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:	

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

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

gusz@cs.vu.nl

Co-Supervisor

Prof. Dr. Antal Iványi

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

tony@inf.elte.hu

Second Reader

Content

1. Introduction

Perfect Maps (or *de Bruijn Tori*), namely, two-dimensional arrays in which every possible rectangular sub-array (of fixed size) occurs precisely once, have been studied for a long time and a vast literature exists concerning them (see, for example, [1]). A number of construction methods have been devised [2, 3, 4], but the existence question has not been completely answered.

...

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.*

$$[0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1]$$

Figure 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 we have two cycles:* $[0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1]$ *(found clockwise) and* $[0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1]$ *(found counter-clockwise).*

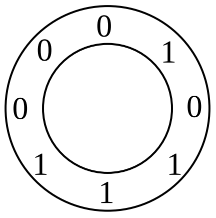

Figure 2

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 [5], and rediscovered in 1946 by de Bruijn [6] and Good [7]. An excellent survey on the topic by Fredricksen can be found in [8].

De Bruijn Cycles have applications in the study of position-detection [9, 10, 11, 12, 13, 14, 15, 16], pseudorandom numbers, cryptography, nonlinear shift registers and coding theory, and a vast literature exists [17, 18, 19, 20, 21].

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 [9, 33, 12] 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 [22].

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 1986, Cummings and Wiedemann [23] 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 $\bar{k} = \langle k_1 k_2 \ldots \rangle$ 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 [47] in 1994 constructed a Growing Sequence for the case when $\bar{k}$ is the sequence of the even number. Then years later, Iványi and Horváth [25] 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 [25] the following

THEOREM 5 *If $n \geq 1$ and $\bar{k} = \langle k_1 k_2 \ldots \rangle$ 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 [24].

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 [25].

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 [4] and later by Hurlbert and Isaak [42] as *Equivalence-Class de Bruijn Cycles*. Perfect Factors have proved useful in constructions for de Bruijn Tori and have been extensively studied in [26, 27, 28, 29].

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.*

$$\begin{Bmatrix} [0 & 0 & 0 & 1 & 2 & 2 & 1 & 2 & 1], \\ [1 & 1 & 1 & 2 & 0 & 0 & 2 & 0 & 2], \\ [2 & 2 & 2 & 0 & 1 & 1 & 0 & 1 & 0] \end{Bmatrix}$$

Figure 1 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 [28].

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

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

ii) $n < R \leq k^n$ *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 [4] has shown that the conjecture holds in the binary case. This was extended to cases where k is a prime power by Paterson [29]. Mitchell and Paterson [30] 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 [29] that the conditions of Lemma 3 are sufficient when $p_i^{s_i} > n$ for every index i. In [26] 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 $n < 5$ [27] and that Perfect Factors exist for all triples $(R,6,k^6/R)_k$ satisfying the conditions of Lemma 3 with some possible exceptions [26]. He obtained Perfect Factors for some of those exceptions in [28].

This result was extended to $n < 7$ in [30]. Regarding the cases $n = 7$ and 8, unresolved parameter sets and remarks can be found in [30].

Mitchell [27] has shown that the following Perfect Factors exists:

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

2.1.5. Perfect Multi-Factors

Mitchell introduced two auxiliary classes of combinatorial objects: *Perfect Multi-Factors* [26] and *Generalized Perfect Factors* [27] (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 $k \ge 2$. 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 $0 \le j < m$, τ occurs at a position $p \equiv$ j (mod m) in one of these sequences.*

$$\begin{Bmatrix} [0 & 0 & 0 & 0 & 1 & 1 & 1 & 1], \\ [1 & 0 & 0 & 0 & 0 & 1 & 1 & 1], \\ [0 & 1 & 0 & 1 & 1 & 0 & 1 & 0], \\ [0 & 0 & 1 & 0 & 1 & 1 & 0 & 1] \end{Bmatrix}$$

Figure 1 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 [26].

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 [26] 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 Rm 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 [27].

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 [27], but there are some (constructive) existence results for Generalized Perfect Factors in [27, 30].

