above train of thoughts, as well:

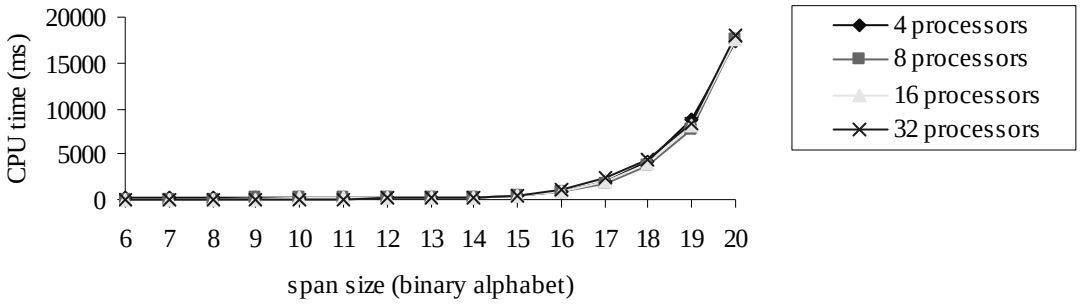


Figure 4-11 Speed-up (finding a single cycle)

(b) In this case the maximum speed-up is defined by the number of threads (t). The whole search tree needs to be checked, but this job is divided into equally sized parts: each thread has to examine the $1/t$ part of the search space, which means utmost $t\times$ speed-up.

There are two parameter sets that I could experiment with, because the number of possible cycles sets limits to the search:

parameter set	number of such cycles
$(16;4)_2$	16
$(32,5)_2$	2048

The speed-up is similar in both cases (the graph has the same shape), here only the results regarding the tests of the first set are published. In case of 32 processors there is a $9\times$ speedup instead of the expected 32. This difference can be attributed to the communication overhead and the control of the threads.

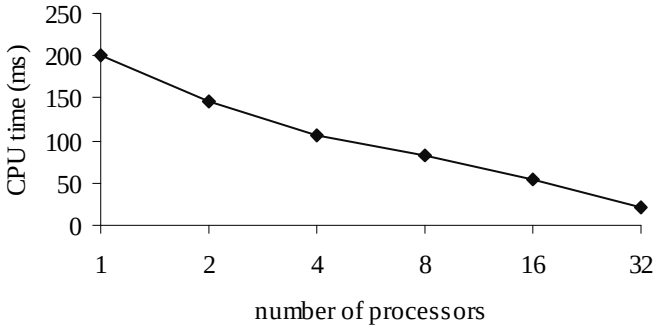


Figure 4-12 Speed-up (finding all the cycles)

4.2.3 Experiments with the Genetic Algorithms

4.2.3.1 Comparison of the Permutation and Integer Representations

4.2.3.1.1 Algorithm Setups

I compared the performance of both the permutation and the integer representation on two parameter sets. I search for a $(2^4;4)_2$ - dBC and a $(2^5;5)_2$ - dBC, respectively. Note that in case of the *permutation representation* the length of the chromosome does not equal to the length of the cycle (k^n), but to the number of tokens ($k^n / 2$). The detailed setup of the GAs is shown in the tables below.

Representation	permutation
GA model	steady-state
Chromosome length (L)	$k^n / 2$
Population size	$L / 2$
Recombination	ordered crossover ($p_c = 1.0$)
Mutation	swap mutation ($p_m = 0.5$)
Selection	ranking (s = 2.0, roulette wheel)
Replacement	best from union

Table 1 GA setup (permutation representation)

Representation	integer
GA model	steady-state
Chromosome length (L)	k^n
Population size	$L / 4$
Recombination	uniform crossover ($p_c = 1.0$)
Mutation	random resetting ($p_m = 0.25$)
Selection	ranking (s = 2.0, roulette wheel)
Replacement	best from union

Table 2 GA setup (integer representation)

4.2.3.1.2 Test Results

The AES and SR values are almost similar in case of the 16-length cycle, in both cases the integer representation is superior with a subtle difference. This behaviour changes significantly in case of longer cycles. The similarity of the outcomes in case of short cycles is due to the fact, that the prerequisites of the permutation representation are much more demanding, hence in case of small cycles the integer representation is able overcome it through probability reasons – but in larger cases this issue does not matter.

The MBF values are more optimal (lower) in case of the integer representation, which shows that the integer representation reaches near-optimal values quicker, but from there it makes further progress very slowly.

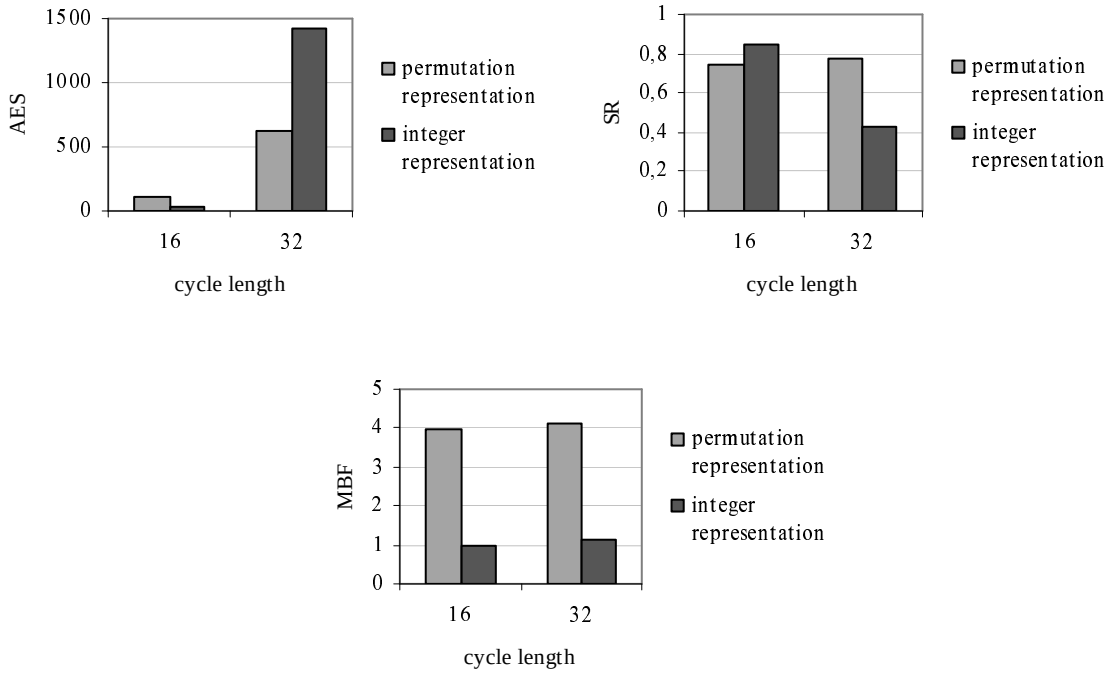


Figure 4-13 AES, SR and MBF of the two representations

4.2.3.2 Comparison of the Sequential and the Parallel GAs

4.2.3.2.1 Algorithm Setups

I performed my experiments on the *DAS-2* (see section 3.2) with 1, 2, 4, 8 and 16 processors. Due to the scheduling system and the workload of the *DAS-2*, I base my conclusions on the outcomes of 5 independent runs.

I applied the following parameter set. I searched for a $(32;5)_2$ - de Bruijn Cycle, and evolved 1, 2, 4, 8 and 16 populations in parallel (depending on the number of processors) with two migrants in every epoch. I examined three cases according to the length of an epoch (2, 4 and 8 generations). The certain islands evolved their populations with the same parameters (see the table below).

GA model	steady-state
Chromosome length	$2^5 / 2$
Population size	16
Representation	permutation
Recombination	ordered crossover (p_c = 1.0)
Mutation	swap mutation (p_m = 0.5)
Selection	-
Replacement	best from union

Table 3 Island model GA setup

4.2.3.2.2 Test Results

The below figure shows the progress due to migration. The best results were produced by the PGA (parallel GA), where the migrants were exchanged in every 4th generation. It is also clear to see that the progress is the most significant in case of 2 islands, while the any additional island cause to the performance only a subtle increase.

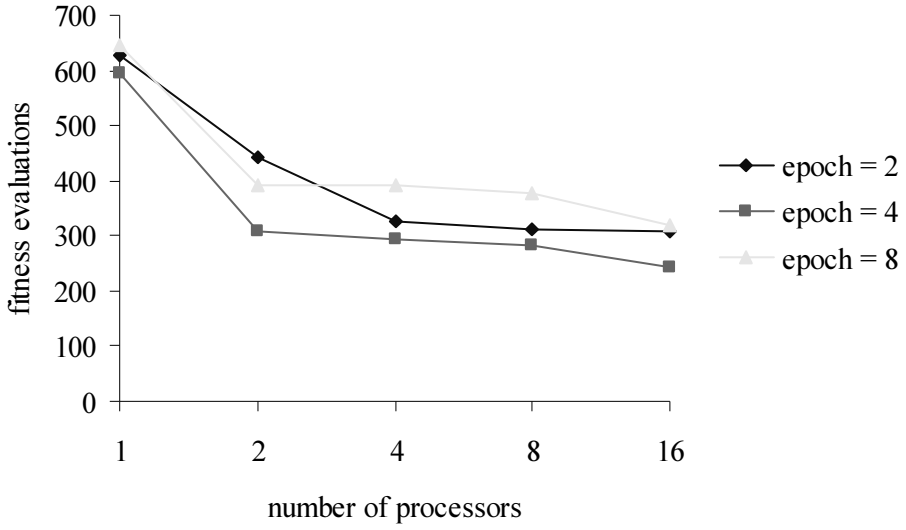


Figure 4-14 The progress due to migration

Although my conjecture was that the scheduling of migration won't have a detrimental impact on the performance (see the in reasoning section 4.1.1.2.6), the practice disproved this assumption as illustrated in the figure below.

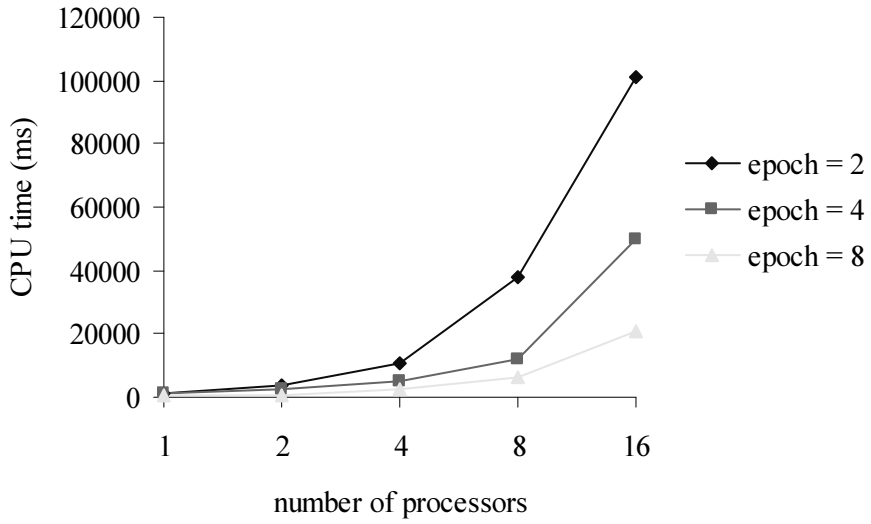


Figure 4-15 The impact of migrating on the performance

The conclusion is that the island model GA could be a useful tool in finding perfect maps. Evolving two or four populations in parallel can be very efficient, because the exchange of individuals with good fitness (in our case: low enough) and with different genotype, facilitates exploration. Furthermore in this case of two or four populations the impact of migrating on the performance is still reasonable.

In spite of the fact that the outcomes of my experiments are satisfying, there are practical limits of the applicability of this method. The *DAS-2* is dedicated to experiments, not for production work, hence it is allowed to run a program only for 15 minutes. The most challenging aim of this thesis – namely find a map, whose existence was only a conjecture so far – demands much more time than provided by this system.

4.2.3.3 Parameter Tuning of the One-dimensional GA

4.2.3.3.1 Algorithm Setups

My experiments concerning the parameter tuning are based on the AES measure. I examined the performance of the certain operators in case of a $(2^5;5)_2$ - dBC. The parameter setup of the different experiments differs only with respect to the actual operator, otherwise it is the based on the table below.

