FAKULTÄT FÜR !NFORMATIK

Enhancing a Genetic Algorithm by a Complete Solution Archive Based on a Trie Data Structure

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andrej Šramko
Matrikelnummer 0125391

an der:
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl

Wien, 10.02.2009

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

Genetic algorithms are robust optimization techniques that have proven to be effective for a large variety of optimization problems. Many particular improvements have been designed to solve special problems. However, it is difficult to find techniques which can be used more generally. In this thesis, I describe a mechanism that should improve the ability to find a better solution for a larger spectrum of genetic algorithm applications. A complete solution archive based on a trie data structure is introduced.

The idea of the archive is to efficiently store all visited solutions, avoid revisits, and have a good and intelligent mechanism for transforming already visited solutions into similar unvisited ones.

The genetic algorithm can be seen as a separate module which generates solutions in a specific way. Every created solution is forwarded to the trie. As the trie accepts the solution, it checks whether it is already included in the archive. If the solution is not yet in the archive, it is simply inserted. On the other hand, when the solution is in the trie, a revisit would occur. Avoiding the revisit can be done in several ways; in general the aim is to derive a similar but yet unvisited solution, and the trie data structure supports this well. It is important to find a good balance between the quality of the changed solution and the effort needed to change it. After inserting or altering a solution, it is sent back to the genetic algorithm module and then handled as usual.

The approach has been implemented and tested on three problems; the Royal Road function, NK landscapes, and the MAX-SAT problem. The standard genetic algorithm is compared to the different variations of the new approach that use the archive. Results show that in many cases the archive contributes to the quality of the solutions, or reduces the number of iterations for finding optimal solutions.

Acknowledgements

First I want to thank my advisor Professor Günther Raidl, for his help, guidance, patience and also for giving me valuable feedback and reviews.

I want to dedicate this thesis to my grandfather Ľudovít Treindl. His steady scientific work in many universities throughout the world and his love for his wife were the best example for me. His never-ending interest about my studies even in the late ages of his life and his support gave me strength to finish this work.

Also many thanks to my closest family. My mother's love, my father's faith in me and the nearest friendship of my brothers Martin and Filip are the most valuable things for me.

I thank to my closest friends Rastislav Habala and Maria Dolezal for their love and time they spent listening to me over all years of my studies. This thesis also would not be finished without the support and guidance of Stephan Turnovszky, auxiliary bishop of Vienna.

Special thanks to Mary Jo Szekeres for her English-grammar correction in this thesis.

Last but not least, the biggest thanks belongs to Almighty God, whose goodness and mercy shall follow me all the days of my life [Ps 23, 6]. I am nothing but just a tool in His hands and even this thesis is testimony of His love for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Global optimization

One of the biggest challenges for computer science is optimization. Mankind searches for optimality in every part of life. Everyone wants to get all reachable goods with minimal effort. Also in nature is a hidden search for the optimal state. This is described by physical laws. So, there is no need to wonder when we see that from the beginning one of the main challenges for computer science is optimization of different problems [42, 60].

> "Global optimization is the branch of applied mathematics and numerical analysis. The goal of global optimization is to find best possible elements s^* from a set S according to set of criteria $F = \{f_1, f_2, f_3\}$."[60]

A wide variety of optimization methods has been developed [43]. These strategies can generally be divided into two classes: exact (or deterministic) and heuristic (Figure 1.1).

Exact methods do an exactly determinded decision at every step of the algorithm. This kind of algorithms is used efficiently if there is a possibility to explore the search space in a systematic way.

The simplest exact algorithm is the exhaustive search algorithm. It tries all possible solutions from a given set and picks the best one. However, this effort is often undesirable even for small problems [60].

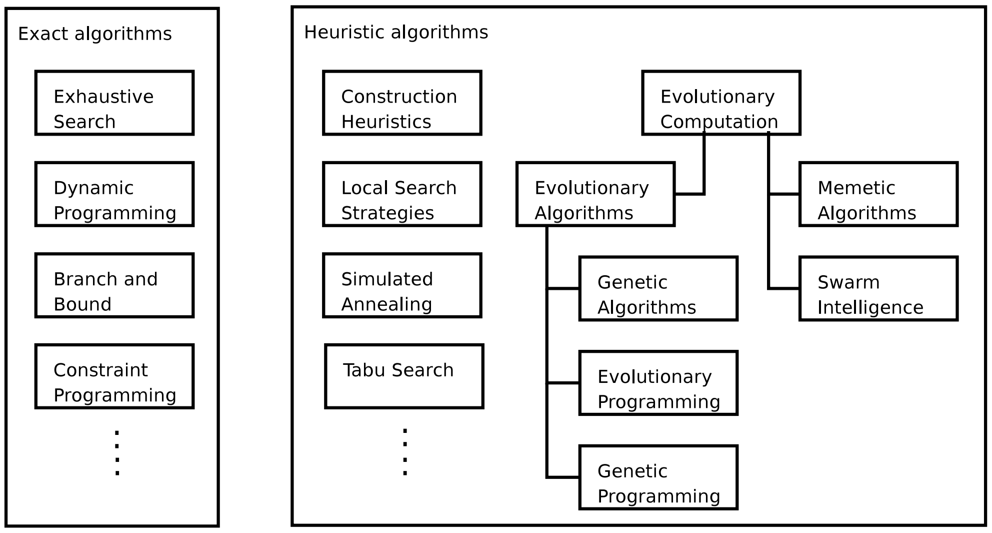


Figure 1.1: Exact and heuristic algorithm classes - overview

Similar to exhaustive search is Dynamic programming. Its advantage is that it avoids re-computation by storing the solutions of subproblems. The main problem of this method can be the formulating of the solution process as a recursion.

The Branch and Bound algorithm belongs also to the class of exact algorithms. It was proposed by Land and Doig in 1960 for linear programming [34]. This method is based on the divide and conquer strategy. It enumerates all the candidate solutions inside estimated bounds. All other candidates, which are outside these bounds, are discarded. The bounds are iteratively optimized. The algorithm stops when the set of candidates is reduced to a single element.

The decision whether to use an exact algorithm or not depends on search space and possible solution characteristics. The factors which make the application of exact methods easier are: lower dimensionality of the search space, clear relations between solution candidates and their fitness values, etc.

Another estabished strategy class is heuristic algorithms. Heuristics help the algorithm to decide which one of a set of possible solutions is to be examined next.

"A heuristic [38, 47, 44] is a part of an optimization algorithm that uses the information currently gathered by the algorithm to help to decide which solution candidate should be tested next or how the next individual can be produced. Heuristics are usually problem class dependent." [60]

I will describe some basic algorithms which belong to the heuristic class closer.

A construction heuristic is an algorithm that generates a solution according to some construction rules. This method is often fast, but very problem-specific.

Local search algorithms are probably the biggest group of heristic algorithms. They start to work on a single current state and then they transcend only to neighbors of the current state [51]. Local search algorithms are not systematic but have two major advantages: they are often able to find solutions in large or infinite search spaces, and they use often only a constant amount of memory. Their biggest disadvantage is the processing time. There are many different search strategies: Depth-First Search, Depth-Limited Search [51], Greedy Search, Random walks [26] and Adaptive Walks.

