Compressing Sparse Tables using a Genetic Algorithm

Karel Driesen
Programming Technology Lab
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2 B-1050 Brussels
kjdriese@vnet3.vub.ac.be

0. Abstract

A genetic algorithm is applied on a sparse table compression technique. The latter takes the form of a variant of the knapsack problem. Since this problem is NP-complete, weak search strategies are promising in giving acceptable solutions in general. The genetic algorithm, with carefully constructed genotype and genetic operators, shows good performance on random samples.

1. Introduction

Sparse tables have many uses. Sparse matrices, for instance, are abundant in linear algebra problems ([VDV 88]). Finite state machine representations, such as parsing DFA's, are often represented as 2-dimensional tables which are mostly empty ([DEN 84]). In [AOE 82], an efficient implementation of trie-trees is described that represents nodes as arrays of constant size. Thus a trie tree can be represented as a 2-dimensional table, associating a node number with an array. The resulting table is sparse. In [DRI 93a], a method lookup technique for dynamically typed object-oriented languages is presented in which the association of an object class and message selector to a method is performed through a sparse table.

In numerical analysis, a sparse matrix is usually represented as a collection of linked lists, each gathering all the non-empty elements of a column or row. This representation takes little memory (twice as much pointers as the number of non-empty entries[1]) and is adequate for a number of numerical analysis algorithms, such as Gaussian elimination. The main operation employed is an enumeration of the elements of columns. However, for all other aforementioned applications the most important operation is the retrieval of a single element. In general, this takes in the order of O(*n/2*) operations, with *n* the number of non-empty entries in a particular column or row. This is often too slow for practical purposes.

Tarjan, in [TAR 79], presents a sparse table representation which performs retrieval of an element in constant time. This technique, which we will discuss in section 2, is preferable over a list representation when the table is static. Otherwise the cost of insertion of a non-empty element can be prohibitive. Memory use is potentially larger, since slices of the table need to be fitted together as one-dimensional puzzle pieces. The unused space that results from inadequate fitting is pure overhead. At the moment, no efficient algorithm exists to find the best possible fitting in the general case.

[1] For clarity and convenience, we assume that entries and pointers occupy the same amount of space. In the pure object-oriënted language context that motivated this work, this is the case.

This is where the genetic algorithm can play a part. As we will discuss in section 4, the only way to tackle a hard (NP-complete) problem in general is by employing weak search. I.e. we do not look for the **best** solution, but are satisfied with one approaching it. Weak search techniques also have the property of rendering better results if one is willing to dedicate more computing power to the calculation.

2. Sparse arrays

In this section we will treat Tarjan's table representation. An example sparse table is given below:

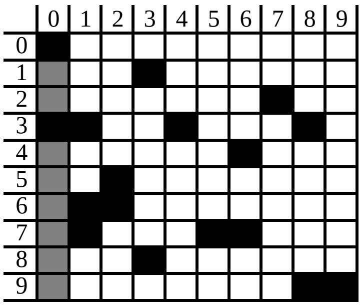

Figure 1: a sparse table

The standard representation of a two-dimensional table *T* of *m* rows and *n* columns is a one-dimensional array *A* of size $m \, . \, n$. The rows of the array are placed, one after the other, in *A*. Element *T[i,j]* at row *i* and column *j*, can be found in entry $A[i \, . \, n + j]$.

In Tarjan's scheme, the rows are not placed after eachother, but overlap in such a way that only non-zero elements have a unique index in *A*. In other words, a vector *R* of size *m* is calculated, which gives the offsets of each row of *T*, such that for all non-empty entries *A[z]* there is exactly one non-empty element *T[i,j]* for which $R[i] + j = z$. The time needed to retrieve a non-zero element is constant, as opposed to sparse matrix representations based on linked list datastructures.

If zero elements can be retrieved as well, two alterations of the above setup need to be implemented. First, a separate array *B* of the same size as *A* needs to be maintained, in which $B[z] = o$, if entry *A[z]* contains a non-zero element of the row starting at offset *o*. Secondly, the offsets need to be unique, so that each row has a unique offset (otherwise it can not be determined to which row a non-empty element belongs). This is easily accomplished by adding dummy non-zero elements at column 0, for all rows (the gray entries in figure 1).