GA model	generational (gap = 0.8)
Chromosome length (L)	2^5 / 2
Population size	16
Representation	permutation
Recombination	ordered crossover (p_c = 1.0)
Mutation	swap mutation (p_m = 0.5)
Selection	ranking (s = 2.0, roulette wheel)
Replacement	best from union

Table 4 GA default setup (for parameter tuning)

4.2.3.3.2 Test Results

Parent selection

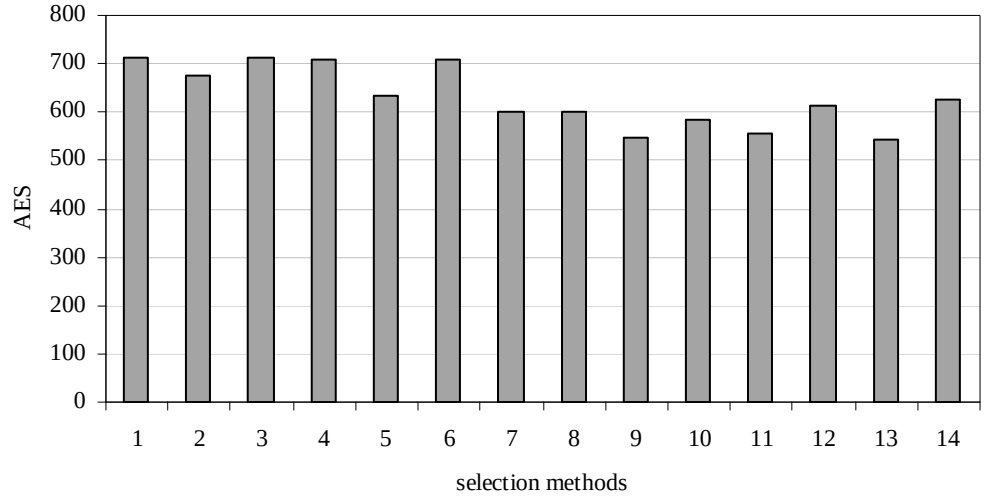


Figure 4-16 Comparing the parent selection methods

The meaning of the certain columns:

1) ranking selection (s = 1.5, roulette wheel)
2) ranking selection (s = 2.0, roulette wheel)
3) ranking selection (s = 1.5, stochastic universal sampling)
4) ranking selection (s = 2.0, stochastic universal sampling)
5) fitness proportional (roulette wheel)
6) fitness proportional (stochastic universal sampling)
7) deterministic tournament (k = 2)
8) deterministic tournament (k = 4)
9) deterministic tournament (k = 8)
10) stochastic tournament (k = 2, p = 0.5)
11) stochastic tournament (k = 2, p = 0.6)
12) stochastic tournament (k = 2, p = 0.7)
13) stochastic tournament (k = 2, p = 0.8)
14) stochastic tournament (k = 2, p = 0.9)

There is no significant difference between the ranking and the fitness proportional selection, and it is also difficult to say which selection algorithm (the roulette wheel or the stochastic universal sampling) is superior. In case of ranking selection the difference is imperceptible, while in case of fitness proportional selection the roulette wheel algorithm performed better with 10,3 %. The tournament selection methods proved to be the most efficient, and the best solution was produced by a deterministic tournament with tournament size 8.

Recombination

As illustrated in the figure below the rate 0.8 is a dividing value; the smaller values show significantly inferior performance, and the larger ones are quite similar to each other. The figure shows the outcome of an experiment series, where the 1.0 gave the best result, however, I conducted more experiment series where also 0.9 and 0.95 gave superior results. Owing to the subtle difference I don't see any reason for applying smaller crossover rate than 1.0.

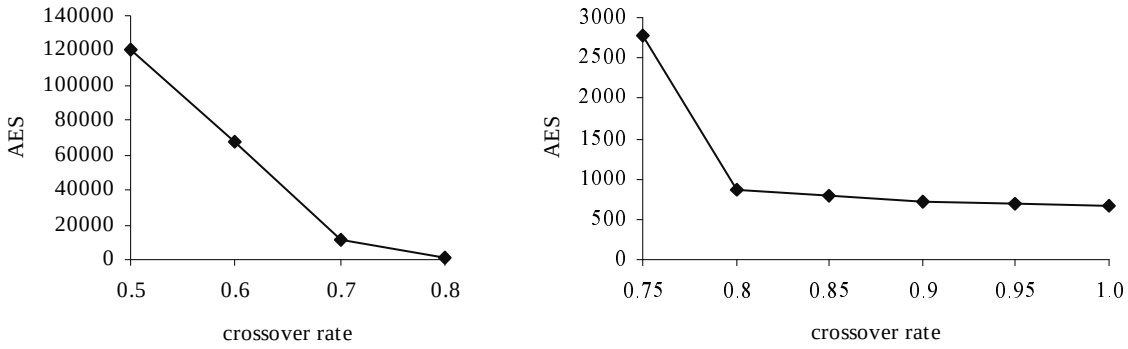


Figure 4-17 AES plotted against the crossover rate

Mutation

On average, the swap mutation turned out to be superior with 30%, notwithstanding the two operators perform roughly similarly in half of the cases (if the mutation rate is ≥ 0.5).

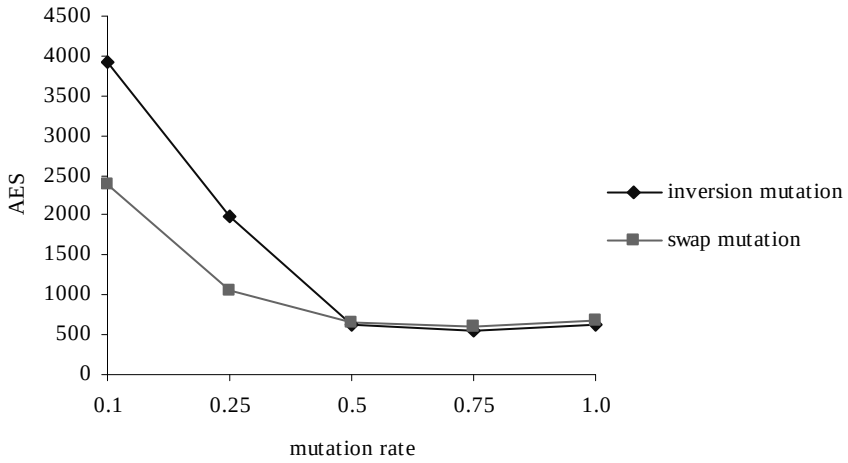


Figure 4-18 Comparison of the mutation operators

Survivor selection

It is unambiguous that the "best from union" selection method performs most efficiently. Almost such results were produced by the "replace worst" selection method with $0.25-0.5$ generational gap.

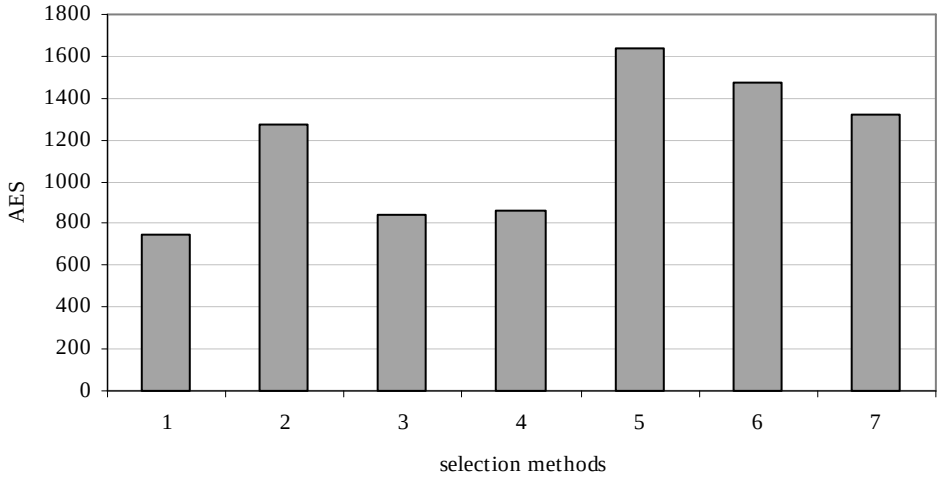


Figure 4-19 Comparison of the survivor selection methods

The meaning of the certain columns:

1) best from union
2) replace worst (gap = 0.1)
3) replace worst (gap = 0.25)
4) replace worst (gap = 0.5)
5) replace worst (gap = 0.75)
6) deterministic tournament selection (k = 2, with replacement)
7) deterministic tournament selection (k = 2, without replacement)

Population size

Every problem size has its own ideal population size. In case of small populations the individuals are getting similar to each other in due course and the evolution can rely only on the mutation operator, while if the populations are too big, we may perform extra evaluations.

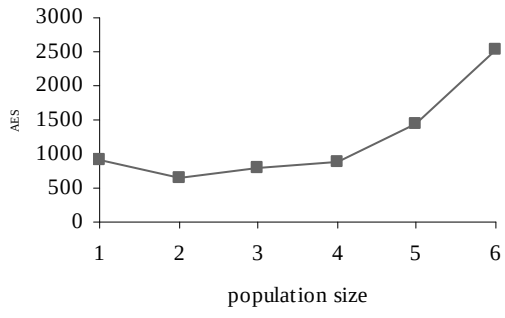


Figure 4-20 AES plotted against the population size (32-length cycle)

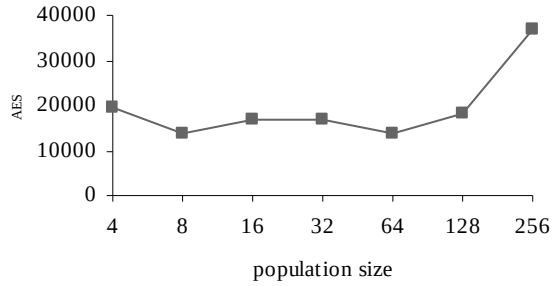


Figure 4-21 AES plotted against the population size (64-length cycle)

4.2.3.3.3 Conclusions

Here I report the summary of the experiments of the previous section. These are the parameters that proved to be the most efficient, hence they will form the basis any of the further experiments.

GA model	steady-state
Chromosome length (L)	$k^n / 2$
Population size	$L / 2$
Representation	permutation
Recombination	ordered crossover ($p_c = 1.0$)
Mutation	swap ($p_m = 0.5$)
Selection	deterministic tournament (k = 8)
Replacement	best from union

Table 5 The outcome of the hand-tuning

4.2.3.4 Parameter Tuning of the Two-dimensional GA

4.2.3.4.1 Algorithm Setups

My experiments concerning the parameter tuning are based on the AES measure. I examined the performance of the certain operators in case of a $(4,4;2,2)_2$ - PPM. The parameter setup of the different experiments differs only with respect to the actual operator, otherwise it is the based on the table below.

GA model	generational (gap = 0.8)
Chromosome length (L)	$4 \cdot 4 = 16$
Population size	16
Representation	integer
Recombination	uniform crossover ($p_c = 1.0$)
Mutation	random resetting ($p_m = 0.25$)
Selection	ranking (s = 2.0, roulette wheel)
Replacement	best from union

Table 6 GA default setup (for parameter tuning)

4.2.3.4.2 Test Results

Parent selection

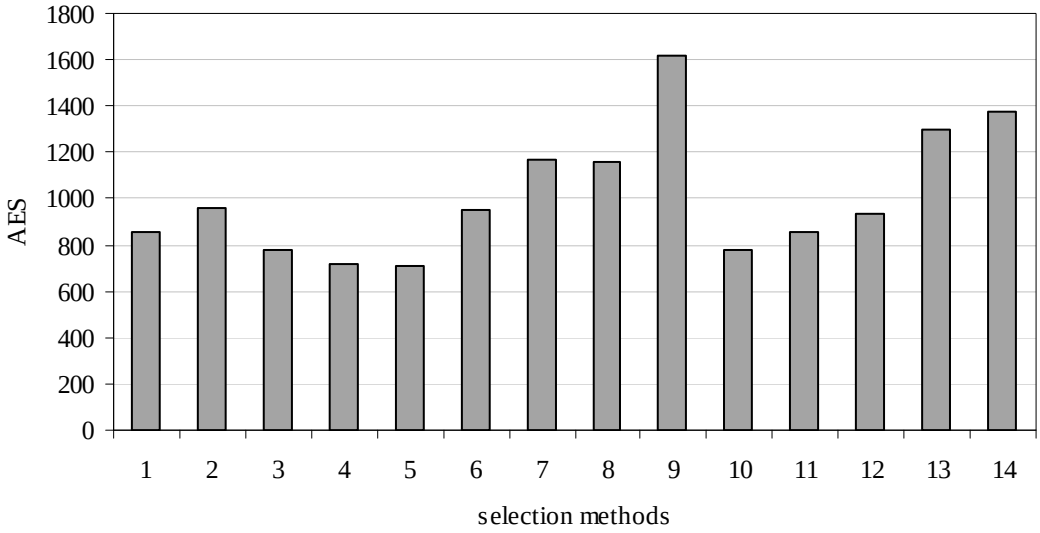


Figure 4-22 Comparison of the parent selection methods

The meaning of the certain columns:

1) ranking selection (s = 1.5, roulette wheel)
2) ranking selection (s = 2.0, roulette wheel)
3) ranking selection (s = 1.5, stochastic universal sampling)
4) ranking selection (s = 2.0, stochastic universal sampling)
5) fitness proportional (roulette wheel)
6) fitness proportional (stochastic universal sampling)
7) deterministic tournament (k = 2)
8) deterministic tournament (k = 4)
9) deterministic tournament (k = 8)
10) stochastic tournament (k = 2, p = 0.5)
11) stochastic tournament (k = 2, p = 0.6)
12) stochastic tournament (k = 2, p = 0.7)
13) stochastic tournament (k = 2, p = 0.8)
14) stochastic tournament (k = 2, p = 0.9)

The best performing method is the fitness proportional selection, however, both the ranking selection (4) and the stochastic tournament (10) provided similar results and the difference is quite subtle. It is also apparent that the performance of th stochastic tournament is deteriorating as the value of *p* is increasing. It is due to the fact that the diversity of the population is an important issue, and if only the best individual survives, then it may lead the evolution to a possibly wrong direction.

Recombination

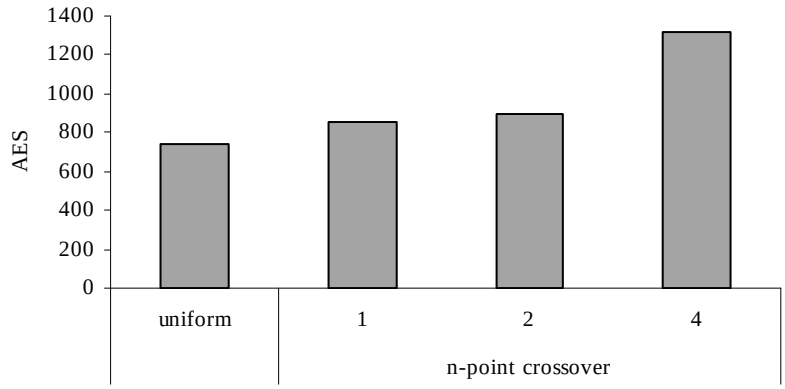


Figure 4-23 Comparison of the crossover operators

The values below the axis stand for the number of crossover points. There is no significant difference between the methods; the uniform crossover, the 1-point and the 2-point crossover performed almost similarly, however, the uniform crossover was the superior with a subtle difference.