Simulated annealing algorithm [4], was invented in 1983. It occasionally accepts solutions that are worse than the current. The probability of accepting the worse solution decreases with time.

Tabu search [7] is a method, which uses memory structures to store a number of last moves or evaluated solutions. It prohibits the repetition of these moves or re-evaluating of solutions to escape from local optima.

Evolutionary Computation a spectial heuristic class. It is based on iterative improvement of a selected set of multiple solution candidates. This set of solution candidates is called population. The most important members of Evolutionary Computation are Evolutionary Algorithms (EA), which are described later in this chapter, Memetic algorithms, and Swarm Intelligence [60].

Memetic algorithm is a combination of the Evolutionary algorithm with the Local earch algorithm [41]. Basically, the approach combines local search heuristics with crossover operators. After creating the initial population randomly or using a heuristic, a local search is started to improve the fitness of each population individual.

Swarm intelligence [13] is a technique based on the study of collective behavior in decentralized, self-organized, systems. The most important classes of this approach are Ant Colony Optimization and Particle Swarm Optimization.

1.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a population-based type of optimization algorithm that use biological principles. These are: mutation, crossover, natural selection, and survival of the fittest [5, 6]. By applying these principles of Darwinian evolution, they try to find solutions for difficult problems. Additionally, the EA is goal-driven, which is a change in semantics compared to typical biological processes. The EAs only make a few assumptions about the underlying fitness landscape. This is an advantage compared to other optimization methods, because they consistently perform well in many different problem categories [60].

The principles of the EA can be written in the following four steps:

- Initialize population – create initial population of random individuals
- Evaluation – compute fitness values of the solution candidates
- Selection – select the fittest individuals for reproduction
- Reproduction – create new individuals from the mating pool by crossover and mutation

The process is started with the initialization step. The next three steps are repeated until the termination criterion is reached. The EA process is displayed in Figure 1.2.

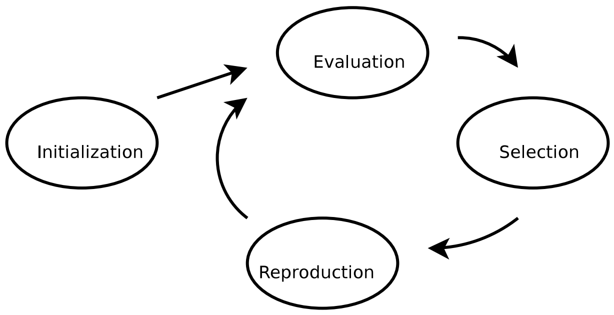


Figure 1.2: Four basic steps of the Evolutionary algorithm

Different schools of EAs have evolved during past decades: Genetic Algorithms, mainly developed by J. H. Holland in the USA [25], Evolutionary Strategies, developed in Germany by I. Rechenberg [48] and H.-P. Schwefel [52], and Evolutionary Programming [16]. Each of these constitutes a different approach, however, they are inspired by the described principles of natural evolution. Every class of the EAs implement these principles in a specific way. Because I was focused on Genetic Algorithms, they are described in detail in Section 1.3.

1.3 Genetic Algorithms

Genetic Algorithms (GAs) are one of the most-used dialects based on the principles of EAs. They are momentarily widely spread and well-established metaheuristic for solving NP-Problems. For them, it is typical that the elements of their search space are binary strings. The roots of the GAs stretch back to the middle of the 20^{th} century, but they were not popular or widely recognized until J.H.Holland published his work [24, 25]. Today the GAs are used in many different areas like scheduling, chemistry, medicine, data mining and data analysis, geometry and physics, economics and finance, networking and communication, electrical engineering and circuit design, image processing, combinatorial optimization, etc. [60].

1.3.1 GA structures

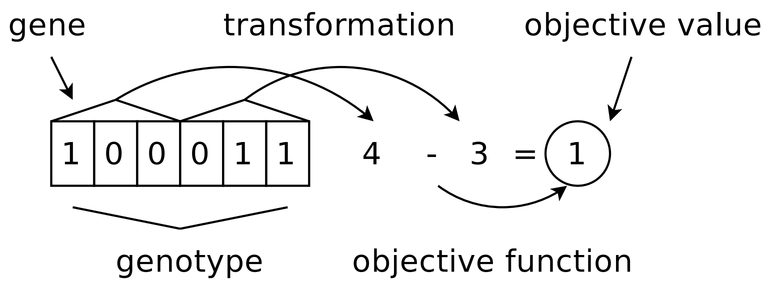


Figure 1.3: basic structures of the GA

Let me define the basic element structures used in the GAs. The basic informational unit is a gene. A number of genes which are together form a genome, genotype, or chromosome.

Depending on the genome, a gene can be a bit, a real number, or any other structure. At any time, the whole genome can be transformed into its real representation and evaluated with an objective function. An example is given in Figure 1.3. The genotype, which consists of six genes, represents two binary-coded numbers. Each number is represented by three genes. After the transformation of the genotype, two numbers result (4 and 3). Then, the defined objective function subtracts the second number from the first one. In the example, the objective value (also called fitness value) of the given genotype is 1. The maximum objective value could be reached by the genotype 111000 $(7 - 0 = 7)$. The chromosomes used in the GA can also have different lengths, but in my work I only used chromosomes with a fixed length. In Section 1.3.2, the operators usually used in GAs with fixed length chromosomes are defined.

1.3.2 GA operators

As is shown in Figure 1.2, the whole process of each EA, and therefore also the process of the GA, is based on four steps: initialization, evaluation, selection, and reproduction. The evaluation step is usually problem-dependent, so I will describe it later when I introduce the test functions I used. Initialization, selection, and reproduction can be problem-independent. Moreover, the reproduction step can be divided into two individual steps: crossover and mutation. These steps are performed repetitively by applying the so-called GA operators, which I define in this section. There is a variety of each of these operators, but I will define only those which I applied in my work.

Initialization The GA, just like every other EA, starts with an initialization. In this step, the first population is created and initialized. Usually, the population individuals are simply created by random initialization of each of their genes. This is necessary because no solutions are created yet, so we cannot use them to derive new ones.

Selection The second step in the GA is the selection. During this step, some individuals are picked up according to their fitness values from the population and placed into the mating pool. Afterwards, in the reproduction step, individuals are used from this mating pool. Selection may be carried out in several manners, depending on the algorithm chosen.

In my approach, I used the tournament selection algorithm. The algorithm holds a tournament among s_t solution candidates, where s_t is tournament size. The winner of the tournament is the individual with the highest fitness value of the s_t competitors. Afterwards, the winner is inserted into the mating pool. The mating pool has a higher average fitness value than the average population. This fitness value difference provides a selection pressure. The selection pressure drives the GA to improve the population fitness value over succeeding generations. The convergence rate of the GA is largely determined by the selection pressure. The higher selection pressure results in higher convergence rates. Increasing the s_t parameter increases selection pressure [39].