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 $< x_2 x_3 \ldots x_{n+1} >$.*

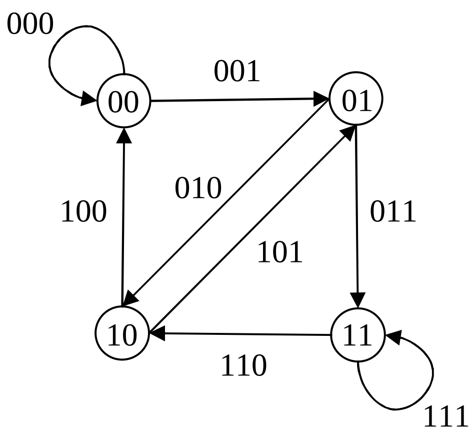

Figure 1 A $(2,2)$ - de Bruijn Graph

Since a (k,n) - de Bruijn Graph is connected and each vertex has k ingoing and k outgoing edges, it has an Euler path [31]. 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 [31]. 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 4.1.3.2.

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 1 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 [43], 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".

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 [43].

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, (R,S) the period and (m,n) the order of the torus.*

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

Figure 1 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 [32, 33], pseudorandom arrays [4, 34, 35, 36], and the design of mask configurations for spectrometers [37]. (For an interesting variation on this theme see [38]). Even cloth patterns have used these designs, long before their mathematical properties were discovered [39].

In 1984, Ma [2] proved the binary case of the following 1988 theorem of Cock [40] (see also [41]).

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 [42] and by Mitchell in [43].

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 [44] 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 [45] 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 [46] 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 [47] they proved a sub-case (see Theorem 1 in Section 2.2.2.4.) with the hope that it will help extend this result.

The most progress toward the previous conjecture by Paterson [48] 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* $0 \leq r_i \leq k_i mn$ *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 [49] to examine the parameter sets where $p_i^{r_i} \leq m$ for some indices and $p_i^{s_i} \leq n$ 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 [50] 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 [3], 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 [42] 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 [48] 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* $m > n \geq 2$*. Then there exists an* $(m^4, n^4; 2,2)_{mn}$ *- de Bruijn Torus.*

THEOREM 3 *Suppose* $n > 2$ *is odd. Then for every* $k \geq 1$*, 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 1 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 [43], 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 [33, 51]. Decoding means a method for computing the position of a given sub-array within a Perfect Map. In [51] 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 [43].

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* $x=1,2,\ldots$.

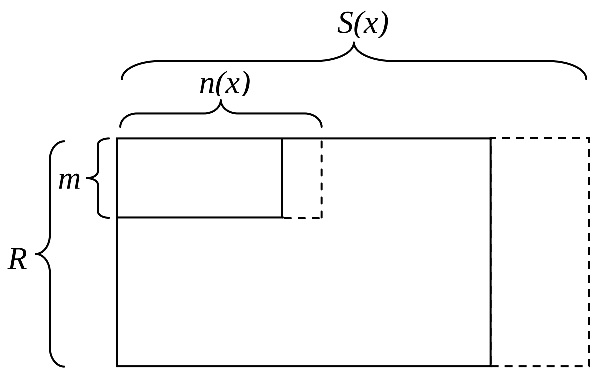

Figure 1 Sketch of an Increasing Perfect Map

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.*

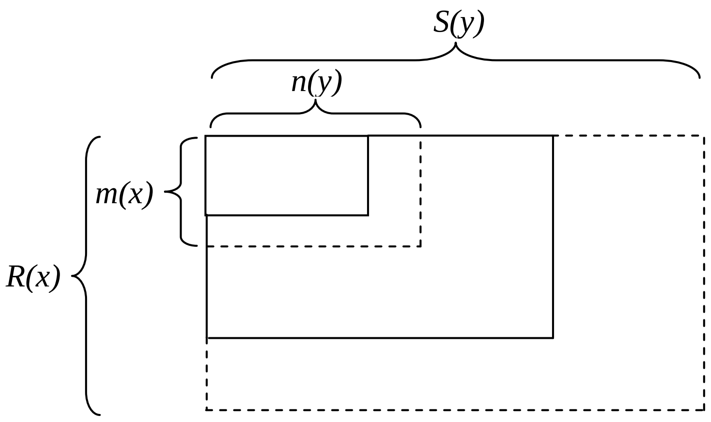

Figure 1 Sketch of an Expanding Perfect Map

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 [49] 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 [40] and proved in [46].