Mutation

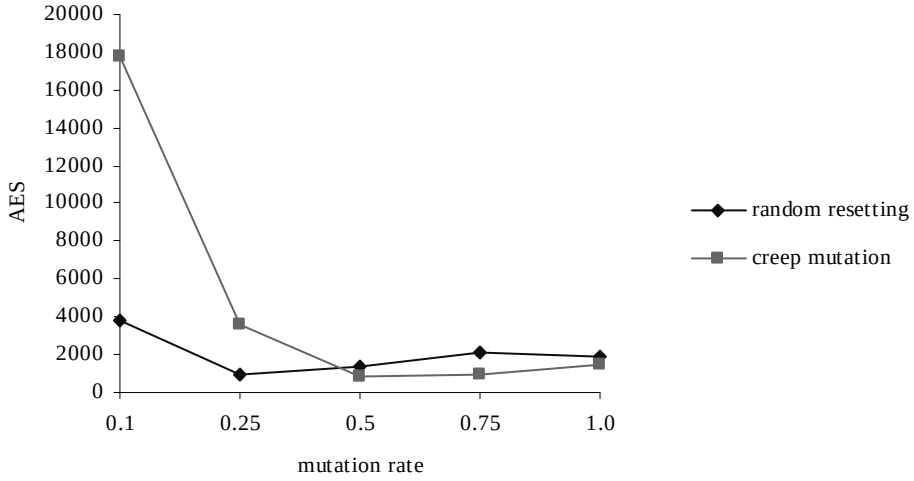


Figure 4-24 Comparison of the mutation operators

The rate 0.5 is a dividing value, because the random resetting is significantly superior with smaller rates, but the creep mutation performs better with larger ones, however, in this case the difference is not so considerable.

Survivor selection

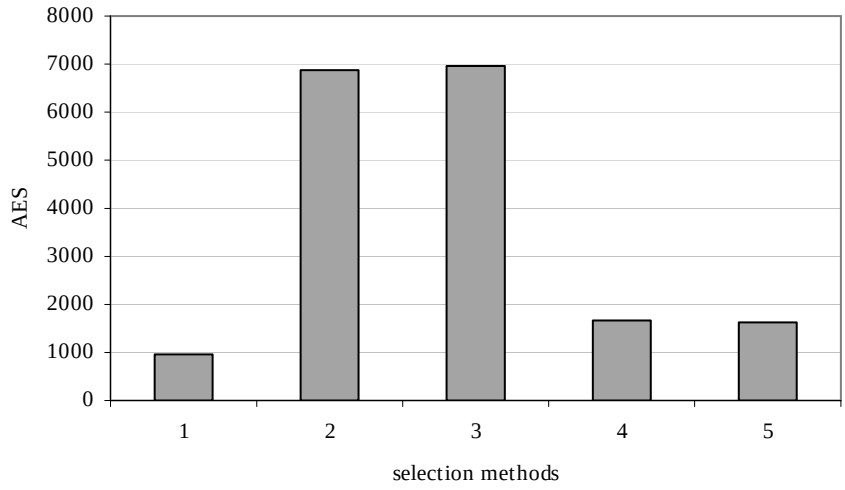


Figure 4-25 Comparison of the survivor selection methods

The meaning of the certain columns:

1) best from union
2) replace worst (gap = 0.75)
3) fitness proportional (roulette wheel algorithm)
4) deterministic tournament selection (k = 2, with replacement)
5) deterministic tournament selection (k = 2, without replacement)

The “best from union” selection method is significantly superior to the replace worst and the fitness proportional selection, and it is also superior to the deterministic tournament selection methods, but in this case the difference is not so considerable.

Population size

Likewise in the case of the one-dimensional GA, every parameter set has its ideal population size. The figure shows that 32 is the ideal population size for a map with 16 elements.

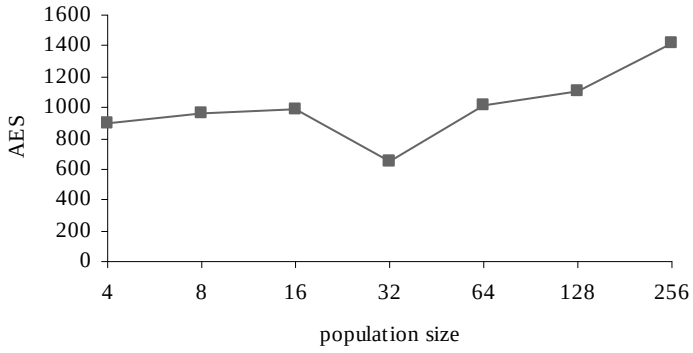


Figure 4-26 AES plotted against the population size

4.2.3.4.3 Conclusions

Here I report the summary of the experiments of the previous section. These are the parameters that proved to be the most efficient, hence they will form the basis any of the further experiments.

GA model	steady-state
Chromosome length (L)	k^n
Population size	L / 4
Representation	integer
Recombination	uniform crossover (p_c = 1.0)
Mutation	random resetting (p_m = 0.25)
Selection	fitness proportional
Replacement	best from union

Table 7 The outcome of the hand tuning

4.2.4 Comparison of the Reference and Genetic Algorithms

It is difficult to compare two different algorithms that have practically nothing in common. The only possibility is to choose a measure in both cases that characterizes the behaviour of the algorithm. These measures are the number of *basic steps* + *backtracks* in case of backtrack search and the *AES* in case if GA.

I compared the one-dimensional algorithms, because the permutation representation works more efficiently. I applied the GA with the parameters that turned out to be the best working during the hand tuning (see section 4.2.3.3), the parameters are summarized in the section **Error! Reference source not found.**

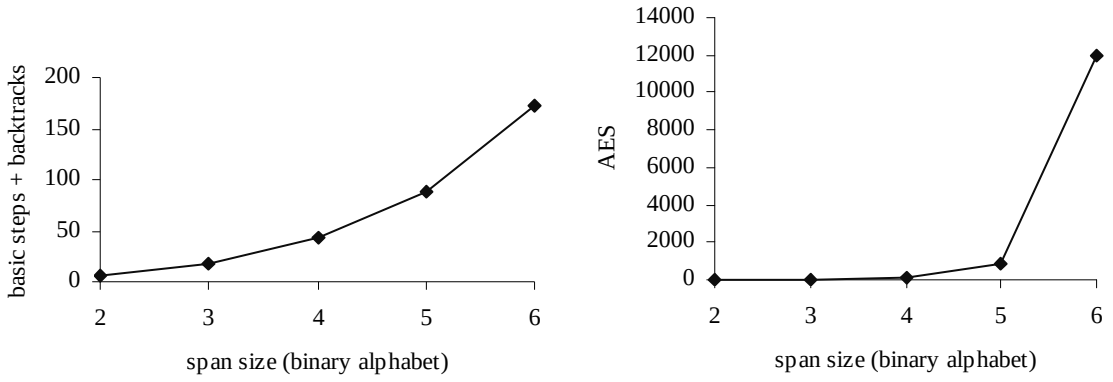


Figure 4-27 Performance of the algorithms

It is apparent that the graph of the backtrack search is smoothly increasing, while that of the GA is very steep. This shows that the backtrack search is superior in case of small parameter sets. I also made some experiments with large parameter sets, and my observations about the

progress of the search suggest that the GA is able to find a solution where the backtrack search get stuck and has no chance due to the size of the search space. The supposed relation between the behaviour of the algorithms in the long run is illustrated in the figure below.

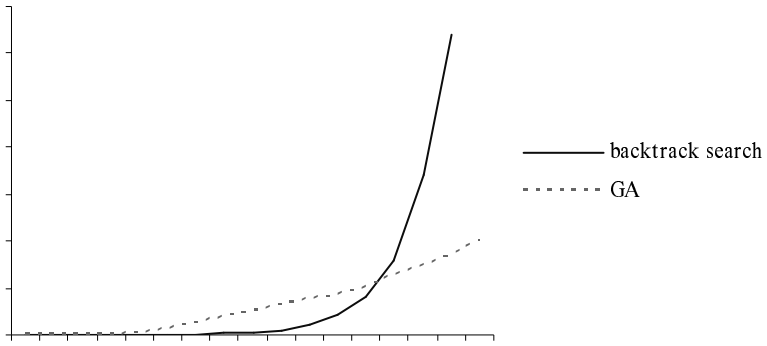


Figure 4-28 The relation between the algorithms in the long run

4.2.5 Practical Results

I managed to find maps with a great variety of small parameters. In case of larger parameters the integer representation was not able to produce results in reasonable time, because the evaluation of an individual is very costly in case of large maps. Unfortunately, this means that my aim to find a "brand-new" map was not granted, however, the progress of the evaluation indicates the capability of GAs in this problem context.

In the following I report my conclusions about the experiments with large maps.

The smallest periodic perfect map whose existence is not shored up by the theory has the parameters $(216,216;3,2)_6$. Its CPU demands are nearly the same as the smallest four-dimensional tori and it takes quite much time to evolve even an individual (see below).

Results regarding the three-dimensional case

I searched for the smallest possible three-dimensional torus, a $(64,2,2;2,2,2)_2^3$ - de Bruijn Torus. Note that there are more choices concerning the dimensions of this map and this is merely my preference among the possible ones.

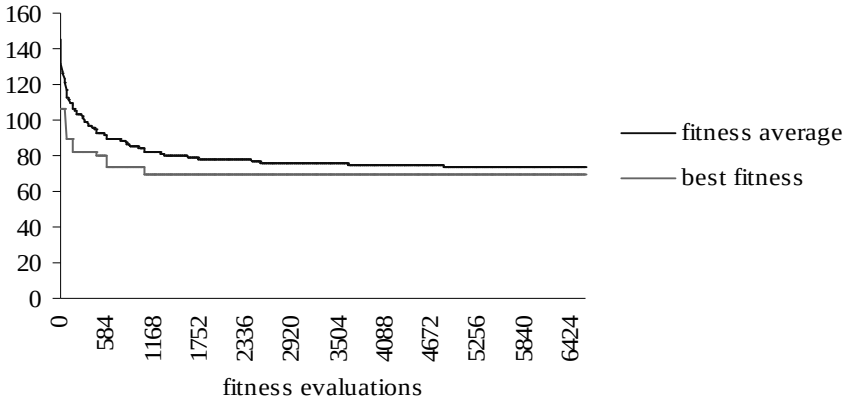


Figure 4-29 The progress of the evolution

The above figure illustrates the progress of the evolution: rapid progress in the beginning (the fitness average decreased from 145 to 80 during ~1300 evaluations) and flattening out later on (it decreased from 80 to 74 during ~3500 evaluations).

Results regarding the four-dimensional case

In the four-dimensional case the search meets with obstacles, because the size of an individual is too big and my fitness definition is quite costly in case of big individuals: it makes

$$\frac{length \cdot (length + 1)}{2}$$

comparison (the *length* stands for the length of the chromosome). One comparison means the comparison of two windows. It takes one step in the best case (if the first element doesn't match), $m \cdot n$ steps in the worst case (if the two windows are equal), so the average number of steps is:

$$\frac{m \cdot n \cdot (r_1^2 \cdot r_2^2 \cdot r_3^2 \cdot r_4^2 + r_1 \cdot r_2 \cdot r_3 \cdot r_4)}{4}.$$

I made my experiments with the smallest possible four-dimensional torus, a $(16,16,16,16;2,2,2,2)_2^4$ - de Bruijn Torus. The above formula gives $\approx 4 \cdot 10^9$ steps in this case, which can be made in ~6,4 hours on a 1200 MHz CPU. It can be seen that finding such a map is a very long-lasting venture with the current computational capacity, but at least not impossible.

4.2.6 Theoretical Results

4.2.6.1 The Number of Tokens in a de Bruijn Cycle

While improving the one-dimensional genetic algorithm I noticed that every de Bruijn cycle consists of a definite number of tokens. Let's consider the case when $k = 2$ and $n = 4$. Figure 4-30 shows a possible $(16;4)_2$ - de Bruijn Cycle.

[0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1]

Figure 4-30 A $(16;4)_2$ - de Bruijn Cycle

A token is an uninterrupted sequence of identical numbers. The de Bruijn cycle in Figure 4-30 has the following tokens:

$$\{<0\ \ 0\ \ 0\ \ 0>, <1\ \ 1\ \ 1\ \ 1>, <0\ \ 0>, <1\ \ 1>, <0>, <0>, <1>, <1>\}$$

The relation between *k*, *n* and the number of tokens is as follows.

Length of the token	Number of the token
n	k
$n-1$	$(k-2)k$
$n-2$	$(k-1)^2 k$
$n-3$	$(k-1)^2 k^2$
...	...
$n-i$	$(k-1)^2 k^{i-1}$

4.2.6.2 The Number of de Bruijn Cycles

The number of spanning trees of a (k,n) - de Bruijn Graph (see Section 2.1.7) is as follows:

$$k^{k-2} \cdot x \cdot \prod_{i=1}^{n-2} f^i(x), \text{ where } f(x) = k^{k-1}(x)^k \text{ and } x = k^{1+\sum_{j=2}^{k-1} 2j}.$$

I obtained the above formula by observing the results of a number of experiments. I used a program (*DBGraph.java*) to create de Bruijn graphs and my final goal was to determine the number of their spanning trees. These graphs needed to be converted to an equivalent form without self-loops before creating their in-degree matrix. When having these matrices I used the *Maple* software to get the determinant of their minors. I verified the formula for the cases when $k = 1,2,\ldots,6$ and $n = 1,2,3,4$.