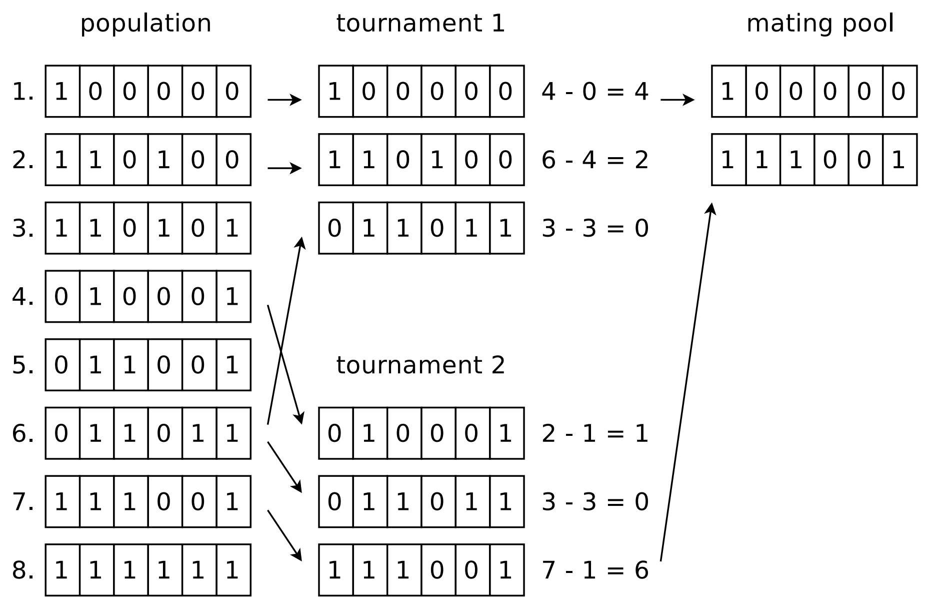


Figure 1.4: Tournament selection

In Figure 1.4 an example of tournament selection performed on a population of 8 solutions is displayed. The two spaces in the mating pool have to be filled, so two tournaments with $s_t = 3$ are arranged. The two winners of these tournaments (the solutions with highest fitness values) are put into the mating pool.

Crossover The crossover is a recombination of two string chromosomes. It is an operation which creates a new solution candidate by combining the features of two existing ones. The crossover is performed in the following way: two parental chromosomes are split at a randomly determined crossover point. Afterwards, the new child chromosome is created by joining the first part of the first parent together with the second part of the second parent. This method is called single-point crossover and it is shown in Figure 1.5. There are also crossover methods, where, for example, both parental chromosomes are split at two or more points (multi-point crossover). For fixed-length strings, the crossover points for both parents are always identical.

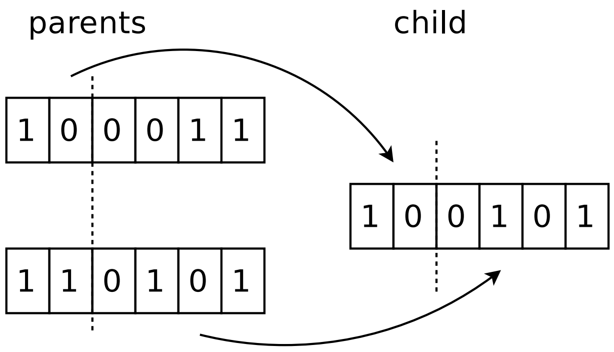


Figure 1.5: Single-point crossover

Mutation In contrast to the other classes of EAs, the mutation in the GA is usually a background operator. However, the mutation is still an important method of preserving the diversity in the population. Mutation creates a new chromosome by modifying an existing one. In fixed-length string chromosomes, it can be achieved by modifying the value of one element of the chromosome, as illustrated in Figure 1.6. In binary coded chromosomes, the gene bits are simply toggled.

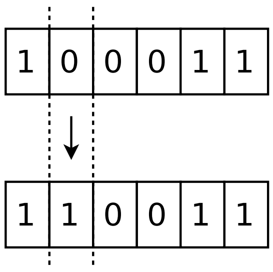


Figure 1.6: Mutation of one gene

1.3.3 GA parameters

In the area of GAs, there is a wide variety of possible parameter settings. The performance and success of the GA approach applied to a problem is also given by parameter settings. In my work I operated with following parameters:

- Tournament size s_t – size of the tournament when performing the selection
- Crossover rate r_c – probability for performing the crossover between two selected individuals
- Mutation rate r_m – probability for performing the mutation
- Archive use u_a – defines whether the GA stores visited solutions in the archive or not

1.4 Complete archive for GAs

Many parameters and improvements have been designed to solve special problems. However, it is difficult to find techniques which can be used universally. In my thesis, I will describe a mechanism which should improve the ability to find a better solution for each GA.

The idea is based on a complete archive, which is capable of storing all visited solutions and suggesting new, unvisited solutions effectively. With its help, the GA should be able to escape from the local optima easier, or to find better solutions which lie next to already visited solutions.

Chapter 2 describes similar approaches, which were already implemented. I describe the duplicate removal method, memory-based GAs, approach using an adaptive mutation rate, GA with an archive for solving single objective problems, and GA with an archive for solving multi-objective problems. Chapter 2 shows that there have been some particular ideas of the GA archive already implemented, but none of these approaches implements it in the way that I have done here.

In the next chapter, Chapter 3, I introduce and discuss the implemented structure of the archive. Its structure is based on the trie data structure. Furthermore, I introduce the special properties and functionality of the ealib archive, such as randomized structure and handling of revisits.

Test problems, which I used for performance comparison of the GA with and without use of archive, are described in Chapter 4. Two of them are special GA problems (Royal Road function and NK landscapes problem), and the third one is MAX-SAT problem, which is well known in the area of computer science.

Chapter 5 gives us a more detailed insight into the implementation of the ealib trie archive by describing the basic packages and object structures.

Performed tests and their results are discussed in Chapter 6. In this chapter, I compare the results achieved by the standard GA to the results achieved by GAs which have used the ealib trie archive.

Conclusions and ideas for future work are written in Chapter 7.

Chapter 2

Previous work

2.1 Improvements of traditional GAs

Traditional genetic algorithms use only basic operators. They apply selection, crossover and mutation repetitively. Using these stochastic principles they try to find the best solution. Many improvements have been developed to enhance the efficiency of the GAs. Designing an improvement we must always keep in mind the universality of the improvement. It might be easier to implement an GA improvement for one one specific problem. Thus approaches which bring better results for many problems gain even more importance.

The idea of the No Free Lunch theorem (NFL) might be helpful. It shows that any improved performance over one class of problems is offset by performance over another class [61]. Many discussions have been held about contribution and relevancy of this theorem in the area of evolutionary computing.

Weinberg and Talbi [58] point at some limitations of the NFL theorem, which were described by Woodward and Neil [62]:

- Revisiting of solutions: The NFL hypothesis assumes that any solution may be visited only one time. This is not realistic, because many heuristic algorithms do not memorize all the visited solutions during the search.
- Complete space of optimization problems: The NFL result is based on the predicate that the algorithm runs on the whole space of problems. The NFL result can vary if there is a restriction on the set of optimization problems to solve.