There are still many solutions for *R* that respect all the above conditions. The standard representation does, for instance. In order to gain memory, offsets have to be found that fit the rows of *T* tightly together. This can be measured by the fillrate *f* of *A*, given by *e/s*, where *e* is the number of non-zero elements of *T*, and *s* is the size of *A* (the index of the rightmost non-empty entry). If *f* approaches 1, the memory overhead approaches that of a linked list. The figure below gives a good mapping for the above example:

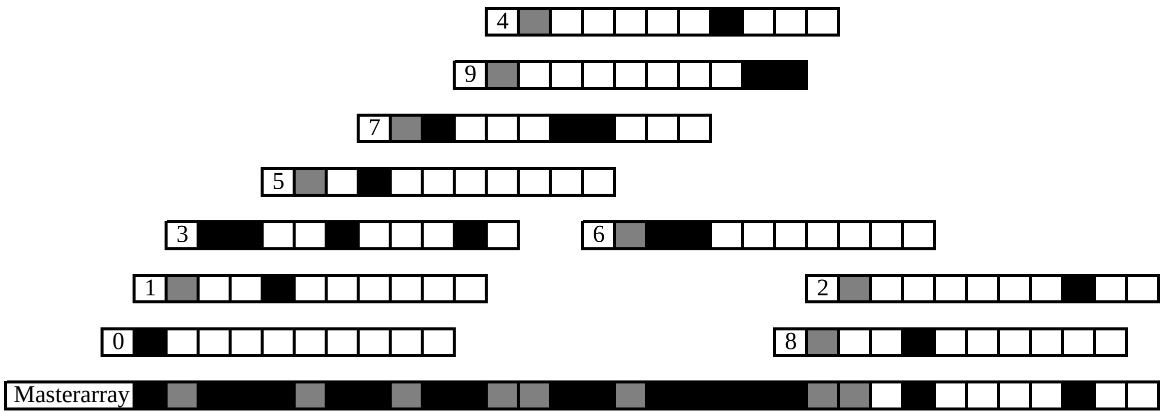

Figure 2: Mapping of table rows onto a masterarray

Finding the offsets that maximize the fillrate is an NP-complete problem. In [TAR 79] R is constructed by inserting the rows of T, one after the other, in the first fitting space in A. If m is much larger than n this is adequate. In [AOE 82] a fill rate of 99% was obtained for a trie-tree representing an English dictionary of thirty-thousand words. In terms of the table, the number of columns was about 30, while the number of rows was little less than 50,000. A very large number of extremely small puzzle pieces makes the fitting process efficient. This is because the natural diversity in row signatures makes sure that almost every occurring empty space eventually gets filled up by a matching piece. However, if m is of the same order of magnitude as n, and the non-zero elements are spread over the rows in a random way, the fillrate obtained is very low.

Tarjan's first fit scheme has one degree of freedom: the order in which the rows are fitted in A. The numbering of columns gives an additional parameter to play around with. In some cases the applications allows the column numbers to be chosen freely. If not, then a small retrieval overhead has to be paid in the form of a mapping of a column to a different column number. This can be implemented by an array C, in which a permutation of the column numbers is stored. Element $T[i,j]$ is then retrieved as $A[R[i] + C[j]]$. Maximalisation of the fillrate in this context amounts to finding permutations of rows and columns that minimalise the size of A[2].

3. Using a genetic algorithm

Finding a selector numbering and a row order which maximises the fillrate can be viewed as an combinatorial optimisation problem. Since the problem is NP-complete, the best solution cannot possibly be found, in the general case, in less than exponential time[3].

Weak search techniques provide the means of finding good solutions to NP-complete problems. They have the advantage of being generally applicable, and easily rendering better solutions by adding computing power. Domain knowledge remains important since it guides the setup of the search process ([MAN 91]).

[2] Note that this does not cover the total search space of all possible offsets. However, it allows sufficient variation to find adequate fillrates.

[3] To our current knowledge, that is. Although it has not been proved that P ≠ NP, experience strongly suggests that NP-complete problems cannot be solved in polynomial time in the general case.

A wide choice of weak search strategies is available from the AI field. Artificial neural networks, simulated annealing and genetic algorithms, among others, have been successfully applied to a wide range of problems.