Note that because the number of spanning trees of a (k,n) - de Bruijn Graph will be always the power of *k*, it is sufficient to compute the logarithm of the above formula, namely

$$(k-2)+y+\prod_{i=1}^{n-2} f^{i}(y)\text{, where } f(y)=(k-1)+k\cdot y \text{ and } y=1+\sum_{j=2}^{k-1}2j\,.$$

Applying this formula the number of spanning trees of a (k,n) - de Bruijn Graph can be determined in $\Theta(n+k)$ time, which is much faster than any other algorithm known so far (see section 2.1.7).

Considering the facts about Euler paths (see section 2.1.7) the number of $(k^n;n)_k$ - de Bruijn Cycles can be given by the following formula:

$$\left((k-2)+y+\prod_{i=1}^{n-3} f^{i}(y)\right)\cdot\left((k-1)!\right)^{k^{n-1}}\text{, where } f(y)=(k-1)+k\cdot y \text{ and } y=1+\sum_{j=2}^{k-1}2j\,.$$

4.2.6.3 The Number of Tokens in a Two-dimensional Periodic Perfect Map

Although I didn't manage to devise the correct mathematical relationship concerning the number of tokens in a periodic perfect map, my experiments show that there is some kind of relationship. Here I report my observations.

I had the possibility of observing only two periodic maps ($(4,4;2,2)_2$ - PPM and $(9,9;2,2)_3$ - PPM), because the search space of next possible map ($(16,16;2,2)_4$ - PPM) is too large, and none of my algorithm gave result in reasonable time. I wrote a Java program (*parse.java*) that parses the matrices and outputs the number of tokens both in horizontal and in vertical directions (it takes the file output by my search algorithm as input). The result is the following.

(1) $(4,4;2,2)_2$ - PPM

$$\begin{pmatrix}0&0&0&1\\0&0&1&0\\1&0&1&1\\0&1&1&1\end{pmatrix},\begin{pmatrix}0&0&0&1\\0&1&1&1\\1&0&1&1\\0&0&1&0\end{pmatrix}$$

Figure 4-31 The two possible $(4,4;2,2)_2$ - PPM

The two above tori show the following regularity: both of them have two $<1>$ tokens and two $<111>$ tokens in both directions. This applies also to the all zeros tokens.

(2) $(9,9;2,2)_3$ - PPM

The number of such maps are unknown so far, hence I examined 208 of them.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 2 & 1 & 1 & 2 & 0 \\ 2 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 1 & 1 & 1 & 2 & 0 & 2 \\ 0 & 2 & 2 & 1 & 0 & 2 & 1 & 1 & 2 \\ 2 & 1 & 0 & 2 & 1 & 1 & 1 & 1 & 2 \\ 2 & 0 & 2 & 2 & 1 & 2 & 2 & 1 & 1 \\ 2 & 1 & 0 & 1 & 2 & 2 & 2 & 2 & 0 \\ 0 & 1 & 2 & 2 & 0 & 2 & 1 & 1 & 2 \end{pmatrix}$$

Figure 4-32 A possible $(9,9;2,2)_3$ - PPM

There are 54 tokens in both directions: 18 *0*-token, 18 *1*-token and 18 *2*-token. For example, the map in the above figure has the following *0*-tokens:

$<0\quad 0\quad 0\quad 0\quad 0\quad 0\quad 0\quad 0>$	1 piece
$<0>$	16 pieces
$<0\quad 0\quad 0>$	1 piece

Other possible maps have different distribution regarding the size of the tokens. I could not determine which sizes are permissible, but I suspect that not all of them. Just recall that in case of the $(4,4;2,2)_2$ - PPM there are two $<1>$ tokens and two $<111>$ tokens, but there is no $<00>$ token at all.

To devise a similar formula (*F*) that of the one-dimensional case, we have to observe many cases and furthermore it will be a bit more intricate, because it has not two but five arguments: $F(R,S,m,n,k)$. These observations take much time applying both the backtrack search and the genetic algorithm.

4.2.6.4 The Number of Two-dimensional Perfect Map

Although I didn't manage to devise a similar characteristic function that gives the number of perfect maps as in the one-dimensional case, here I report all my observations and conclusions regarding this issue.

The following table contains all the possible binary maps with two by two windows. Their existence is proven theoretically (see section 2.2.2.2).

	parameter sets	the number of the maps
periodic	$(4,4;2,2)_2$	2
semi-periodic	$(2,16;2,2)_2$	$20736 = 2^8 \cdot 3^4$
	$(3,8;2,2)_2$	$264 = 2^3 \cdot 3 \cdot 11$
	$(5,4;2,2)_2$	$64 = 2^6$
aperiodic	$(2,17;2,2)_2$	$331776 = 2^{12} \cdot 3^4$
	$(3,9;2,2)_2$	$2560 = 2^9 \cdot 5$
	$(5,5;2,2)_2$	$800 = 2^5 \cdot 5^2$
	$(9,3;2,2)_2$	$2560 = 2^9 \cdot 5$

I counted up the number of these maps by means of a backtrack search algorithm with a view to observe the inherency in their numbers and devise a formula as in the one-dimensional case (see section 4.2.6.2). Unfortunately, these pieces of information are insufficient to draw the correct conclusion, and finding all the maps is a time-consuming venture even in case of small parameters. For example, the algorithm made more than 90 million backtracks to find all the $(2,16;2,2)_2$ - Semi-Periodic Perfect Maps (it took 123 minutes on a 3000 MHz CPU).

The theoretical basis of their number is in close connection with the number of Euler paths in a graph as in the one-dimensional case (see section 2.1.7). I constructed the two-dimensional equivalents of de Bruijn graphs as follows. Note that both the de Bruijn graphs and their two-dimensional equivalents are two-dimensional graphs, the attribute “two-dimensional” stands for the dimension of the map represented by the graph.

The vertices of the graph stand for the decimal values of the possible windows, and the edges are generated as illustrated in the figure below (every vertex has k^m incoming and k^m outgoing edges):

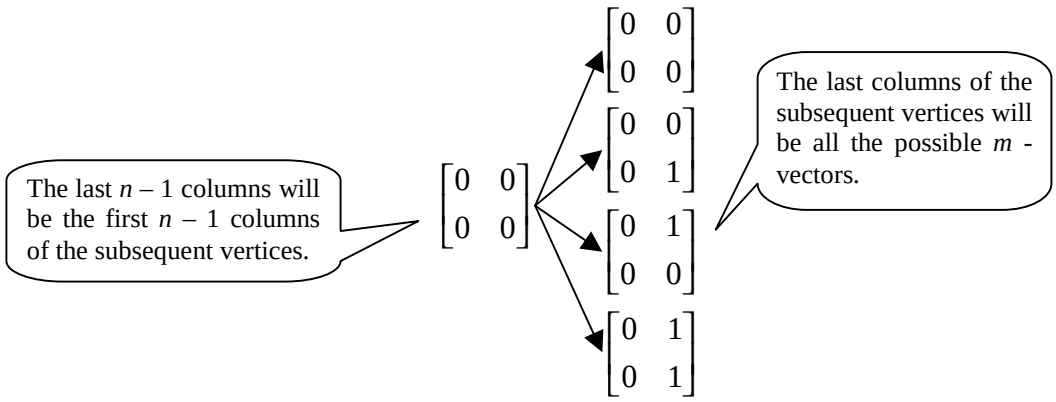


Figure 4-33 Generation of the outgoing edges (binary alphabet, 2×2 window)

Let us consider the following $(4,4;2,2)_2$ - Periodic Perfect Map:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}.$$

The corresponding graph (the edges $5 \rightarrow 15$, $9 \rightarrow 13$, $10 \rightarrow 0$ and $6 \rightarrow 12$ are omitted for layout reasons):

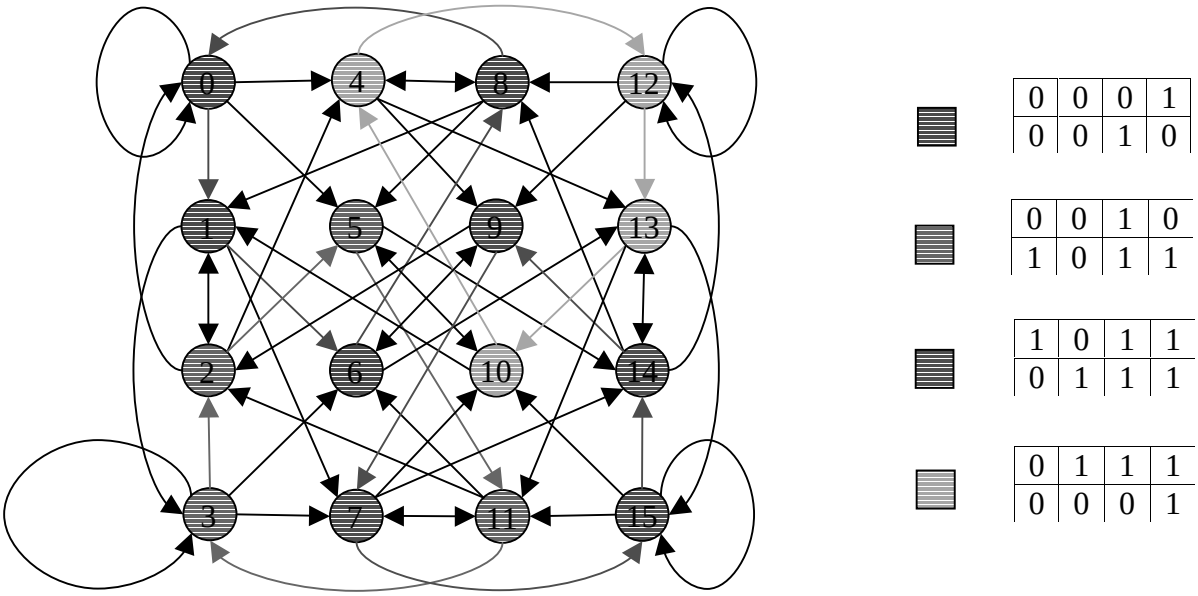


Figure 4-34 The de Bruijn graph of a two-dimensional map

Note that every row of the map is a semi-periodic $m \times S$ matrix. These matrices form m disjoint cycles in the graph (illustrated by different colors). The number of such disjoint cycles gives the number of the corresponding perfect map. This mathematical relationship is unknown as yet.

5 Summary and Final Remarks

We have seen that the backtrack search algorithm can be useful in a few cases, if the maps are small and hence their search spaces are manageable with the current computational capacity within a reasonable time. I used this algorithm to observe interesting characteristics of maps with small parameters, and apply the obtained information to tune the genetic algorithms.

Genetic algorithms can cope with large search spaces, where the backtrack search algorithm has no chance whatever. In the one-dimensional case I applied a *permutation representation*, while in the higher dimensions only the straightforward *integer representation* was available. Unfortunately, my fitness definition was costly and it took quite much time to find a larger map.

My experiments show that GAs are suitable for giving answers to this complexity problem. Compared to the backtrack search algorithm it is inferior in case of small parameter sets, but the progress of the algorithms in case of larger parameter sets suggest that GAs are capable of providing solutions in reasonable time, while the backtrack search algorithms are not.

Although my ambitions to realize my challenging aim – namely finding a map, whose existence was only a conjecture so far –, were not granted, I think this thesis made a step towards getting to know perfect maps, and designated a path where it is maybe worth to continue the research.

Future Work

First, we have to observe a feature that makes the permutation representation possible in higher dimensions, as well. I believe that the perfect maps have this outward appearance – namely the regularity of the number of tokens – not only in the one-dimensional case but in higher dimensions, as well. Moreover I believe that a genetic algorithm with a representation provided by the tokens could be the most efficient tool in finding such maps.

Appendix A

Documentation of the Perfect Map Generator software

A.1 User Documentation

A.1.1 System requirements

- Java Runtime Environment (version 1.4.2.)
- Netscape browser in case of Linux operating system (optional, only for visiting a webpage for further information about the program)

A.1.2 Parameter Settings

First we have to choose the problem (regarding the periodicity of the map), the method and the parameters. Filling in the parameters is possible either by clicking in the appropriate field, or by navigating between the fields with the *Tab* key. The entered input value is checked by the program immediately when the field has lost the focus, i.e. if we are trying to fill in the next field. If the value is legal, the field becomes inactive, indicating that the program has accepted the input. Otherwise it gives a warning and the set of possible values. Note that as long as an input is not accepted, there is no possibility to go on with filling in the next field. If we would like to alter an already accepted value, it can be done by pushing the *Clear* button. See that it will reset all the parameters (it is needed because the parameters influence one another's allowable value).

Remarks about the certain fields:

i) alphabet: This is the first one we have to fill in (there is no other choice, because only this field is active). Its maximum value is 127.

ii) window size: After having the alphabet, first the *m*, then the *n* field needs to be filled in. The program checks whether k^{mn} exceeds the value of $2^{31}-1$. Note that k^{mn} is the number of the elements in the map.

iii) map size: Given the alphabet and the window size, there are more possibilities concerning the size of the map. The user shows a preference by filling the *R*. Depending on the type of periodicity the program determines the maximum allowable value of both *R* and *S*, so the *S* field is not editable by the user, but it is filled in automatically right after we have left the *R* field.

After having all the required parameters, we have to decide whether to find all the possible maps. Note that this feature is available only in case if we have chosen backtrack search algorithm as method, otherwise this box is inactive.