- Overheads: If the algorithm has to memorize all the visited solutions it implies a huge overhead due to the size of search spaces, which is exponential in general. This is the consequence of the first point.

I tried to take advantage of these weaknesses of the NFL theorem. Despite that, it is a very important theorem to think about, when designing an improvement for any metaheuristic.

Now, I will describe some used and established improvements for genetic algorithms.

2.2 Duplicate removal

Using only the basic GA operators, we do not have any control over the individuals in the generation. When creating a new solution, the regular GA does not care about the other solutions in the population. This process can lead to the occurrence of duplicates in one generation. It means that in one generation we could have two or more individuals with the same genotype. There are three scenarios how a duplicate solution can be created [50]:

- a duplicate genotype appears in the first (randomly generated) population
- a child is identical with one of its parents, after applying crossover and mutation operators on them
- a child is identical with any other individual in the population

When solving one concrete group of problems, it needs not to be obvious that the duplicates are hindering. It may look like the duplicate removal preserves the genetic diversity and inhibits the solution of particular problems. It may also seem that it is at odds with the schema theorem [25]. The argument is that by allowing the duplicates it would be easier for the schemata with higher fitness to prevail in the population. However, this would lead to loss of diversity in the population. Simon Roland [50] has proven that

> "the diversity loss through duplicates is a serious weakness in the steady-state GA model."

He also shows that there is no need for the steady-state GA to add or allow duplicates to claim advantage for better schemata. They achieve satisfactory extension also without them. Therefore, removing duplicates in a steady-state population is not at odds with the bulding-block hypothesis.

The next argument against the duplicate removal principle could be the computational overhead. Let us assume that we have a population with δ solutions. In a simple implementation for each new child δ genotype-to-genotype comparisons would be needed. Naturally, comparison time becomes an unnecessary overhead in a GA with a large population and a large genotype. However, to identify duplicates we can use a hash tagging duplicate removal algorithm as it is introduced in [49]. It is an efficient way to deal with the overhead, with irrelevant memory costs. This approach operates in the following way. After generation of a new solution, it is looked up in the hashtable, which stores the whole population. If it is already inside, the new solution is thrown away and the population remains unchanged.

There are also some other approaches how to use the duplicate removal mechanism. For example Mauldin in [37] measures the Hamming distance to all members of the population and introduces a special uniqueness operator, which allow a new child to be inserted into the population only if this distance is greater than a certain threshold. This ensures even greater diversity in the population. Anyway, the practice has showed that the use of duplicate removal mechanism can have a great influence on the efficiency of the GA.

2.3 Memory based GAs

In the practice, there are also some approaches, which try to profit from the visited solutions by storing them in a long term memory. This approach is often used when the objective function changes with time, the so-called changing environments. In these problems the optimal solution is not fixed. Practice has showed that storing several best visited solutions can provide a good basis for generating new generations, even when the landscape and the position of the optimal solution has been changed [36].

Another approach presented in [18] is to concentrate on reducing the number of fitness function evaluations required by a genetic algorithm. It makes the search more effective and rapidly improves the fitness value from generation to generation.

"In the standard genetic algorithm, a new population may contain solution candidates that have already been encountered in the previous generations, especially towards the end of the optimization process. The memory procedure eliminates the possibility of repeating an analysis that could be expensive."

The fitness function evaluation is provided with the aid of the binary tree. After a new generation of solutions is created by genetic operations, the binary tree is searched for each new solution. If the solution is found, the fitness value is obtained from the binary tree. The analysis is not necessary. If the solutions is not in the tree, the fitness is obtained by an exact analysis. This new solution and its fitness value are then inserted in the tree as a new node. The following pseudocode outlines this approach:

Algorithm 1 Evaluation of fitness function using binary tree [18].

```
search for the given solution in the binary tree;
if found then
  get the fitness function value from the binary tree;
else
  perform exact analysis;
end if
```

2.4 Adaptive mutation rate

Mutation rate is often taken as a background GA operator. However, many say that a mechanism of adaptive mutation would help to improve GA performance and would help them to find the global optimum more efficiently [35, 14]. This brings the question, whether the principle of the adaptive mutation rate could be embeded into an archive for GA.

There is a variety of approaches to implement the adaptive mutation into a GA. Smith and Fogarty [54] add additional genes in the chromosome to encode the mutation rates. This approach assigns a individual mutation rate for each chromosome.

Algorithm presented in [22] uses a second GA to adapt the mutation rate. This approach optimizes the control parameters for GA through another GA in the meta-level.

Another strategy is presented by Hartono, Hashimoto and Wahde [23]. It is the Labeled genetic algorithm (LGA), which assigns a specific label to each gene. This label then traces the consistency of the gene's contribution. In this model, an individual mutation rate is assigned to each gene. A consistently fit chromosome, which is a chromosome that has higher amount of good genes, will have low mutation rate. On the other hand, the chromosomes that are less fit will have the mutation rate set higher. This is very helpful for the exploiting the promising region in the search space. The rest of the search space is explored by the less fit chromosomes. The mutation rate in the LGA is embedded as a label. The experiments of Hartono, Hashimoto and Wahde have shown that

> "...the LGA can eventually lose its diversity but is able to spread its population in a wider area compared to the simple GA in the most important period of the search process. (...) The advantage of the proposed LGA can be considered to be in its ability to adaptively regulate the mutation rate with regard to the position of each chromosome in the search space and the contribution of each gene to the fitness."

The simple GA with mutation rate $1/length\ of\ the\ chromosome$ performed also well for most of the tested problems, but the proposed LGA often outperformed it.

The adaptive mutation operator introduced in [29] provides new solution elements and maintains the best schemata in the old population at the same time.

> "The mutation operator is applied in every generation and it works with a single chromosome as follows:
>
> $$X(new\ value\ of\ gene) = (1-\theta) * X(rnd\ value) + \theta * X(old\ value)$$
>
> Where θ is an adaptive parameter, which varies between 0 and 1. When θ is very small, the new value is a completely random value. As the

value of θ increases, the new gene value is based partially on the random value and partially on the old value. Finally, when θ approaches unity, the new solution is the old one. Experiments were conducted with different values of θ and have found that the optimal value of θ is the average fitness of the population. At first generations, the average fitness value is low as we are far from the optimal solution. Thus, the adaptive mutation provides the search space with new solution elements. The average fitness is improved as the GA successfully moves towards a better solution. At last, the average fitness approaches the best fitness and the adaptive mutation operator maintains the best solution elements."

In my approach I do not use a mutation parameter, which should be adapted. The principle of non-revisiting archive guarantees automatic mutation, if it is necessary. If the algorithm wants to visit a solution, which has been visited already, it will be mutated immediately. This adds the conception of the adaptive mutation into the archive. In the parts of the search space, which are not explored is the mutation only rare. However, in the parts, where there are only a few unvisited solutions, the mutation can take place in every new generation.

2.5 GA with archive for solving single-objective problems

As it has already been mentioned, revisiting may be a serious weakness for genetic algorithms. Presented heuristics like duplicate removal and adaptive mutation rate deal with this issue in a specific way.