We chose genetic algorithms to explore our problem space for the following reasons:
- it is a simple, elegant technique, easily implemented
- it is robust, and, if necessary, can be made more robust in a straightforward way (increase the size of the population, for instance)
- the actual process to optimise can be treated as a black box, thus enabling us to reuse the fitting code (implemented for [DRI 93a]) unaltered
- domain knowledge is easily incorporated by altering mutation and crossover rules
- to our knowledge, the best solution for the 442-city travelling salesman problem (an important benchmark in the optimisation field) was reached by employing a parallel GA ([MUH 88])

In the following section we will discuss the setup of the employed genetic algorithm.

3.1. The algorithm

The different aspects of the GA are treated in the following section. First the genotype is defined. Then the genetic operators are specified. Finally the particular flavour of GA we use, as well as its scalar parameters are discussed.

3.1.1 The genotype

The first step in applying GA's to a problem is to define the encoding of points in the problem space. Every possible solution needs to be expressed as a fixed-length string of symbols from a given alphabet[4]. This string is called a genotype. The chosen representation will influence the form of the problem space, and its associated fitness landscape. The latter is the result of applying the fitness function, which represents how good a certain solution is, to every point of the search space. The topology of the fitness landscape is known to be a determining factor for the effectiveness of a GA. If the landscape is gently sloped, the correlation of fitness between neighbouring solutions (which share most characteristics) will be high. If it is rugged, on the contrary, similar solutions will have widely different fitness. In the latter case an abundance of local maxima will prevent the GA from finding good solutions.

The figure below gives the genotype encoding used in our experiment, for a tabel with 6 rows and 10 columns:

column	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	row
renumbering	7	4	2	5	6	1	9	8	0	3	3	1	0	5	2	4	fitting order

Figure 3: Genotype encoding of column renumbering and row processing order

[4] This is not a contingent property of GA's. In principle, genotypes can be arbitrarily complex constructions. The genetic operators are easier to define on fixed-length strings, however.

Only the lower row is actually stored (the higher is implicitly given by the index in the vector). Hence, a valid genotype is an ordered sequence of one instance each of all numbers between 0 and the number of columns (exclusive), followed by the same for the rows. The neighbourhood defined by such a scheme adheres closely to an intuitive feeling of which solutions are close together. Two gens which have most numbers in the same order will produce similar row signatures and similar row orderings

3.1.2 The genetic operators

3.1.2.1 Mutation operator

The next step in the construction of the GA is the definition of a mutation operator. This is a function which takes one genotype as argument and returns a slightly altered one. In [MAN 91], three mutation operators are given:

Swap operator:

	column	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	row
gen	renumbering	7	4	**2**	5	6	1	9	**8**	0	3	3	1	0	5	2	4	fitting order
				^					^									
result	renumbering	7	4	0	5	6	1	9	8	2	3	3	1	0	5	2	4	fitting order

Figure 4: swap operator on entries 2 and 7 of the column part

Two randomly chosen column (or row) numbers are swapped.

Reverse operator:

	column	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	row
gen	renumbering	7	4	**2**	**5**	**6**	**1**	**9**	**8**	0	3	3	1	0	5	2	4	fitting order
				^					^									
result	renumbering	7	4	8	9	1	6	5	2	0	3	3	1	0	5	2	4	fitting order

Figure 5: reverse operator on entries 2 and 7 of the column part

The column (or row) numbers between two randomly chosen points are reversed.

Remove&insert operator:

	column	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	row
gen	renumbering	7	4	**2**	**5**	**6**	**1**	**9**	**8**	0	3	3	1	0	5	2	4	fitting order
				^					^									
result	renumbering	7	4	5	6	1	9	8	2	0	3	3	1	0	5	2	4	fitting order

Figure 5: remove&insert operator on entries 2 and 7 of the column part

The column (or row) number at a random position is removed and inserted at a random position.

These three possibilities give nine possible mutation operators for the genotype as a whole. In [MAN 91], the correlation between the fitness of a gen and its mutation gives a measure of the suitability of a mutator. The higher the correlation, the gentler the slope of the fitness landscape, as it is determined by the operator. We calculated the correlation coefficient for each mutator seperately on rows and columns[5].