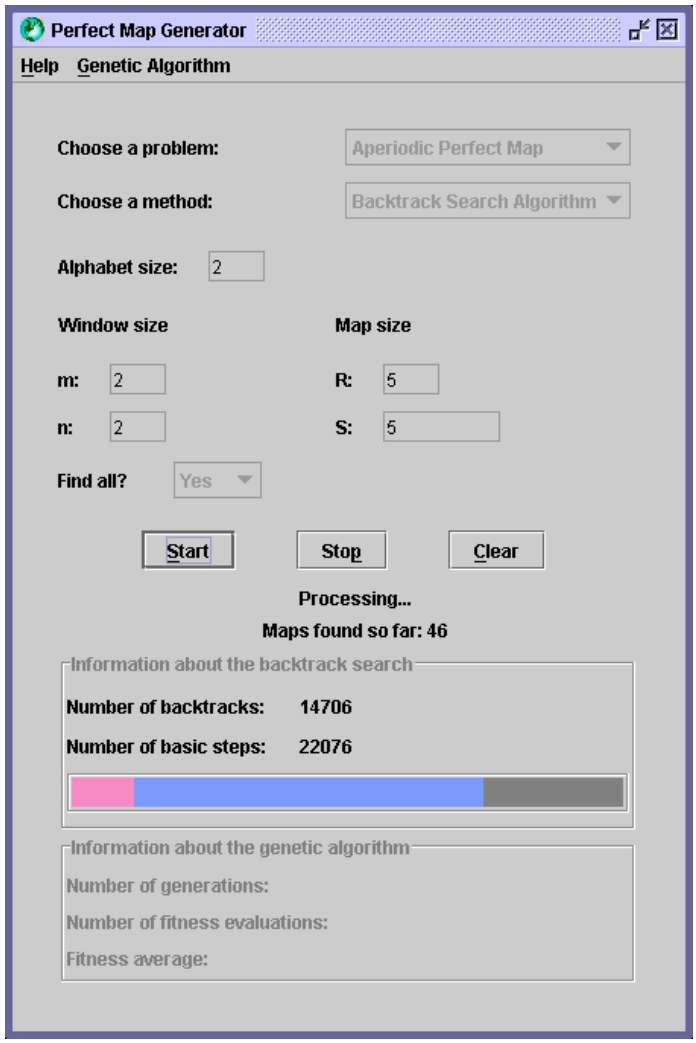


Figure 0-1 The application

Settings for the Genetic Algorithm

There is a separate panel (see Figure 0-2) to control the parameters and the operators of the genetic algorithm. It can be reached through the *Genetic Algorithm* menu of the menu bar, by clicking on the *Settings* item.

There are choices regarding the following options and operators, respectively:

- Population size
- Parent selection
- Recombination
- Mutation
- Survivor selection
- Print option (whether to print the populations to the output file)

For further information about the certain operators see the specification of the genetic algorithm (Section 4.1.2.2).

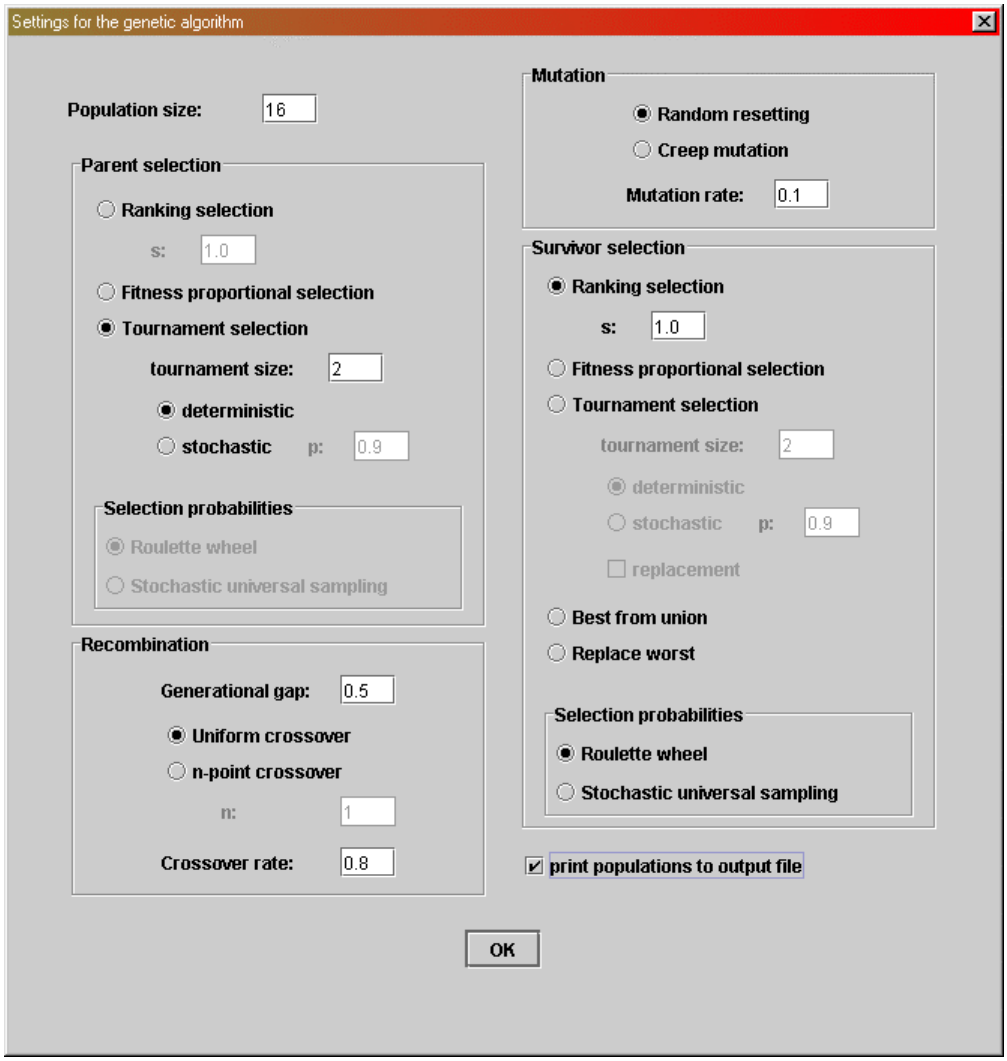


Figure 0-2 Settings for the genetic algorithm

A.1.3 Interpretation of the Output

After having started the search, the appropriate information panel will be active. In case of backtrack search algorithm the upper panel informs about the number of backtracks and basic steps made so far, and a progress bar helps to keep track of the actual state of the search. It shows the levels of the search tree, and there are three different stripes rolling on it: the gray one indicates the maximum level reached so far, the blue one stands for the actual level, and the pink one shows the lowest level to where we have made a backtrack already. In case of genetic algorithm the lower panel informs about the number of generations, number of fitness evaluations and about the fitness average in the certain populations.

If we have chosen the option to find all the possible maps, the number of ones found so far is printed on the screen, as well. If the search had terminated the user is informed about the time needed (in milliseconds). These pieces of information and also the maps found are printed to an output file named *{apm, spm, ppm}_k_(R,S)_(m,n)_{bt, ga}.txt*, where *R*, *S*, *m* and *n* stand for the actual value of the parameters.

A.2 Development Documentation

The program was written in *Java* (Java 2, Standard Edition, v1.4.2.).

A.2.1 Graphical User Interface

The GUI is implemented in *PerfectMapGenerator.java*.

A.2.1.1 Components

The GUI consists of a main frame and three modal dialog windows that can be launched by clicking on the certain items in the menu bar. Two of them provide information about the program (help and author information) and the third one (*GADialog*) provides an interface to control the parameters and the operators of the genetic algorithm.

The following two figures illustrate the main components of the main frame and the *GADialog* dialog window, respectively. By main components I mean the ones on the first level (note that the components are embedded hierarchically into each other, so the first-level ones are the components contained directly by the root pane) and those ones that are on lower levels but contain other components or have an important role in the layout.

Notice that this embedding is one level deep in case of the main frame, and four levels deep in case of the *GADialog* window. The more-level embedding has no special role; it serves only convenience considerations to facilitate the layout.

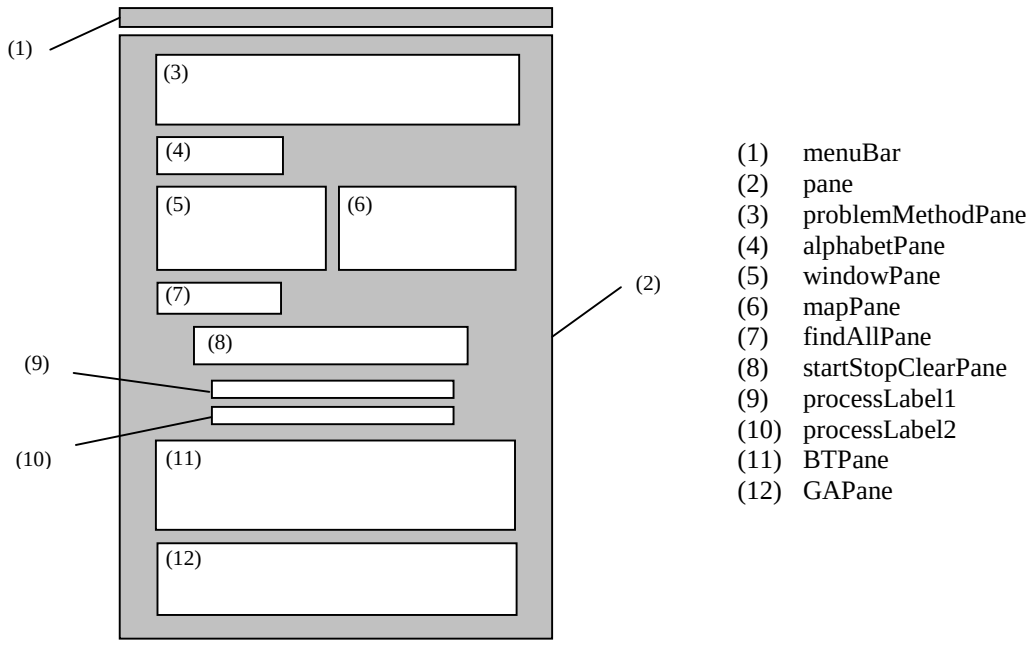


Figure 0-3 Main components of the GUI

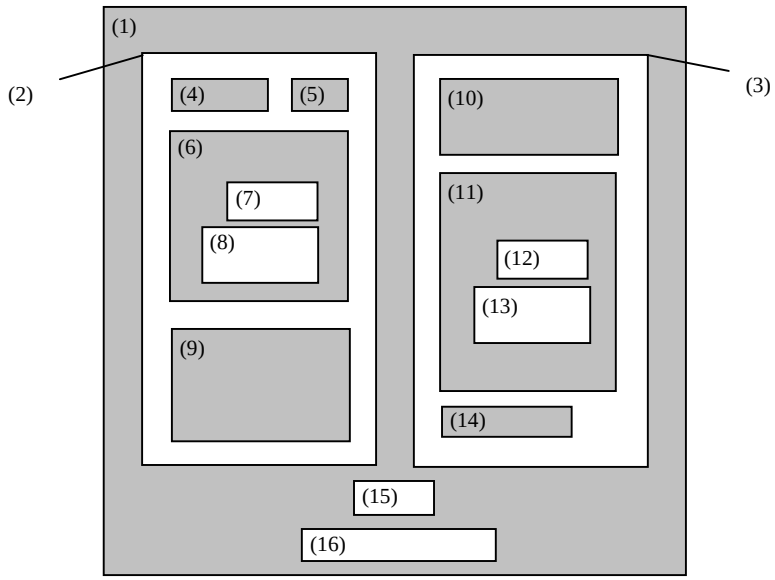


Figure 0-4 Main components of the *GADialog* window

There are four methods directing the graphical look of the application:

`private JMenuBar createMenuBar()`	Creates the menu bar with two menus: *helpMenu* and *GAMenu*. It adds two menu items (*helpMenuItem* and *aboutMenuItem*) to the *helpMenu*, and one item (*settingsMenuItem*) to the *GAMenu*.
`private Container createGADialogPane()`	Provides the appearance of the *GADialog* window by creating its top-level container, and embedding the required components into this container hierarchically.
`private Container createContentPane()`	Provides the appearance of the main frame by creating its top-level container, and embedding the required components into this container hierarchically.
`private static void createAndShowGUI()`	Creates the main frame, and sets its various features, among others the menu bar and the content pane.

There are components, which are controlled by another object. These are the ones, which inform the user about the actual state of the search: *nbtValue* (number of backtracks), *nbsValue* (number of basic steps), *ngsValue* (number of generations), *nfesValue* (number of fitness evaluations), *fitnessAverageValue* (fitness average), *pbta* (progress bar), *processLabel1* (informs about the actual state of the search, namely whether it is in progress, stopped or finished), *processLabel2* (number of maps found so far). All of these components are objects of type *JLabel*, except *pbta*, which is an instance of the embedded class *ProgressBTArea*.

This class performs the custom painting of the progress bar. These components are public fields and can be reached by other objects without any function call.

A.2.1.2 Events

The class *PerfectMapGenerator* implements four kinds of event listeners, which are discussed in detail in what follows.

i) ActionListener

Except the components that serve for parameter input in the main frame (*alphabetTextField, mTextField, nTextField, RTextField* and *STextField*), the two checkboxes (*replacementButton* and *printToFileButton*) and the one that realizes a hyperlink (*webPageLabel*), all the other components use this kind of listener for event handling.

This method has as many branches as the number of components that registered themselves for an action event. To decide which component had fired the actual event, every one of them has to provide the event object with an identifier by invoking their *setActionCommand()* method. The branch-on conditions can be gained from the *ActionEvent* object by invoking its *getActionCommand()* method.