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:*

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

REMARK 5 *So Cock's technique* [40] *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) $k^{r_i} > n_i$ *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 [42] 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 [24] Iványi and Horváth 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 [24] Iványi and Horváth proposed the following definitions and proved Theorem 3.

Definition 1 *Let $\bar{k} = \langle k_1 k_2 \ldots \rangle$ be a strictly increasing sequence of positive integers. A $(\bar{k}^{n^d/d}; n)_{\bar{k}}^d$ - 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 $n, k \geq 2$ 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 $d \geq 1$, $n \geq 2$, $k \geq 2$ and $k_i = N^{\frac{di}{\gcd(d,n^d)}}$ for $i = 1,2,\ldots$ then exists a $(\bar{k}^{n^d/d}; n)_{\bar{k}}^d$ - Growing de Bruijn Cube.*

2.3.2. Infinite de Bruijn d-Tori

2.3.2.1. Increasing de Bruijn d-Tori

Definition 1 *An $(\bar{R}(x); \bar{n}(x))_k^d$ - Increasing de Bruijn Torus is a de Bruijn d-Torus with the property that every $\bar{R}(x)$ sized prefix of the torus is an $(\bar{R}(x); \bar{n}(x))_k^d$ - de Bruijn Torus, where $\bar{n}(x) = \langle n_1, n_2, \ldots, n_{d-1}, x \rangle$ and $\bar{R}(x) = \left\langle r_1, r_2, \ldots, r_{d-1}, n^{f_1 f_2 \cdots f_{d-1} x} / r_1 r_2 \ldots r_{d-1} \right\rangle$ 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* $(\overline{R}(k_i);\overline{n})_{k_i}^{d}$ *- 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* $(\overline{R};\overline{n})_{\overline{k}}^{d}$ *- Alternating de Bruijn Torus is a de Bruijn d-Torus with the property that every* i^i *sized prefix of the torus is an* $(\overline{R};\overline{n})_{i}^{d}$ *- de Bruijn Torus with* $\prod n_j = i$*, and every* $(i+1)^i$ *sized prefix is an* $(\overline{R};\overline{n})_{i+1}^{d}$ *- 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* $\overline{n}$ *Perfect Factor (or de Bruijn Family) of size t and period* $\overline{R}$ *is a family* $\{B_1,\ldots,B_t\}$ *of d-dimensional k-ary toroidal arrays, of period* $\overline{R}$ *each, with the property that for every d-dimensional k-ary matrix M of size* $\overline{n}$ *there is a unique j and a unique* $\overline{i}$ *so that M appears in* B_j *at position* $\overline{i}$*. (We will say that a particular matrix M of size* $\overline{n}$ *appears in B at a position* $\overline{i} = \langle i_1,\ldots,i_d \rangle$ *if M appears in the positions* $\overline{i}$ *through* $\overline{i}+\overline{n}$*.) We call such a Perfect Factor an* $(\overline{R};\overline{n};t)_{k}^{d}$ *- 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 [52] obtained the following

Theorem 3 *Let* $k = \prod_{i=1}^{s} p_i^{\alpha_i}$ *for primes* p_i *and for* $j \leq d$ *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* $i \leq s$ *there is a permutation* $\sigma_i = (\sigma_{i,1},\cdots,\sigma_{i,d})$ *of* $\{1,\ldots,d\}$ *so that for each* $l \leq d$ *we have* $\sum_{j=1}^{l} \beta_{i,\sigma_{i,j}} \leq \alpha_i 2^l$*. 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 [53].

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 1, where we have three populations with three individuals migrating, one from island 2 to1 and two from island 3 to 2.

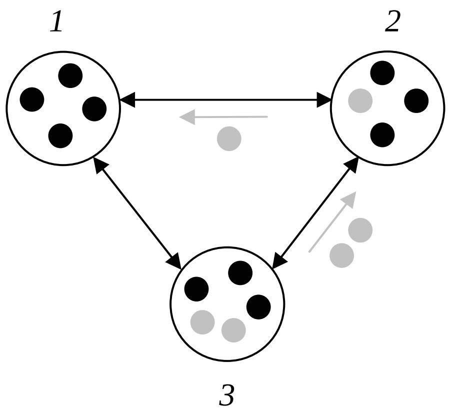

Figure 1 Sketch of the Island Model

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, that is 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 recommendation 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 1, where the black individual in the middle communicates exclusively with the grey ones in its immediate vicinity.