	Col	Row
Swap	0.36	0.55
Reverse	0.58	0.71
Rem&Ins	0.58	0.77

Figure 6: correlation coefficients of mutation operators.

The Swap operator is obviously inferior to both Reverse and Remove&Insert. Since the latter is better on rows, and the same on columns, we chose Remove&Insert for both parts of the genotype. For a given mutation, the chance of choosing row mutation was taken proportional to the number of rows versus the total size of the genotype (for a 10 by 30 table, 10 out of 40 mutations take place on rows).

3.1.2.2 Crossover operator

The crossover operator is a function which takes two genotypes as argument and returns a genotype that shares some characteristics of both arguments. We implemented two operators:

PMX operator:

	column	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	row
gen1	renumbering	7	4	2	5	**6**	**1**	**9**	8	0	3	3	1	0	5	2	4	fitting order
gen2	renumbering	**2**	**7**	**8**	**3**	9	0	1	**5**	**4**	**6**	4	0	3	2	1	5	fitting order
						^		^										
result	renumbering	2	7	8	3	6	1	9	5	4	0	4	0	3	2	1	5	fitting order

Figure 7: PMX-crossover with crossover points 4 and 6

Two random positions in either the row- or the column part of the genotype determine the crossover points. The numbers between these points are copied from the first to the second gen. To render a valid genotype, numbers that are overridden in the second genotype are swapped with the number that overrides them. In the example, crossover points 5 and 7 in the column part give a genotype that retains six column numbers of the second genotype and three of the first, while one new column number occurs. This emergence of new column numbers, not present in either of the two parent genotypes, is unavoidable.

OX operator:

	column	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	row
gen1	renumbering	7	4	2	5	**6**	**1**	**9**	8	0	3	3	1	0	5	2	4	fitting order
gen2	renumbering	**2**	**7**	**8**	**3**	9	0	1	**5**	**4**	**6**	4	0	3	2	1	5	fitting order
						^		^										
result	renumbering	7	8	3	0	6	1	9	5	4	2	4	0	3	2	1	5	fitting order

Figure 8: OX-crossover with crossover points 4 and 6

[5] Each mutator was applied 10 times on 400 different gens.

In OX, or Ordered Crossover, the numbers between two random crossover points are copied from the first genotype. Then the numbers of the second genotype are copied to the result, in the order in which they occur, starting behind the copied section. Numbers that were copied from the first gen are skipped in the second.

Again the correlation between the parents and the child gen was calculated:

	Col	Row
PMX	0.21	0.29
OX	0.10	0.24

Figure 9: correlation coefficients of crossover operators

Although PMX performs better then OX, the correlation is still very low. This means that the GA will perform as well with or without crossover. Experiments confirmed that the time needed to reach an adequate fillrate, measured in realtime and in number of generations, is the same if only mutation is used. Although this observation indicates that the problem is GA-hard, the GA still reaches good solutions faster than if one would just generate many random permutations and keep the best solution. In the GA, better solutions are constructed from good ones, even if only mutation is employed.

3.1.2.3 Fitness function

The last function we need to define is the fitness function. The fillrate, as defined in section 2, serves this purpose. All rows are fitted, in the order specified by the genotype, into the masterarray in the first space that accommodates them. The column numbering determines the signature of non-empty entries. After the fitting process is terminated the total number of non-empty entries is divided by the distance between the leftmost and rightmost non-empty entry. This is a number between 0 and 1. The fitness function could be scaled to reward good points in a more than linear way, for instance by squaring it. For our purpose this is irrelevant since the offspring of a genotype is determined by it's ordering relative to the rest of the population (see next section). This ordering is not influenced by scaling the fitness function.

It should be noted that the calculation of the fitness function is several orders of magnitude slower than the mutation and crossover operators. This gets worse for large tables, since the fitness calculation is of order O(n^2) for the first fit approach, with n the number of non-empty entries. The crossover and fitness operators perform at O(n). In [DRI 93b] we discuss an algorithm that does the fitting approximately in O(n), but is only applicable when the rows are sufficiently clustered. This property depends on the problem domain, so it does not apply in the general case, and we have to stick to the "vanilla" first-fit scheme. Due to the large time overhead of calculating the fillrates, the test samples had to remain fairly small.