Yuen and Chow [63] presented an approach, which combined these ideas and adds some more improvements to get even better results. It contains a mechanism, which ensures that there are no revisits during the whole runtime and a principle, which helps to seek through the parts of the search space, where the better solutions are situated. It is the only case, where the use of complete adaptive archive for GA was presented. This archive was constructed for continuous and single objective problems. Also, only real functions were used as test functions for it.

In my approach I have implemented a similar archive, but I focused on the discrete combinatorial problems. Therefore, I had to consider the implementation of a different archive structure. Also, an exact comparison to the results of Yuen and Chow was impossible, because of the different problem definitions.

Yuen and Chow chose to use a novel dynamic binary space partitioning (BSP) tree archive for their purposes. It works in the following way:

When the GA generates a solution, the tree is accessed. A leaf node is appended to the tree, if the solution has not been visited before. In this way all visited solutions are stored. It guarantees that each solution is visited only once. If GA generates a solution, which is already in the tree, a search from the leaf is initiated. The tree then searches for the nearest neighbor solution in the search space that is not visited. In this way a self adaptive mutation mechanism is implemented. After the GA has visited all leafs of any subtree, the whole subtree may be pruned. This reduces the memory use during the runtime. Experimental results reveal that the GA with archive is superior to the standard GA with revisits in performance. Moreover, the tree archive is not so memory intensive either.

2.5.1 Storing the results

As already mentioned, the complete archive is used for the purposes of storing the results. This is based on a BSP tree.

The BSP tree is a special kind of the binary search tree, which is often used in many areas, for example in computer graphics or computational geometry. It stores the whole space, which is sequently splitted into subspaces. Each node of a BSP tree splits an area or a volume into two parts along a line or a plane.

> "The subdivision is hierarchical; the root node splits the world into two subspaces, then each of the root's two children splits one of those two subspaces into two more parts. This continues with each subspace being further subdivided, until each component of interest (each line segment or polygon, for example) has been assigned its own unique subspace."[1]

Figure 2.1 displays how the BSP tree stores the visibility of planes in the space.

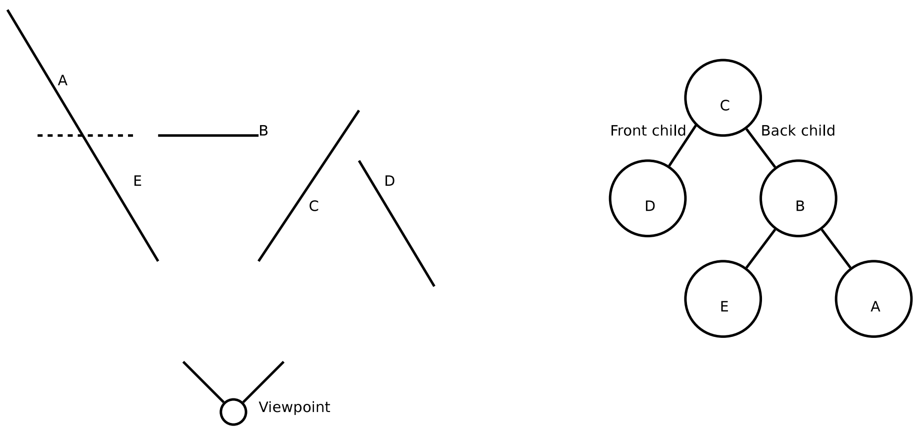


Figure 2.1: The use of a BSP tree in the computer graphics

Generally, the GA with population size δ produces a sequence of solutions. Steady state GA produces δ solutions in the beginning and 1 solution in every new generation. Each solution of these solutions is inserted into the tree. That means that everytime a new solution is generated, a new end-leaf is created and inserted into the tree (we do not consider the revisits now). The BSP tree proposed for GA has the following specific properties [63]:

- It is constructed dynamically using by inserting the solutions generated by GA. Therefore, each run of GA produces a different tree.
- It stores all solutions s visited by GA.
- For a balanced tree, the mean number of steps to decide whether a search position s_n has been visited is at most $O(\log(a^\gamma))$, where a is number of possible values for a variable and γ is number of dimensions.

For continuous problems, Yuen and Chow [63], have applied the BSP tree in the following way. They have defined a resolution d, which divided the search space and controlled the number of possible real numbers for each genome. High resolution enlarges the whole search space exponentially. Yuen and Chow [63] say that, at least for the 14 benchmark functions studied by them, the optimal fitness is only slightly dependent on the axis resolution (d). Therefore, a proper selection of the d is not a key factor for the sufficiency of the algorithm with an archive. Each solution generated by the GA is stored as a single node. In the begining we have an empty tree. Then a soluton s_1 is generated and inserted into the BSP tree as a root. By a definition of the BSP tree, each node splits its subspace into two parts.

Thus, when a solution s_2 is generated and $s_1 > s_2$, it is inserted into the left part of the tree etc.

Following example helps us to understand the construction principles of the BSP tree archive:

There is a two dimensional search space S $[1, 3] \times [1, 3]$. We have a genome with 2 genes. Possible values of the first and second gene are $1, 2, 3$. Initally the archive is empty. As a first solution, a solution $[3, 1]$ is generated and inserted into the tree. This causes that the whole space S is divided into two subspaces considering the first dimension ($1st$ gene): left subspace $S_1 = [1, 2] \times [1, 3]$ and right subspace $S_2 = [3] \times [2, 3]$. The root node $[3, 1]$ united with its subnodes gives us the whole space. It is shown in Figure 2.2.

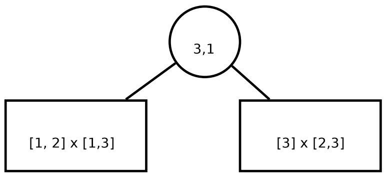


Figure 2.2: Solution $[3, 1]$ in the BSP tree

Other nodes are added when further solutions are inserted into the tree. Inserting solutions $[2, 2]$, $[2, 3]$, $[1, 2]$ and $[3, 3]$ in this order produces the tree displayed in Figure 2.3. It is obvious that each node represents one solution. Moreover, there is a difference in the trie structure, which I use. Only leaf nodes represent concrete solutions. Other nodes serve as splitting points for the search space.

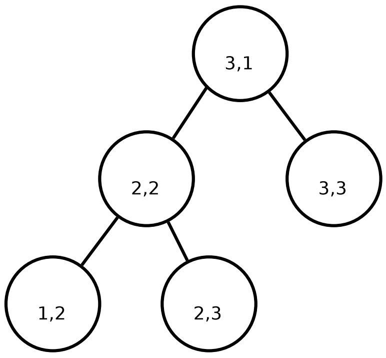


Figure 2.3: Solutions $[1, 2]$, $[2, 2]$, $[2, 3]$, $[3, 1]$, and $[3, 3]$ in the BSP tree

The size of the archive can be also reduced by pruning. Since we want to know, whether a solution was visited or not, there is no need to store the solutions, when a whole subtree under any node was already visited. Hence the entire subtree can be pruned. This helps to keep the tree compact and improves the memory usage.