About the certain actions:

- **help**, **about** Both of them create a standard modal dialog window ("*Help*" and "*About*") provided by the class *JOptionPane*, the former by invoking *the showOptionDialog()* method, the latter by invoking the *showMessageDialog()* method of the class. The structure of the message dialog is strictly defined, but in case of the option dialog we may perform many kinds of customization: its content pane consists of a lot of inactive labels (most of the text is read from a file named *help.txt*), and an active one (*webPageLabel*), which is registered for mouse events (see the paragraph *MouseListener*).

- **GASettings** It creates a fully customizable modal dialog window ("*Settings for the genetic algorithm*") by means of the class *JDialog*. All the components in the *GADialog* dialog window have a very simple role, namely to set a variable that represents a parameter or an operator of the genetic algorithm (*populationSize, parentSelection, s, s2, k, k2, tournamentSelection, p, p2, replacement, probMethod, probMethod2, recombination, n, generationalGap, crossoverRate, mutation, mutationRate, survivorSelection, printToFile*). It is very important that all these variables should have an initial value, because the user is not obliged to fill in any value or even to open this dialog window.

- **OK** The values in the *GADialog* dialog window will be saved if the user clicks this button. This saving is not a permanent but a temporary one, namely it affects the functioning of the program during one run. If it is closed and launched again, the default values will be loaded.

  There are a lot of commands (*rankingSelection, FPSelection*, etc.) – not itemized here, whose role is only to indicate that the state of a radio button or a combo box had

changed, and the appropriate variable needs to be set. Note that the states of radio buttons and combo boxes in the *GADialog* dialog window will be saved automatically, without having to click the *OK* Button. The reason for this is that the states of these options are loaded according to the values of the variables mentioned in the GASettings item of this paragraph.

- **problem**, **method**, **findAll** The mechanism is very similar in case of these commands. First, two pieces of information are needed: which *JComboBox* object was the event fired by, and which item of the combo box is selected. The former can be gained by invoking the *getSource()* method of the *ActionEvent* object and the later by invoking the *getSelectedItem()* method of the returned *JComboBox* object. After having the selected item, the appropriate variables (*whichProblem*, *whichMethod* and *findAll*, respectively) can be set accordingly. In case of "method" we have to take care also of the state of the *findAll* combo box, because it should be editable only in case of backtrack search algorithm.

The following three commands have the joint feature that they are applicable depending on the actual state of the search (indicated by the variable *inProgress*). The *start* and *clear* have an effect only in the case when the search is not in progress, and the *stop* is applicable otherwise.

- **start** It is legal only if the user had filled in all the required parameters (it is indicated by *mapSizeRReady* – note that if the field *R* is ready, all the other fields are bound to be ready). Otherwise only a warning is written on the screen (via *processLabel1*). Depending on the problem and the method (indicated by *whichProblem* and *whichMethod*) different algorithms need to be launched. Launching means that we instantiate the class of the appropriate search algorithm, and pass the yielding object to a thread that will invoke its *start()* method. In case of backtrack search algorithm this thread is an instance of the embedded class *BTTrigger*, in case of genetic algorithm it is an instance of the embedded class *GATrigger*. The only task of these classes is to launch the algorithm.

- **stop** Its only task is to invoke the *kill()* method of the executive thread. Every algorithm has a static variable *stopped* and the only thing the *kill()* method has to do is to set this variable of the appropriate algorithm to true.

- **clear** This command resets all the text fields in the main frame to their initial state.

ii) ItemListener

There are two checkboxes in the *GADialog* dialog window (*replacementButton* and *printToFileButton*) that are registered for item events, hence this method has two branches according to the actual checkbox. The branch-on conditions can be gained from the *ItemEvent* object by invoking its *getItemSelectable()* method, which returns the actual checkbox object. Each branch has following two branches, because we have to decide whether the click selected or deselected the check box. This can be done by means of the *getStateChange()* method of the *ItemEvent* object.

iii) FocusListener

This listener is used for parameter input in the main frame. The program checks the input values as soon as it gets them. Because most of the inputs have influence on one another's allowable value, it is important to inform the user at the earliest opportunity if there is something illegal. This earliest opportunity is when the user had finished the filling in of a field. The application should be informed somehow about this event, and I chose an ergonomical solution that deviates from the traditional OK-button technique, namely the loss of focus. To this end the class *PerfectMapGenerator* implements the *focusLost()* method of the *FocusListener* interface.

This method has five branches according to the actual text field. The branch-on conditions can be gained from the *FocusEvent* object by invoking its *getComponent()* method, which returns the actual text field object. The value control and the field state settings (it is important that user should fill in the fields in the proper order, that's why only the actual field is active) are managed accordingly in the certain branches.

iv) MouseListener

This listener is used in the only case when the user clicks on the hyperlink in the *Help* dialog window. The class *PerfectMapGenerator* implements three of the mouse listener methods, viz. *mouseClicked(), mouseEntered(),* and *mouseExited()*. The first one invokes the *displayURL* method of the class *BrowserControl*. This is an embedded class whose task is to launch the appropriate browser application (Netscape under Linux, and the default browser under Windows). The latter two methods provide a hyperlink-like feel by displaying an underlining when the mouse moves over the web address (note that this is necessary, because the web address is just a plain label).

A.2.2 Search Algorithms

In what follows there is an itemized list of the classes that implement the search algorithms. The documentation of the classes comprises the explanation of the methods, remarks about their role and all the implementation notes that I have found important, respectively.

The constructors of the classes mainly serve for parameter passing, namely they set the private variables of the class according to the ones got by parameter. They won't be mentioned in case of the certain classes separately, only if they have something extra role.

A.2.2.1 Backtrack Search Algorithms

A.2.2.1.1 Class BackTrackMethods

All the three kinds of the backtrack search algorithm (*ApmBackTrack*, *PpmBackTrack* and *SpmBackTrack*) extend the abstract class *BackTrackMethods*. The reasons for using inheritance:

i) "Reusing of code", namely the class *BackTrackMethods* provides some common methods linked with the problem context, which are necessary for all the three algorithms. The process and the implementation of the search are very similar in all the three of the cases and the only difference is the handling of periodicity. Hence only those methods are not implemented by this class, which are concerned in the matter of periodicity.

ii) The launching of the algorithms is carried out by a separate thread. The type of the algorithm is passed by parameter to the thread, so if there is some kind of relationship between the types of the algorithms, a common ancestor, for instance, then the same thread object can be used in every case.

Global variables

`byte alphabet`	The alphabet size.
`int[] windowSize`	The dimension of the window. It is an array with two elements, the first one stands for *m* and the second one for *n*.
`int[] mapSize`	The dimension of the map. It is an array with two elements, the first one stands for *R* and the second one for *S*.
`boolean findAll`	The value of this boolean variable indicates whether to find all the possible maps.
`byte[] possibleWindows`	It is an array to store all the possible windows, it has the length of k^{mn}. The indices of the array stand for the decimal value of a certain window. So the windows need not to be stored actually, there is a conversion function instead that converts a window into a decimal value if needed. The elements of the array indicate that the corresponding window is used (1) or free (0).
`public boolean stopped`	It is a public variable to provide the user interface the possibility to control the running of the algorithm. The algorithm checks the value of this variable systematically, and if it is false, the search terminates.

Methods

`public BackTrackMethods( byte alphabet, int windowSizeM, int windowSizeN, int mapSizeR, int mapSizeS, boolean findAll)`	The constructor. It sets the private variables of the class according to the ones got by parameter, initializes the *possibleWindows* array, and sets the value of the public field *stopped* to false.
`protected void`	It fills the given map with the given element.

`fill(`
`  byte[][] map,`
`  byte element)`	

`private int`
`encode(`
`  byte[] what,`
`  int alphabet)`
	A private function that converts a one-dimensional alphabet-ary array into a decimal value.

`protected int`
`encode(`
`  byte[][] what)`
	It encodes a two-dimensional alphabet-ary array into a decimal value as follows. First it maps the two-dimensional array into a one-dimensional one, then invokes the private *encode()* function.

The conversion is carried out in row-major order (see Figure 0-5). It is an unambiguous operation, so there is a one-to-one correspondence between the two-dimensional and the one-dimensional arrays.

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} \rightarrow 6$$

Figure 0-5 The encoding

`protected byte`
`pickElement(`
`  byte[] triedAlready)`
	It picks an element from the alphabet, which is not in the array *triedAlready*. This array is based on the same idea as *possibleWindows*, so this method inspects whether it has a zero element in the appropriate position.

`protected abstract boolean`
`copy(`
`  byte[][] candidate,`
`  byte newElement,`
`  int level)`
	This is an abstract method to be overridden in the descending classes, because this method is concerned in the matter of periodicity, so it cannot be applied universally.

`protected abstract void delete(`
`  byte[][] candidate,`
`  int level)`
	Likewise the previous method, it is an abstract method, as well. The reason is the same, namely it is concerned in the matter of periodicity, and so the deletion of an element should be treated differently in the descendants according to the periodicity.

`protected byte[][]`
`createWindow(`
`  byte[][] candidate,`
`  int row,`
`  int column)`
	It creates a window at the specified position in such a way that the specified position should fell on the left upper corner of the window. In most of the cases it is used such a way that it is invoked with actual parameters that demand the newly inserted element to fall on the right bottom corner of the window as illustrated in the figure below.

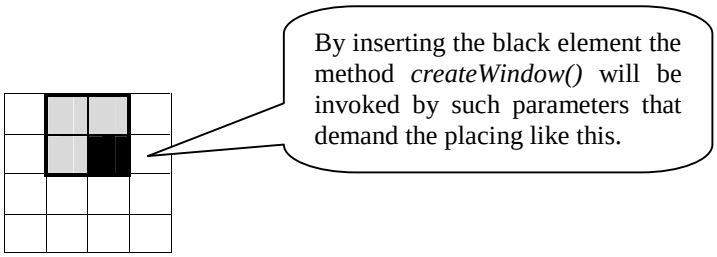


Figure 0-6 Sketch of a $(4,4;2,2)_2$ - Periodic Perfect Map

`private static boolean`
`areTheyTheSame(`
`  byte[][] array1,`
`  byte[][] array2,`
`  int kindOfPeriodicity)`
	Inspects whether the two arrays are equal according to the kind of periodicity. It is a static method invoked by the method *isItInPMAlready()*, so it cannot be implemented as an abstract method, however, it would be the most obvious solution.
	
	The arrays should be compared taking into account that their shifted equivalent should differ, as well. In the semi-periodic case this shifting needs to be considered in one dimension, and in the periodic case in both dimensions.

`private static boolean`
`isItInPMAlready(`
`  byte[][] pm,`
`  Vector PM,`
`  int kindOfPeriodicity)`
	In the aperiodic case it simply returns with false (note that the found maps are bound to be different), otherwise it inspects whether the given map differs from the previously found ones that are stored in the Vector *PM*. It is a static method invoked by the method *start()* if a new Perfect Map is found.

`protected byte[][]`
`clone(`
`  byte[][] spm)`
	It creates a deep copy of the specified map. If we are about to find all the possible maps, it is important to check whether it differs from the previously found ones. To make this comparison possible we need to store the maps found so far.

`protected void`
`printMap(`
`  byte[][] apm,`
`  PrintStream pout)`
	It prints the given map to the given output.

`protected void`
`start(`
`  PrintStream pout,`
`  int kindOfPeriodicity)`
`  throws`
`  OutOfMemoryError`
	It performs the search as follows. After having the appropriate assistant variables and arrays initialized, comes next the loop of the backtrack search. This search could be implemented as a recursive one, but I used the *break-continue* technique instead. It is more applicable in case of deep search trees, where the recursive one may get stuck because of stack overflow error.

Every step of the search corresponds to a level of the search tree, namely a position in the candidate. In every step we choose a non-tried element from the possible ones, put it into the appropriate array of *triedAlready*, and try to fit it into the candidate. If it is a legal operation (i.e. the *copy()* returns with true), then we step to the next level, else backtrack to the previous one. If there is no more non-tried element, also a backtracking is needed, but it is made from a ramification of the search tree this time, so the appropriate array of *triedAlready* needs to be reset.

The user interface is informed about the actual state of the search by setting the proper variables of the class *PerfectMapGenerator*. It means that the control of some of the GUI components is managed within this method.

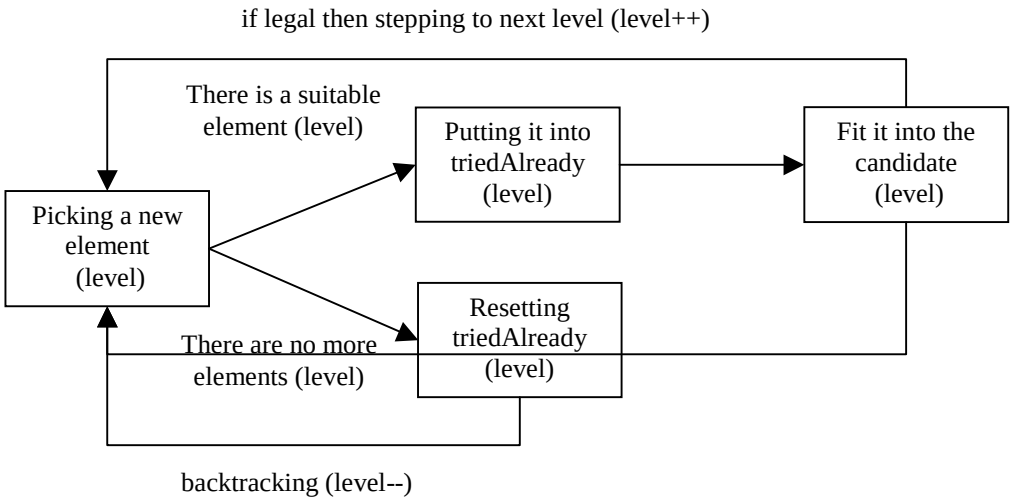


Figure 0-7 Flowchart of the backtrack search

A.2.2.1.2 Class ApmBackTrack

This class extends the class *BackTrackMethods* and implements its two abstract methods according to the aperiodic case.