3.1.3 Setup and parameters of the GA

To give a clear view on the choices still left in making the GA operational, we given an outline of the basic genetic algorithm:

Step 0: Set the time t=0 and generate the initial population P_0, of N points

Step 1: Calculate the fitness of every point in P_t

Step 2: Select points from the population P_t , proportional to their fitness (i.e. better points are more prone to be selected)

Step 3: Apply mutation and crossover on this selection, which gives population P_{t+1}

Step 4: Go back to step 1

Note that this scheme generates a totally new generation with each iteration. For large populations, this is not a problem, since there will probably be many equally good solutions to choose from with each generation. For small populations this is rather wasteful, as the best current solution will be destroyed in step 3, and the small number of genotypes does not guaranty that an equally good solution is generated. The complexity of the fitness function in our problem domain rules out large populations. Therefore, we adhere to the scheme described in [BOO 89], in which the new generation is merged with the old one and the inferior solutions are dropped from the population. This gives the following algorithm:

Step 0: Set the time t=0 and generate the initial population P_0,, of N points

Step 1: Calculate the fitness of every point in P_0,

Step 2: Select points from the population P_t , proportional to their fitness (i.e. better points are more prone to selection)

Step 3: Apply mutation and crossover on this selection, which gives population P'_{t+1}

Step 5: Calculate the fitness of every point in P'_{t+1}

Step 6: Merge P_t and P'_{t+1}. Drop all but the best N solutions. This gives P_{t+1}

Step 7: Go back to step 2

This setup ensures that a good solution remains in the population until a number of better ones are found.

A number of parameters remain to be fixed. The size N of the population is set at 100. In [MAN 93], this number is suggested. In the experiments we observed that this was large enough to allow for wide diversity in the population.

The selection of points to be crossed over is the next parameter. The crossover rate, which is the fraction of the population selected, was set at 80%. The best genotype has 100% chance of being selected. Then the chance of selection decreases linearly. The worst genotype has 60% chance of being selected. This scheme selects good solutions more often, but does not exclude inferior solutions from contributing to the next generation (otherwise there would be no point in having them around).

The last parameter to set is the number of generations to be calculated. We observed that the population converged, after 20-50 generations, to identical genotypes. This happens because the best solutions are preserved from one generation to the next. Instead of fixing the number of generations, the stopping criterium employed is the variety of the population. When the populations consists only of identical copies of the same gen, we stop.

3.2. Experiments

We tested the GA on twelve kinds of tables of varying dimensions. In the table below, Rand3 to Rand6 have the same dimensions, but a varying proportion of non-empty entries. Rand7 to Rand12 show the effect of row and column size, for tables with comparable standard fillrate.

Sample	**r**	**c**	**r*c**	**ne**	**stand**	**twa**	**gen**
Rand1	10	10	100	27	27 %	34 %	83 %
Rand2	20	10	200	55	28 %	40 %	81 %
Rand3	10	20	200	43	22 %	30 %	74 %
Rand4	10	20	200	56	28 %	33 %	79 %
Rand5	10	20	200	80	40 %	47 %	76 %
Rand6	10	20	200	108	54 %	60 %	77 %
Rand7	20	20	400	75	19 %	27 %	77 %
Rand8	30	10	300	64	21 %	37 %	78 %
Rand9	10	30	300	59	20 %	26 %	74 %
Rand10	30	20	600	112	19 %	26 %	78 %
Rand11	20	30	600	121	20 %	24 %	72 %
Rand12	30	30	900	189	21 %	25 %	72 %

Figure 10: results of the genetic algorithm on various test samples. **r** gives the number of rows, **c** is the number of columns, **r*c** is the total number of entries in the table, **ne** the number of non-empty entries, **stand** is the resulting fillrate of a regular 2-dimensional array, **twa** is the fillrate reached by placing rows after each other without overlap, **gen** is the fillrate reached by the genetic algorithm.

The last column gives the average, over 5 runs, of the fillrate of the best solution found by the genetic algorithm. The “twa” column (from Table Width Allocation in [DRI 93b]) gives the fillrate that results when all the rows are placed one after the other in the masterarray, the rightmost non-empty entry of a row touching the leftmost of it’s successor. This is the minimal solution, as none of the rows are fitted together.