If a $[2, 1]$ solution is inserted into the tree displayed in Figure 2.3, there are no other solutions in the right subtree of the node $[2, 2]$ to be explored. Thus the entire right subtree of the this node can be deleted. Instead of it there is a *Closed* flag inserted, meaning that all solutions of this subtree are stored in the archive already. The pruned subtree is shown in Figure 2.4.

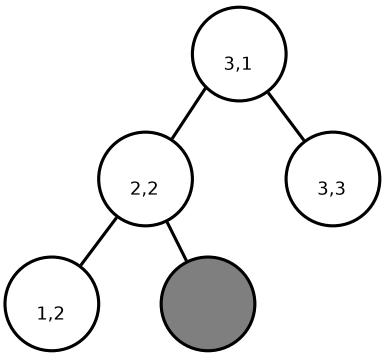


Figure 2.4: Pruned subtree under the node $[2, 2]$

2.5.2 Handling the revisits

The handling of the revisits is done in the following manner: When a solution generated by GA is identical with a solution that is already stored in the archive, a revisit has occurred. In this case the archive needs to generate a solution that was not visited before. Generally two cases of a revisit can occur [63]:

- one or both subspaces of the revisited node are not *Closed* – In this case any node from the opened subtree is chosen as a new solution. An example of this case (see Figure 2.5) is the revisit of the $[2, 2]$ solution. The algorithm steps down through the left subtree and inserts $[1, 1]$ solution instead of the revisited $[2, 2]$ solution.
- the revisited node and its subtrees are *Closed* – In this case the algorithm returns to the parent first and starts the search for the free solution in the other subtree. This situation occurs when solution $[2, 3]$ is revisited. As the nearest unvisited solution the $[1, 3]$ solution is found because it is more similar

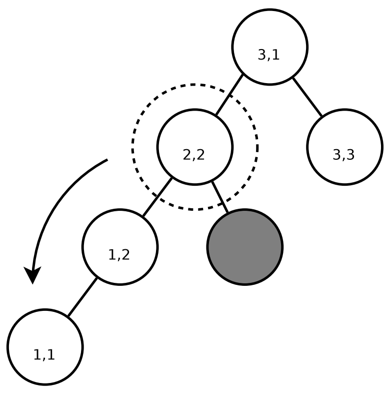


Figure 2.5: Revisit of the $[2, 2]$ solution

to the $[2, 3]$ solution than the $[1, 1]$ solution. Their second genes are identical. Figure 2.6 illustrates this case.

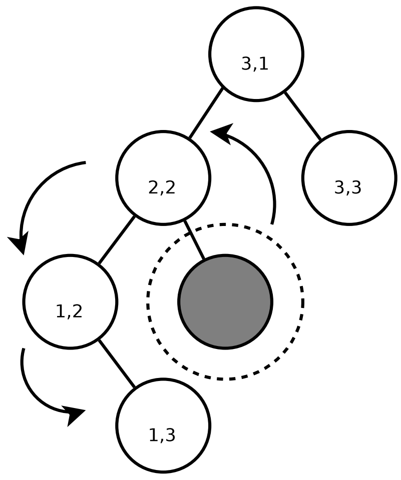


Figure 2.6: Revisit of the $[2, 3]$ solution

By using the described backtracking mechanism in the archive it is not assured that the newly suggested solution is the one with minimal distance to the revisited one. If we define the distance between two solution as the number of places in which are the solutions different, the following exaple shows the weakness of the archive solution suggestion mechanism. In Figure 2.5 the revisit of the $[2, 2]$ solution is displayed. The revisits handling mechanism suggests the $[1, 1]$ solution as the nearest, but the suggestion of the $[3, 2]$ solution could be better, because its genome differs from the $[2, 2]$ only on one place. This problem can be solved by implementing nearest neighbor search, which can add a computational time to the algorithm. In my solution I try to find a mechanism, which can make the process of suggesting new solutions globally better.

Another important aspect of the revisits handling mechanism is that no mutation operator has to be used. When a solution is revisited, it will be mutated automatically. There is also no need to have a mutation rate parameter. The more often a part of a tree is visited, the more mutations will be performed.

2.5.3 Experimental results

The described non-revisiting GA (NGA) was compared with standard GA (with revisits). For performing the tests the following set of functions, which should be minimized, was used [63]:

- Linear function $f_1(x) = -\frac{2}{10100}\sum_{i=1}^{100} i x_i$
- Spherical model: $f_2(x) = \sum_{i=1}^{7} x_i^2$
- Generalized Rosenbrock function $f_3(x) = \sum_{i=1}^{6}(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$
- Generalized Rastrigin function $f_4(x) = \sum_{i=1}^{7}(x_i^2 - 10\cos(2\pi x_i) + 10)$
- Generalized Griewank function $f_5(x) = \frac{1}{4000}\sum_{i=1}^{7} x_i - \prod_{i=1}^{7}\cos\frac{x_i}{\sqrt{i}} + 1$
- Schwefel's problem 2.26 $f_6(x) = -\sum_{i=1}^{7} x_i \sin\sqrt{|x_i|}$

The functions f_1, f_2 and f_3 are uni-modal, whilst the other three functions (f_4, f_5 and f_6) are multi-modal. The function f_1 is pseudo-boolean function, whereas the other functions are real. This shows that the concept can be applied to both real and pseudo-boolean functions. In the search space for f_2 - f_6 100 was chosen as a division number for each dimension of x. This makes the whole searchspace naturally smaller and thus more suitable for the GA. Tests were performed using standard parameters: 1-point crossover, 1-point mutation and elitism selection.

The population size was 30 chosen. Yuen and Chow [63] compared the performance of the NGA and standard GA focusing on two quantities.

- Accuracy - It is a search power within a fixed number of generations. In their case they picked up the best fitness reached in the 60^{th} generation. The results were extracted from 100 independent runs. The improvement rate of the NGA related to the GA showed that the NGA brought significantly better results than the GA.
- Probability of success - success of a run was achieved, when the algorithm met target fitness (F_g) within defined number of generations (500). The F_g was defined as the corresponding best fitness found in accuracy test. Also in this case 100 runs were performed to obtain the PoS-rate. For all test functions, the PoS of the NGA was superior to that of the GA.

Yuen and Chow conclude that the NGA brings better results due to the following:

> "When the GA comes into the basin of attraction of a local or global optimum, the chance of generating a revisiting offspring is higher. The random crossover and mutation then constitute a random, revisiting search within the basin. For a global optimum, it facilitates the location of the optimum more quickly. For a local optimum, it facilitates the complete search of the basin, so that the NGA may escape out of the basin sooner." [63]

The size of the archive was also observed. The test showed that it remains small even for large search spaces.

2.5.4 Differences between the BSP tree and trie archive

In my work I have tried to use similar principles as those of the BSP tree archive. A primary difference is that I consider binary search spaces. I could have taken the advantage of it because, the problems could be coded as binary strings. This allowed me to use another structure (trie), which could provide even better search times as the BSP tree archive did. I also focused on the handling of revisits and I have implemented different possibilities of suggesting of a new solution.