Methods

`protected boolean`
`copy``(`
`  byte[][] candidate,`
`  byte newElement,`
`  int level)`
	Inserts an element into the candidate as follows. The corresponding row and column can be gained from the level (the quotient of the level and *n* yields the row, while the remainder yields the column). After having the element inserted, the raising window needs to be inspected whether it is a legal one. If it is legal the method returns with true, otherwise we let the copy undone and return with false. Note that there are cases when there is no "arising window" (see the semantics of the method *createWindow()* in the class *BackTrackMethods*), these cases are treated as legal.

`protected void`
`delete``(`
`  byte[][] candidate,`
`  int level)`
	Deletes an element determined by the level from the candidate. The window determined by that element needs to be deleted from the array *possibleWindows*, as well.

A.2.2.1.3 Class SpmBackTrack

This class extends the class *BackTrackMethods* and implements its two abstract methods according to the semi-periodic case.

Methods

`protected boolean`
`copy``(`
`  byte[][] candidate,`
`  byte newElement,`
`  int level)`
	The implementation of this method is the same as in the aperiodic case, but there is an additional piece of code in this case that checks the periodicity if needed (if we have all the elements in a row, namely we are in the last column). It is accomplished by means of a rollback-technique, namely all the arising windows are stored in a rollback array, so when checking the windows, not only the *possibleWindows* should be inspected, but the rollback array, as well. If the insertion was legal, the content of the rollback array is copied into the array *possibleWindows*.

`protected void`
`delete``(`
`  byte[][] candidate,`
`  int level)`
	The principle of the deletion is the same as in the aperiodic case, but here we have to be careful by deleting the elements that influence the periodicity, namely the last elements of the columns. In this case all the concerned windows should be deleted from the array *possibleWindows*.

The following figure shows the situation, where the insertion/deletion of an element in the last column (indicated by black) influences not only one window, but two (indicated by grey).

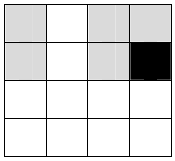


Figure 0-8 Sketch of a $(4,4;2,2)_2$ - Semi-Periodic Perfect Map

A.2.2.1.4 Class PpmBackTrack

This class extends the class *BackTrackMethods* and implements its two abstract methods according to the periodic case.

Methods

`protected boolean`
`copy``(`
`  byte[][] candidate,`
`  byte newElement,`
`  int level)`
	The implementation of this method is the same as in the semi-periodic case, with the difference that here not only the last elements of the rows should be treated carefully, but the last element of the columns, as well. Note that the windows arisen by inserting the right bottom element need to be treated separately because of the semantics of the method *createWindow()* in the class *BackTrackMethods*.

`protected void`
`delete``(`
`  byte[][] candidate,`
`  int level)`
	The principle of the deletion is the same as in the semi-periodic case, but here the last elements of the rows and also those of the columns need to be considered with all the collateral windows.

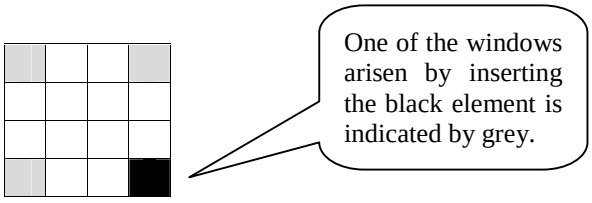


Figure 0-9 Sketch of a $(4,4;2,2)_2$ - Semi-Periodic Perfect Map

A.2.2.2 Genetic Algorithms

The different operators of the genetic algorithm are implemented in different classes (*ParentSelection.java*, *Recombination.java*, *Mutation.java* and *SurvivorSelection.java*). Apart from the operators, the mechanism of initialization and evaluation is implemented separately, as well (*Evaluation.java* and *Initialization.java*, respectively). All of these classes provide public functions, each performing a different realization of the operator in point.

The mechanism of the evolution is implemented in the class *GAMethods* through the method *start()*, hence the declarations of the operators should be done in this class, but the actual instantiating is carried out in a different class according to the periodicity (*ApmGA*, *PpmGA* and *SpmGA)*. Note that the kind of periodicity does not have any influence on the certain operators, only the manner of evaluation differs.

When considering a genetic algorithm in terms of performance, a very important factor is the number of fitness evaluations. It is an expensive operation, so our aim is to reduce its number to such an extent as possible. When applying the certain operators, we have to take care also of the fitness values that we have already, and try to avoid the extra evaluations.

A.2.2.2.1 Package util

This package contains two classes, each providing methods that can be useful in a particular unit of the genetic algorithm. This collecting of the common methods has many advantages: "reusing of code", namely the methods need not be defined and implemented multiple times in separate units, only in this package instead. This organization provides a clearly arranged code and the possibility of simple modification.

A.2.2.2.1.1 Class CommonMethods

This class contains all the problem-specific methods that can be useful for any classes of the genetic algorithm.

`public byte[][] createPhenoType( byte[] individual)`	It maps the individual into the phenotype space, namely it creates a matrix of proper size from its genotype. This method is used for: • evaluation (class *Evaluation*) – because the evaluation is performed in the phenotype space • survivor selection (class *SurvivorSelection*) – because the individuals need to be checked whether they are perfect ones • evolution (class *GAMethods*) – in case if the population is printed to the output file

A.2.2.2.1.2 Class SelectionMethods

This class contains all the methods that can be useful for both of the selection operators (*ParentSelection* and *SurvivorSelection*).

The first two of the following methods take part in ranking the population according to the fitness values.

`public int searchMax( int[] array)`	It is used to determine the maximum element of the given array. This element will be passed to the method *searchMin()*.
`public int searchMin( int[] array int max)`	Returns the position of the minimum element. The parameter *max* stands for the maximum element at the beginning - note that the content of the array changes, every newly found minimum value will be replaced by (max + 1). This replacement guarantees that we will find the correct values in ascendant order.
`public int searchMinPos( int[] array)`	It returns the position of the minimum element. This method is used in course of tournament selection.

A.2.2.2.2 Class GAMethods

All the three kinds of the genetic algorithm (*ApmGA*, *PpmGA* and *SpmGA*) extend the abstract class *GAMethods*. The reasons for using inheritance are the same as in case of backtrack search.

Global variables

The variables `int alphabet`, `int[] windowSize`, `int[] mapSize` and `boolean stopped` have the same meaning as in case of backtrack search algorithm.

`PrintStream pout`	If the value of the *printToFile* variable is true, the populations will be printed to this output (specified in the class *PerfectMapGenerator* according to the actual parameters of the map).
`boolean printToFile`	It is a boolean variable indicating whether to print the populations to the output file.

The classes representing the operators are declared also as global variables, their public functions are invoked in the method *start()*, which performs the evolution.

The following variables determine the behaviour and the quality of the genetic algorithm, namely their values indicate which operator to use in the course of the evolution. They represent the user's choice made in the *GADialog* dialog window, and according to their values the appropriate operators will be applied:

parentSelection (1: ranking selection, 2: fitness proportional selection, 3: tournament selection), *tournamentSelection* (used in parent selection; 1: deterministic, 2: stochastic),

tournamentSelection2 (used in survivor selection; 1: deterministic, 2: stochastic), *recombination* (1: uniform crossover, 2: n-point crossover), mutation (1: random resetting, 2: creep mutation) and *survivorSelection* (1: ranking selection, 2: fitness proportional selection, 3: tournament selection, 4: best from union, 5: replace worst).

Methods

`public GAMethods(···)`	This constructor has many arguments that deliver the user's choice to the algorithms. These choices are "filtered" already, because they had not been sent to this class directly, but to its descendants, who passes only the necessary ones on to the certain classes representing the operators and to the ancestor class, respectively. Besides the parameter passing it has another task, as well. It initializes the value of the public field *stopped* to false.
`private double average( int[] array)`	It computes the average of the values in the given array.
`private void printPopulation( Vector population, int[] survivorFitness, PrintStream pout)`	It prints all the individuals of the given population including the corresponding fitness values to the given output.
`protected void start()`	It performs the evolution, namely applies the appropriate operators in the correct order as illustrated in the figure below.

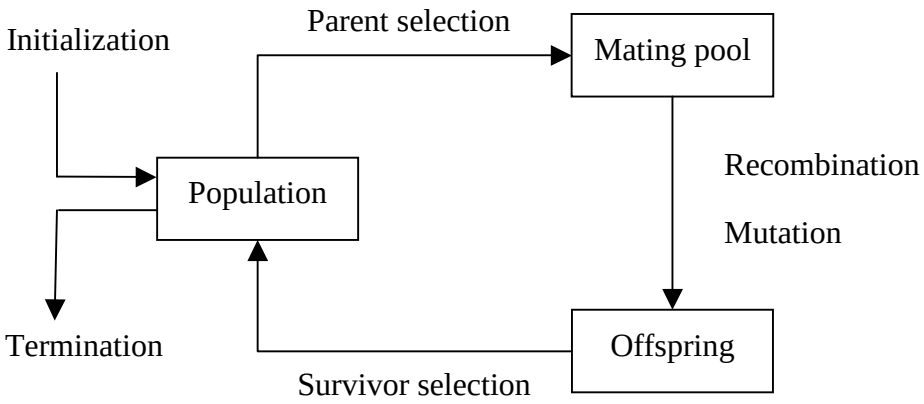


Figure 0-10 Flowchart of the evolution

A.2.2.2.3 Class ApmGA, PpmGA and SpmGA

These three classes are almost the same. They consist merely of a constructor, whose task is to instantiate the classes that represent the operators. The constructor has a lot of parameters, these are the values that the user had chosen in the *GADialog* dialog window. That is to say these values determine the behaviour of the genetic algorithm.

First, it invokes the constructor of the ancestor class with the actual parameters that are necessary there – i.e. only the parameters of the map, no operator-specific ones. Then it initializes a *Random* object and passes it to the classes that need some kind of random function. As well as every class is provided with the parameters, which are necessary to the proper functioning of the operator represented by the class.

As I already mentioned above, the kind of periodicity does not have any influence on the certain operators, only the manner of evaluation differs. Hence the only issue that differs in case of the three classes is the instantiation of the appropriate descendant (*ApmEvaluation*, *SpmEvaluation* and *PpmEvaluation*) of the class *Evaluation*.

A.2.2.2.4 Class Initialization

This class performs a random initialization by means of its only one public function *initialize()*.

A.2.2.2.5 Class ParentSelection

The constructor of the class serves only for parameter passing, namely it sets the private variables of the class according to the ones got by parameter.

The methods in class *SelectionMethods* (in package *util*) – complemented with the following method – make the implementation of the certain selection operators easier.

```
private Vector
rank(
   Vector offspring,
   int[] fitnessValues)
```

It ranks the population according to the fitness values. The fittest individual has the lowest rank. It is realized by searching the array of fitness values for the minimum value through the method *searchMin()*. The minimum fitness value and the corresponding individual will be put in the ranked population and in the array of ranked fitness values, respectively. The minimum value will be replaced with the increased maximum value, hence making the search for the next minimum value possible.

There are three kinds of parent selection operators implemented in this class:

- ranking selection - rankingSelection()
- fitness proportional selection - *FPSelection()*
- tournament selection - *tournamentSelection()*

For determining the selection probabilities I implemented two algorithms:

- roulette wheel algorithm - *rouletteWheel()*
- stochastic universal sampling algorithm - *SUS()*

The implementation of the above operators and that of the ones in the following two classes are not detailed here, since it is consistent with their specification (see section 4.1.2.2).

A.2.2.2.6 Class Recombination

This class implements two kinds of recombination operators:

- uniform crossover - *uniformCrossover()*
- n-point crossover - *nPointCrossover()*

A.2.2.2.7 Class Mutation

This class implements two kinds of mutation operators:

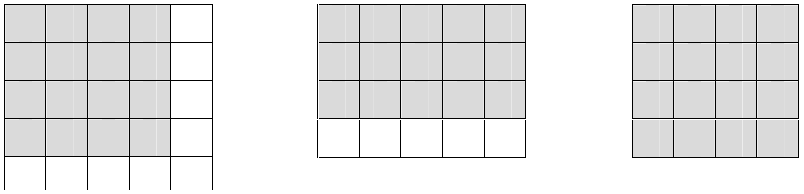
- random resetting - *randomReset()*.
- creep mutation – *creepMutation()*

A.2.2.2.8 Class Evaluation

`protected byte[][] createWindow( byte[][] candidate, int row, int column)`	It creates a window at the specified position in such a way that the specified position should fell on the left upper corner of the window.
`protected boolean equals( byte[][] array1, byte[][] array2)`	Compares two two-dimensional arrays for equality. The two arrays are defined to be equal if they contain the same elements in the same order.
`protected abstract int fitnessValue( byte[] individual)`	Computes the fitness value of an individual. It is implemented according to the kind of periodicity.
`protected int[] evaluate( Vector offspring)`	Evaluates a population, namely computes the fitness value of every individual by means of the method *fitnessValue()*. It returns the array of fitness values. The link between the population and this array are formed by the indices, namely an element of this array corresponds to the individual with the same index in the population.