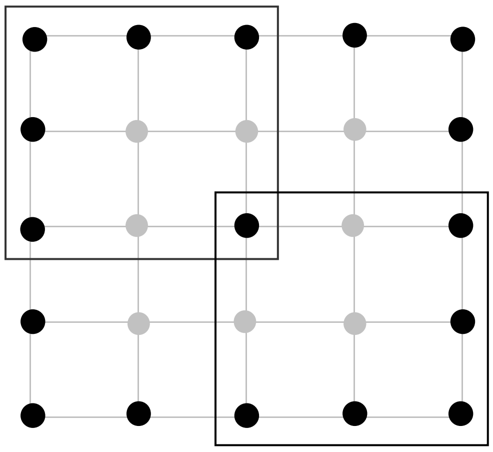

Figure 1 Sketch of the Diffusion Model

Basic parameters and some recommendation 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 1, 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. The Specification of the Evolutionary Algorithm

4.1. One-dimensional Case

4.1.1. Reference Algorithm

A backtrack search is implemented in *DbcBackTrack_v1.java* and *DbcBackTrack_v2.java*. Their functioning is the same, but the implementation differs. The number of basic steps and the backtracks needed is quite less in case of the first algorithm (version 1), but it is much rather memory consuming and slower than the second one (version 2).

4.1.1.1. 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 the all possible candidates for de Bruijn Cycles.

The longest possible cycle that the program can produce, has the length of 127^{127} (the reason for this is the *byte* representation of the alphabet size and the span size, 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 4.1.3.2.), and asks whether to find all the possible ones.

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

4.1.1.2. Specification of the Algorithm

In what follows *k* and *n* denote the size of the alphabet and the span, respectively. The search space is the space of all possible candidate cycles, its size is k^{k^n} (note that k^n is the length of the cycle).

4.1.1.2.1. Version 1.

Basic step: Inserting a suitable tuple into the cycle.

Each candidate is bound to contain the all zeros tuple, so we insert this tuple into the fore-part of the candidate. This part of the candidate is fixed, there is no backtrack from this level (levels of the search tree correspond to the positions in the candidate where we are trying to fit a tuple). 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

At each position of the candidate we try to fit a suitable tuple. Suitable means that we have almost the whole tuple already, only its last element is lacking. The reason for this is that we have inserted the all zeros tuple of size *n* at the first position.

To provide for finding a suitable tuple we have three arrays:

i) **tuples** The possible tuples are stored in a two-dimensional array where the index of the first dimension stands for the decimal value of the *k*-ary tuples.

ii) **triedAlready** On each level we keep a record of the tuples which we have already inspected a branch beginning with. These tuples are stored encoded to decimal in a two-dimensional array where the index of the first dimension stands for the level.

iii) **tuplesInCandidate** The tuples used up to create a candidate are stored encoded to decimal in a one-dimensional array.

The reason for encoding is to make the search in *triedAlready* and *tuplesInCandidate* faster, and to economize on the memory of course.

Once a tuple is picked from *tuples* and if its decimal value is not in *tuplesInCandidate* and in the appropriate array of *triedAlready* then it can be inserted into the candidate. If *tuples* does not contain any suitable tuple, then we make a backtrack and modify the content of *triedAlready* and *tuplesInCandidate* accordingly.

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.

4.1.1.2.2. Version 2.

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

This version is much simpler than the pervious one. The principle of the functioning is similar: inserting something new into the cycle. But in this case that “new” will be an element of the alphabet, not a whole tuple. This solution has a drawback that the cycle should be inspected in every step whether the “perfection” is corrupted by inserting a new element.

To provide the proper functioning of the algorithm we need only one two-dimensional array (*triedAlready*) to store the elements tried already in a level of the search tree.

4.1.1.3. 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 which are 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.

4.1.2. 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 efficient. This is a permutation representation based on *tokens* (see Section 4.1.3.1.), and the components of this algorithm are outlined below.

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

4.1.2.1. 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.