2.6 GA with archive for solving multi-objective problems

Solution archives also have already been used in context of multi-objective optimisation, although they were usually of a limited size [32]. The idea is to store set of solutions called nondominated solutions (NDS) in an archive. This is used then as a repository for all important NDS solutions and provides a pool of possible parents, which are additional to the actual population. This is a key issue in the multi-objective optimization. An improvement of this approach has been presented later in [15]. Even if this approach was the first to introduce a combination of archives in genetic algorithms, it was considered to be unnecessary to adapt it on single objective problems. The difference is that the archive for multi-objective problems does not store all visited solutions, but only several chosen ones, which should later help to drive the GA further. Even though the contribution of these papers to research of using the archive with a single-objective GA was very little, the idea was important.

Chapter 3

Trie

3.1 Motivation - why to use a trie?

My goal was to improve the ability of a GA by a solution archive to find the best possible solution without changing the properties and algorithmic flows of the given GA. As already mentioned before, the GA search space is usually binary strings. Therefore the implemented archive should be designed for discrete problems, especially to store binary vectors. The idea was to efficiently store all visited solutions, avoid revisits, and have a good and intelligent mechanism for transforming of already visited solution into a similar unvisited one. Achieving this goal was possible only through finding an appropriate structure for storing the solutions. The archive structure should sufficiently fulfill the following specifications:

- relatively low memory consumption
- fast solution insertion
- fast check, whether a solution is in the archive already
- fast transforming of an already visited solution into a similar unvisited one

Considering these requirements, I took following well-known structures into account: the hash table, binary search tree, and trie. A short description of these structures is given in the next part.

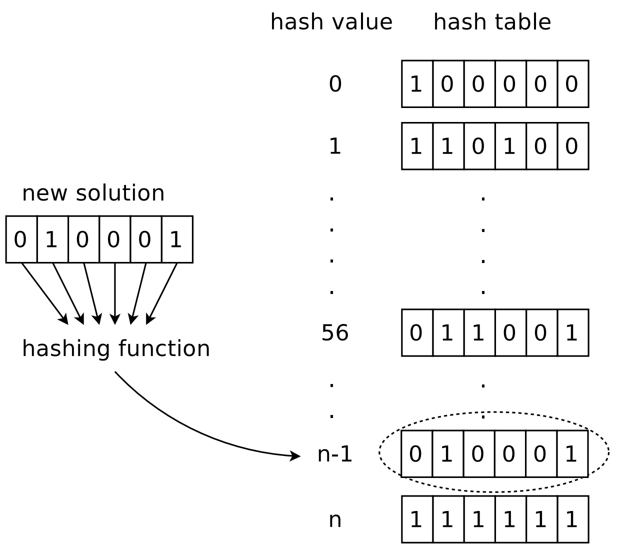


Figure 3.1: Hash table - insertion

3.1.1 Hash table

An appropriate structure for storing the solutions could be the hash table. It is a data structure in which keys are mapped to array positions by hash functions (Figure 3.1). Using a hash table with appropriate hash function, the time for the inserting or recalling of the visited solutions could be $O(l)$, where l is the length of the stored binary string [33], [12]. However, finding the algorithm for transforming the visited solutions is very complicated and could cost, in the worst case scenario, 2^l steps. Secondly, the hash table does not always provide optimal memory usage. Also, it is generally better to use a hash table, when the data is searched more often than inserted or deleted. This could be another argument against the use of the hash table as a suitable data structure for my problem. There are also additional algorithm steps needed, if we consider collisions. There are diffent strategies, how to resolve them. Most popular are chaining and open addressing.

3.1.2 Binary search tree

Trees are structures for representing certain kinds of hierarchical data. The tree consists of a set of nodes and a set of arcs. Each arc links a parent node to one of the parent's children. Every node (except the root node) has exactly one parent. It is possible to reach any node by following a specific set of arcs from the root. The simplest kind of tree is a binary tree where each parent has at most two children.

In the binary search tree, each left pointer points to nodes containing elements that are smaller than the element in the current node and each right pointer points to nodes containing elements that are greater than the element in the current node.

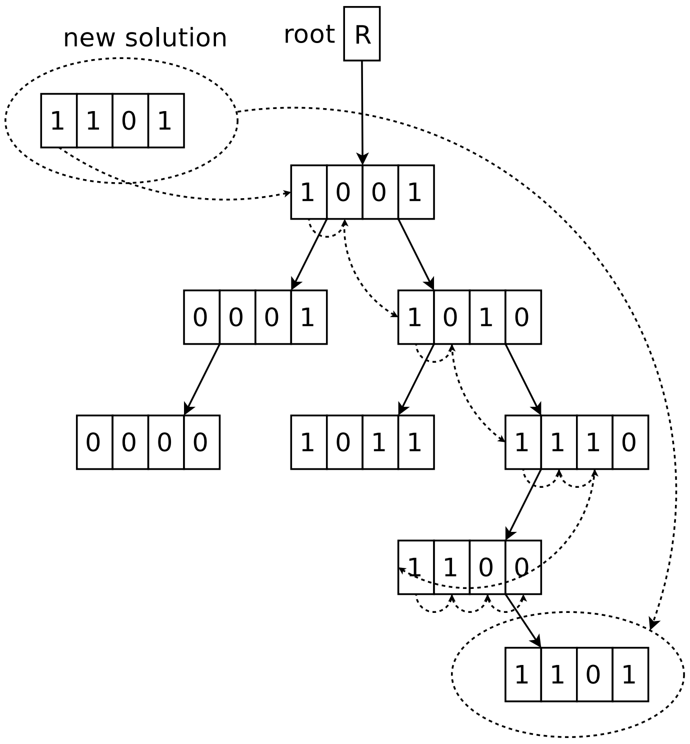


Figure 3.2: Binary tree - insertion

Two binary trees with same elements can be different. It depends on the order of insertion of the elements. Some insertion orders can cause the binary tree to be unbalanced. Therefore a tree type was introduced which contains a rebalancing mechanism. It is called AVL-Tree and it was introduced in [2].

The binary search tree has several important properties of great practical value. One of these properties is the searching speed. Let us consider now the cost of insertion into the tree. Again we insert a binary string with length l into a tree with n nodes. If the tree is perfectly balanced, the cost of inserting is proportional to $l.\log_2(n)$ steps. Rebalancing the tree after an insertion may take only a few steps, but at most it will take $\log_2(n)$ steps. Thus, the total time is of the order $O(l.\log_2(n))$

[8], [33], [12]. For the data retrieval, modification and transformation are the costs of the same order.

Comparing the binary tree with the hash table, the binary search tree as a structure for storing the solutions may seem to be a better choice because of faster implementation of the transformation mechanism and better memory usage. However, considering that the solutions in the GA are coded as binary strings, it forced me to speculate about other structures. There is no need to store all solutions or to have the whole solution key stored in each tree node. This is a serious weakness of the binary search tree, when compare it to the trie. With high amounts of stored solutions, the binary tree demands high amounts of memory usage. However, the implementation of the pruning mechanism, which saves a lot of memory and time, could be complicated because of rebalancing. The time needed to suggest a new unvisited solution, which has to be similar to a concrete revisited solution, could be shorter when using another structure.