A.2.2.2.9 Class ApmEvaluation, PpmEvaluation, SpmEvaluation

All the three of these classes extend the class *Evaluation*. This inheritance is needed because the evaluation should be treated differently according to the kind of periodicity. They implement the abstract method *fitnessValue()* of the ancestor class. The implementation is very similar in the three cases: it inspects the positions in the map and computes the ranks of the certain windows (for the exact specification of the evaluation function see the specification of the genetic algorithm in Section 4.2.2). As for the difference, the set of positions to inspect (indicated by grey) differs in the three cases:



Aperiodic case Semi-Periodic case Periodic case

Figure 0-11 Sketch of Perfect Map with two-by-two window

A.2.2.2.10 Class SurvivorSelection

This class re-implements all the methods of the class *ParentSelection* with a little modification. While in case of parent selection it was not necessary to take care of the fitness values, in case of survivor selection this is essential. The reason for this is that the user is informed about the fitness average in course of the evolution. It is possible only if keep the fitness values during the survivor selection. Hence all the methods are re-implemented in such a way that they return not only the population, but the corresponding fitness values, as well. These methods are not itemized here.

`public`
`SurvivorSelection(`
`  byte alphabet,`
`  int mapSizeR,`
`  int mapSizeS,`
`  int populationSize,`
`  Random rand,`
`  PrintStream pout,`
`  double s2,`
`  int k2,`
`  boolean replacement2,`
`  int probMethod2)`
	It invokes the constructor of the ancestor class and passes almost all of its parameters to it, except *pout*, because it is used only if a map needs to be printed to the output file. This occurs only in case if the population is looked for Perfect Maps, which mechanism is settled within the confines of the survivor selection by means of the *checkPM()* method.

`private Vector`
`unite(`
`  Vector offspring,`
`  int[] fitnessValues,`
`  Vector population,`
`  int[] populationFitness)`
	Unites the offspring and the population. The reason for this is that almost all of the selection methods apply to the union of the population and the offspring. It takes care also of the fitness values by uniting them accordingly. It returns with a vector of size two, whose first element is the united population and the second is the array of united fitness values.

`private boolean`
`checkPM(`
`Vector offspring,`
`int[] fitnessValues,`
`Vector population,`
`int[] populationFitness)`
	Checks whether there is a Perfect Map in the population or in the offspring. It is realized by examining both arrays of fitness values, and if a zero value is found, the corresponding individual is mapped into the phenotype space through the method *createPhenoType()* of the class *CommonMethods* (in package *util*), and the arising phenotype is written to the output file.

`private void`
`printMap(`
`byte[][] map,`
`PrintStream pout)`
	It prints the given map to the given output.

The following selection methods have three issues in common:

i) First it is needed to check whether the population or the offspring contains a Perfect Map (by means of the *checkPM()* method). If yes, the methods return a null value indicating that the search should be terminated. The *checkPM()* method takes care also of the printing of the found maps to the output file.

ii) Except the method *replaceWorst()*, all of them apply to the union of the offspring and the population.

iii) They return a vector of size two, whose first element is the survivor population and the second one is the array of their fitness values.

`public Vector`
`theVeryBest(`
`Vector offspring,`
`int[] fitnessValues,`
`Vector population,`
`int[] populationFitness)`
	Selects the best individuals based on their fitness. It is realized by first ranking the population based on fitness by means of the method *rank()*, then selecting the required number of individuals from the front part of the ranked list. Their number equals to the user defined population size.

`public Vector`
`fitnessBased(`
`Vector offspring,`
`int[] fitnessValues,`
`Vector population,`
`int[] populationFitness,`
`int whichMethod)`
	Performs a fitness-based selection by invoking the method *FPSelection()*. The parameter *whichMethod* stands for the option, which algorithm to use for determining the selection probabilities (1: roulette wheel algorithm, 2: stochastic universal sampling).

`public Vector`
`rankBased(`
`Vector offspring,`
`int[] fitnessValues,`
`Vector population,`
`int[] populationFitness,`
`int whichMethod)`
	Performs a rank-based selection by invoking the method *rankingSelection()*. The parameter *whichMethod* stands for the option, which algorithm to use for determining the selection probabilities (1: roulette wheel algorithm, 2: stochastic universal sampling).

`public Vector`
`tournamentBased(`
`  Vector offspring, int[]`
`  fitnessValues,`
`  Vector population,`
`  int[] populationFitness)`
	Performs a tournament selection by invoking the method *tournamentSelection()*. The size of the tournament (*k2*) and the replacement option (*replacement2*) (whether to put back the winner into the tournament pool) are passed by parameter to the constructor of the class.

`public Vector`
`replaceWorst(`
`  Vector offspring,`
`  int[] fitnessValues,`
`  Vector population,`
`  int[] populationFitness)`
	Its task is to replace the worst individuals in the population with the newly created offspring. It is realized by adding the occurrent “good” individuals from the population (their number equals to the difference of the population size and the offspring size) and the offspring to the survivor pool.

Appendix B

List of the Referenced Programs

All the referenced programs and their source codes are available via the web address http://juditk.web.elte.hu/msc/.

The below list follows the directory structure of the above web address.

	Description of the content and the corresponding section	
[1 Dimension]		
[DbcBackTrack]	One-dimensional reference algorithm.	4.1.1.1
[DbcBackTrack_parallel]	The parallelization of the one-dimensional reference algorithm.	4.1.1.1
[DbcGA]	One-dimensional genetic algorithm.	4.1.2.2
[DbcGA_parallel]	The parallelization of the one-dimensional genetic algorithm.	4.1.1.2.6
[DBGraph]	De Bruijn Graph generator.	4.2.6.2
[2 Dimensions]	Perfect Map Generator software (two-dimensional reference algorithm and genetic algorithm)	4.1.2.1, 4.1.2.2, Appendix A
[doc]	The API specification of the Perfect Map Generator software.	
[solutions]	The output files created during testing.	
[3 Dimensions]	Three-dimensional genetic algorithm.	4.1.3.2
[4 Dimensions]	Four-dimensional genetic algorithm.	4.1.3.3

Bibliography

[1] C. Flye-Sainte Marie: *Solution to problem number 58.* l'Intermediaire des Mathematiciens **1** (1894), 107–110.

[2] N. G. de Bruijn: *A Combinatorial Problem.* Nederlandse Koninklijke Academie van Wetenschappen **49** (1946), 758–764.

[3] I. J. Good: *Normally Recurring Decimals.* Journal of the London Mathematical Society **21** (1946), 167–169.

[4] H. Fredricksen: *A Survey of Full Length Nonlinear Shift Register Cycle Algorithms.* SIAM Review **24** (1982), 195–2.

[5] J. Bondy and U. Murty: *Graph Theory with Applications.* Amsterdam, Elsevier, 1976.

[6] E. Petriu: *Absolute-type Pseudorandom Shift Encoder with Any Desired Resolution.* IEE Electronics Letters **21** (1985), 215–216.

[7] E. Petriu: *Absolute-type Position Transducers Using a Pseudorandom Encoding.* IEEE Transactions on Instrumentation and Measurement **36** (1987), 950–955.

[8] E. Petriu: *New Pseudorandom/Natural Code Conversion Method.* IEE Electronics Letters **24** (1988), 1358–1359.

[9] E. Petriu: *Scanning Method for Absolute Pseudorandom Position Encoders.* IEE Electronics Letters **24** (1988), 1236–1237.

[10] J. Basran and E. Petriu: *On the Position Measurement of Automated Guided Vehicles Using Pseudorandom Encoding.* IEEE Transactions on Instrumentation and Measurement **38** (1989), 799–803.

[11] J. Basran, F. Groen and E. Petriu: *Automated Guided Vehicle Position Recovery.* IEEE Transactions on Instrumentation and Measurement **39** (1990), 254–258.

[12] B. Arazi: *Position Recovery Using Binary Sequences.* IEE Electronics Letters **20** (1984), 61–62.

[13] H. M. Fredricksen: *Generation of the Ford Sequence of Length 2^n, n Large.* Journal of Combinatorial Theory (A) **12** (1972), 153–154.

[14] H. M. Fredricksen: *A Class of Nonlinear de Bruijn Cycles.* Journal of Combinatorial Theory (A) **19** (1975), 192–199.

[15] H. M. Fredricksen and I. J. Kessler: *Lexicographic Compositions and de Bruijn Sequences.* Journal of Combinatorial Theory (A) **22** (1977), 17–30.

[16] H. M. Fredricksen: *The Lexicographically Least de Bruijn Cycle.* Journal of Combinatorial Theory **9** (1970), 1–5.

[17] G. Hurlbert: *Universal Cycles–On Beyond de Bruijn.* Ph.D. Thesis, 1990.

[18] C. J. Mitchell, T. Etzion and K. G. Paterson: *A method for constructing decodable de Bruijn Sequences.* IEEE Transactions on Information Theory **42** (1996), 1472–1478.

[19] N. Vörös: *On the complexity of symbol sequences.* Conference of Young Programmers and Mathematicians, Budapest, 1984, 43–50.

[20] M. Horváth and A. Iványi: *Growing Perfect Cubes.* Submitted.

[21] M. Horváth and A. Iványi: *Perfect Sequences.* International Conference of Applied Informatics, Eger, 2004. Submitted.

[22] T. Etzion: *Constructions for Perfect Maps and Pseudorandom Arrays.* IEEE Transactions on Information Theory **34** (1988), 1308–1316.

[23] C. J. Mitchell: *Constructing c-ary Perfect Factors.* Designs, Codes and Cryptography **4** (1994), 341–368.

[24] C. J. Mitchell: *New c-ary Perfect Factors in the de Bruijn Graph.* In P. G. Farrell, ed., "Codes and Cyphers – Cryptography and Coding IV", IMA Press, Southend-On-Sea, Essex, 1995.

[25] C. J. Mitchell: *De Bruijn Sequences and Perfect Factors.* SIAM Journal on Discrete Mathematics **10** (1997), 270–281.

[26] K. G. Paterson: *Perfect Factors in the de Bruijn Graph.* Designs, Codes and Cryptography **5** (1995), 115–138.

[27] C. J. Mitchell and K. G. Paterson: *Perfect Factors from Cyclic Codes and Interleaving.* SIAM Journal on Discrete Mathematics **11** (1998), 241–264.

[28] C. W. Marshall: *Applied Graph Theory.* Wiley-Interscience, New York, 1971.

[29] F. W. Sinden: *Sliding Window Codes.* AT&T Bell Labs. Tech. Memo, 1985.

[30] J. Burns and C. Mitchell: *Coding Schemes for Two-dimensional Position Sensing.* Cryptography and Coding III, M. Ganley, Ed. London, UK: Oxford Univ. Press, 1993, 31–66.

[31] F. J. MacWilliams and N. J. A. Sloane: *Pseudorandom Sequences and Arrays.* Proceedings of the IEEE **64** (1976), 1715–1729.

[32] A. Fukunda, H. Imai, H. Miyakawa and T. Nomura: *A Theory of Two-dimensional Linear Recurring Arrays.* IEEE Transactions on Information Theory **18** (1972), 775–785.

[33] J. Dénes and A. Keedwell: *A New Construction of Two-dimensional Arrays with the Window Property.* IEEE Transactions on Information Theory **36** (1990), 873–876.

[34] M. Harwit: *Spectrometer Imager.* Applied Optics **10** (1971), 1415–1421.

[35] J. H. van Lint, E. J. MacWilliams and N. J. A. Sloane: *On Pseudorandom Arrays.* SIAM Journal of Applied Mathematics **36** (1979), 62–72.

[36] B. Grünbaum and G. C. Shephard: Satins and Twills: *An Introduction to the Geometry of Fabrics.* Mathematics Magazine **53** (1980), 139–161.

[37] S. L. Ma: *A Note on Binary Arrays with a Certain Window Property.* IEEE Transactions on Information Theory **30** (1984), 774–775.

[38] J. C. Cock: *Toroidal Tilings from de Bruijn-Good Cyclic Sequences.* Discrete Mathematics **70** (1988), 209–210.

[39] K. Dehnhart and H. Harborth: *Universal Tilings of the Plane by 0-1 Matrices.* Discrete Mathematics **73** (1988/89), 65–70.

[40] G. Hurlbert and G. Isaak: *On the de Bruijn Torus Problem.* Journal of Combinatorial Theory (A) **64** (1993), 50–62.

[41] C. J. Mitchell: Aperiodic and Semi-Periodic Perfect Maps. IEEE Transactions on Information Theory **41** (1995), 88–95.

[42] K. G. Paterson: *Perfect Maps.* IEEE Transactions on Information Theory **40** (1994), 743–753.

[43] K. G. Paterson: *New Classes of Perfect Maps I.* Journal of Combinatorial Theory (A) **73** (1996), 302–334.

[44] G. Hurlbert and G. Isaak: *New constructions for de Bruijn Tori.* Designs, Codes and Cryptography **6** (1995), 47–56.

[45] G. Hurlbert and G. Isaak: *A Meshing Technique for de Bruijn Tori.* Contemporary Mathematics **178** (1994), 153–160.

[46] K. G. Paterson: *New Classes of Perfect Maps II.* Journal of Combinatorial Theory (A) **73** (1996), 335–345.

[47] G. Hurlbert, C. J. Mitchell and K. G. Paterson: *On the Existence of de Bruijn Tori with Two by Two Windows.* Journal of Combinatorial Theory (A) **76** (1996), 213–230.

[48] F. R. K. Chung, P. Diaconis and R. L. Graham: *Universal Cycles for Combinatorial Structures.* Discrete Mathematics **110** (1992), 43–59.

[49] C. T. Fan, S. M. Fan, S. L. Ma and M. K. Siu: *On de Bruijn Arrays.* Ars Combinatoria **19A** (1985), 205–213.

[50] C. J. Mitchell and K. G. Paterson: *Decoding Perfect Maps.* Design, Codes and Cryptography **4** (1994), 11–30.

[51] G. Hurlbert and G. Isaak: *On Higher Dimensional Perfect Factors.* Ars Combinatoria **45** (1997), 229–239.

[52] A. E. Eiben and J. E. Smith: *Introduction to Evolutionary Computing.* Springer-Verlag, Berlin, 2003.