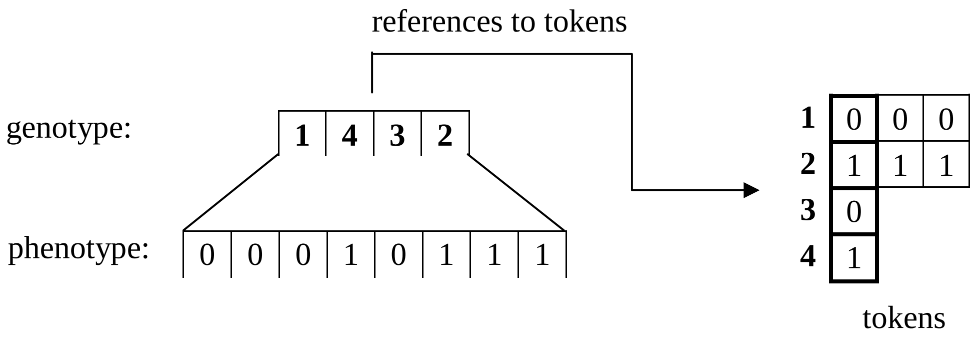

Figure 1 Representation of an individual

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).

4.1.2.2. Initialization and Termination Condition

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

4.1.2.3. 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.

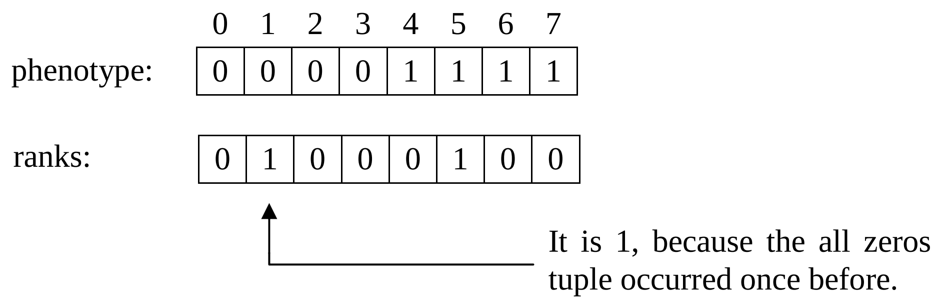

Figure 1 Ranks of the positions

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.

4.1.2.4. Parent Selection Mechanism

4.1.2.5. Variation Operators

4.1.2.5.1. Recombination

4.1.2.5.2. Mutation

4.1.2.6. Survivor Selection Mechanism

4.1.2.7. 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, which are 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, first element reserved for the individuals, 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 object is waiting while it is in progress. 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 the population. 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 populations form a cycle, so an unambiguous neighbourhood relationship can be defined between two populations as illustrated in the figure below.

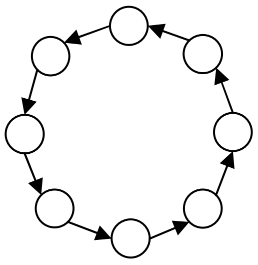

Figure 1 The migration between populations

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, which 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 4.1.1.3.).

4.1.3. Results

4.1.3.1. The Number of Tokens in a Cycle

While inspecting the generated tuples, 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 1 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 1 A $(16;4)_2$ - de Bruijn Cycle

A token is an uninterrupted sequence of identical numbers. The de Bruijn Cycle in Figure 1 has the following tokens:

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

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

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

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

4.1.3.2. 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 *Maple* 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:

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

4.1.3.3. Test Results

4.2. Two-dimensional Case

4.2.1. Reference Algorithm

4.2.2. Genetic Algorithm

4.2.3. Results

4.3. Three-dimensional Case

5. Final Remarks

Appendix A

Here comes the documentation of the PMG software. This will consist of two parts: a user documentation and a development documentation.

Appendix B

Here comes the list of the referenced programs.

Bibliography

[1] I. S. Reed and R. Stewart: *Note on the Existence of Perfect Maps.* IRE Transactions on Information Theory **8** (1962), 10-12.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

[23] L. J. Cummings and D. Wiedemann: *Embedded de Bruijn Sequences.* Congressus Numeratium **53** (1986), 155-160.

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

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

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

[27] 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.

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

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

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

[31] C. W. Marshall: *Applied Graph Theory.* ISBN 0-471-57300-0. Wiley-Interscience, New York, 1971.

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

[33] 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.

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

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

[36] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[49] 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.

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

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

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

[53] A. E. Eiben, J.E. Smith: *Introduction to Evolutionary Computing.* ISBN 3-540-40184-9. Springer-Verlag Berlin Heidelberg, 2003.