A first observation is that the genetic algorithm performs in a fairly constant way. Although the resulting fillrate becomes lower for larger tables, a ninefold increase in size accounts for a drop of only 11% (Rand1 vs. Rand12). The sparseness of the table also has little influence on the results, as is shown by Rand3 to Rand6.

For the same tablesize and fillrate, the number of rows versus the the number of columns makes a difference. As mentioned earlier, tables with many rows and few columns are easier to compress than flat, wide tables[6] (Rand2 vs. Rand4, Rand8 vs. Rand9, Rand10 vs Rand11).

Although the GA performs well, the main disadvantage of using it, is the large amount of computer power needed. The largest sample took in the order of several hours to compute. The lengthy calculation of the fitness function is responsible for more than 90% of this.

4. Conclusions & related work

We have designed and tested a genetic search approach on a sparse table compression technique. A genetic algorithm behaves well on this problem.

The main motivation for this work was our previous experience with sparse arrays as representations of method tables for object-oriented, dynamically typed languages. A dedicated heuristic, which only involved only one calculation of the fillrate, was presented and tested in [DRI 93b]. For small samples, the GA outperformed this method. For real-life samples, the heuristic was better, probably because the population size was not large enough to allow for sufficient diversity.

The enormous amount of computing power needed to calculate large samples prohibits the use of a genetic algorithm for this application. The calculation of the Smalltalk class library, which took a heuristic algorithm 40 minutes, would take several days with the method outlined here. Since changes in a class library occur on a daily basis in a development environment, the technique described in this paper, though potentially reaching very good solutions, is simply not practical.

We do not rule out the use of GA's for sparse table compression in other applications, however. The main factor to consider is the lifetime of the table. If this is sufficiently large, obtaining a high fillrate through a genetic algorithm can be well worth the effort. We conjecture that, given enough processing time, a genetic search approach will reach a better result than any domain-dependent heuristic.

A second factor to consider is the way entries are clustered together. The fitting process can take advantage of the presence of clusters by employing a dedicated heuristic. Some problem domains show a very random scattering on non-empty entries. In such a case, a weak search approach is the only alternative. Parser tables (see [DEN 84]), for instance, could benefit from the technique described here, because the lifetime of a parser table is as long as the lifetime of the compiler in which it is employed, and because there is no inherent clustering present in the table. We plan to look at this application in the future.

[6] It is of course easy to compress the transposition of the table to obtain a better fillrate. The experiments show that this is probably a worthwile effort.

5. Bibliography

[AND 92] P. André, J. C. Royer.
Optimizing Method Search with Lookup Caches and Incremental Coloring
OOPSLA'92 Proceedings p.110-126

[AOE 82] J. I. Aoe, K. Morimoto, T. Sato.
An Efficient Implementation of Trie Structures
Software-Practice and Experience, Vol.22, nr 9, September 1992, p. 695-721

[BOO 89] L. B. Booker, D. E. Goldberg, J. H. Holland
Classifier Systems and Genetic Algorithms
in Artificial Intellingence, vol. 40, p.235-282

[DEN 84] P. Denker, K. Dürre, J. Heuft
Optimizationof Parser Tables for Portable Compilers
ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1994, p. 546-572

[DIX 89] T. Dixon, M. Vaughan, P. Sweizer.
A fast Method Dispatcher for Compiled Languages with Multiple Inheritance
OOPSLA'89 Proceedings p.221-214

[DRI 93a] K. Driesen.
Selector Table Indexing & Sparse Arrays
OOPSLA'93 Proceedings, p.259-270

[DRI 93b] K. Driesen.
Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages
Masters Thesis 1993, Faculty of Sciences, Vrije Universiteit Brussel

[MAN 91] B. Manderick
Selectionism as a Basis of Categorization and Adaptive Behavior
PhD Dissertation 1991, Faculty of Sciences, Vrije Universiteit Brussel

[MUH 88] H. Mühlenbein, M. Gorges-Schleuter, O. Krämer
Evolution Algorithms in Combinatorial Optimisation
in Parallel Computing, North-Holland, Amsterdam

[TAR 79] R. E. Tarjan, A. C. Yao
Storing a Sparse Table
Communications of the ACM, vol 22, no 11, November 1979, p. 606-611

[VDV 88] Henk Van Der Vorst
Parallel rekenen en supercomputers
Academic Service