3.1.3 Trie

A trie (from retrieval), is a specific tree structure useful for storing strings over an alphabet. It is typically used to store large dictionaries of natural words in spell-checking programs and in natural-language understanding applications. Its construction is well designed for the determination of whether a given word is stored in the trie or not. The idea of the trie is that all strings sharing a common stem or prefix hang off of a common node [17].

A trie example is displayed in Figure 3.3. We have a four-character alphabet $\{a, b, c, d\}$. From these characters we are able to create words and store them in the trie. Each of its nodes can store two types of information for each character. The first information is whether the character is also an end of a word. The second is a pointer to the successor node. In the example trie the following words are stored: $a, ba, bda, bdc, c, cb, cba, cc$.

A problem for the trie structure can be the effective use of memory. For example, in Figure 3.3 are nodes which contain many NULL-pointers. This causes the whole structure not be as compact as possible. Several approaches have tried

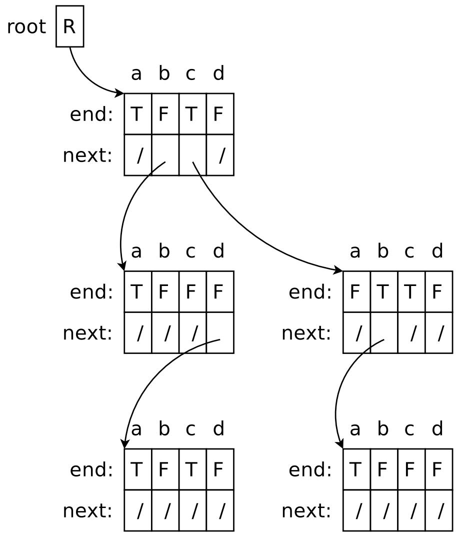


Figure 3.3: Trie example

successfully to solve this problem. The linked trie, indexed trie, packed trie and trie with a suffix compression were developed.

In contrast to a binary search tree, each node of a digital search trie can only hold one character of the keyword. If the keyword is longer than one character, then the node containing the first character of the keyword points to the node containing the next character, and so on. The height of the search trie equals the length of the longest keyword stored. For the inserting and searching a solution $O(l)$ steps are needed, where l is the length of the keyword. This is slightly faster than in binary trees.

I needed to adapt the trie structure for binary problems only, as I have tested it only with functions where the solutions can be encoded as binary strings. Seeing each character of the binary string as a character of a word, the classical trie structure looked like a good starting point. Each node of the trie could possibly require a large amount of memory storage because there could be many NULL-pointers in each node. But in my case I have only two pointers in each node (0,1). Additionally, my trie always stores strings of the same length, which makes the end-flags in each node unnecessary and also saves memory. These attributes eliminate a serious disadvantage of tries - memory usage. Having an opportunity to prune the subtries additionally, it is possible to achieve an ideal memory usage.

The pruning mechanism would additionally help to transform a visited solution into an unvisited one more quickly because it would be easier to determine which parts of the search space are not yet explored.

3.1.4 Comparison

Let us consider once again all pros and cons of the discussed structures.

An important decision factor is speed. Combining the searching, inserting and suggesting of unvisited solutions can be done very efficiently with the trie structure. Also, the usage of memory in binary string tries is better than the memory usage of the other two structures. It could be even better when enhanced by a pruning algorithm. All these arguments are summarized in Table 3.1. Additionaly the trie provides an effcient storing of solutions and their keys. Because of these attributes, I have decided to implement a special structure derived from the trie as an archive for storing all solutions produced by the GA. It is called ealib trie.

Table 3.1: Comparison of the data structures

structure	memory	insertion	check	suggestion
hash table	$O(l.n)$	$O(l)$	$O(l)$	$O(2^l)$
binary tree	$O(l.n)$	$O(l.\log_2(n))$	$O(l.\log_2(n))$	$O(l.\log_2(n))$
trie	$O(l.n)$	$O(l)$	$O(l)$	$O(l)$

3.2 Trie with genetic algorithms

An important question to answer is the following: How will the ealib trie cooperate with a genetic algorithm?

Figure 3.4 describes the GA–Trie cooperation principle. The GA can be seen as a separate module which generates solutions in a specific way. The parameters used for the GA configuration are not relevant for the trie. After creation of a solution, the solution is forwarded to the trie. As the trie accepts the solution, it checks whether it is included in the archive already. This is done with the cost of $O(l)$ steps, where l is the length of the inserted binary string. If the solution is not in the archive already, it is simply inserted into the trie. The effort for inserting is again, $O(l)$. On the other hand, when the solution is in the trie, it comes to a $revisit$. Handling of the $revisit$ can be done in several ways. These are described in Section 3.4.4. It is important to find a good balance between the quality of the changed solution and the effort needed to change it. After inserting or altering a solution, it is sent back to the GA module and then handled as usual.

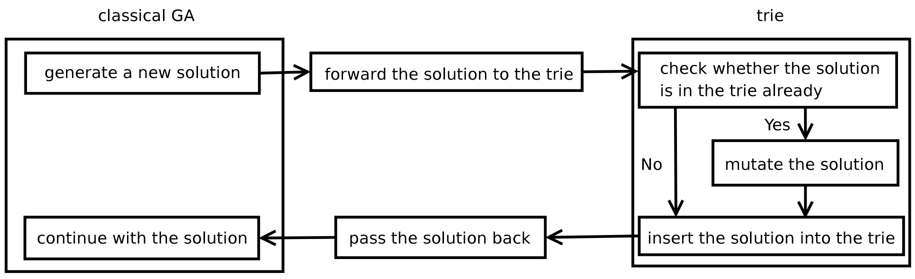

classical GA
trie
generate a new solution
forward the solution to the trie
check whether the solution is in the trie already
Yes
No
mutate the solution
continue with the solution
pass the solution back
insert the solution into the trie

Figure 3.4: The cooperation between GA and trie

It is also possible to describe this interaction in a mathematical way, as it is done in [63]. The GA module generates a sequence of solutions. $sq = (s(1), s(2), ...)$ and then passes it to the trie. The purpose of the trie is to return the amended sequence $sq' = (s(1)', s(2)', ...)$, where two solutions $s(i)'$ and $s(j)'$ satisfy $s(i)' \neq s(j)'$ unless $i = j$.

3.3 General description of the ealib trie

In this chapter, I will try to describe the structure of the ealib trie in detail. Here, only binary strings with the same length l are inserted into the archive . Thus, the non-empty trie always has l levels.

The $root$ represents level 0 and points at the 1^{st} level node. Each node consists of 2 pointers. The left pointer represents a 0. Concerning the 1^{st} level node, all binary strings stored under this pointer begin with a 0. In the same way, the right pointer represents a 1 and all binary strings stored under this pointer begin with a 1. Each node in level n has two pointers, which represents a 0 or a 1 on the n^{th} position in the binary string. The node on the l^{th} position (the last position of the binary string) is a special case. This node does not contain any pointer. It can contain only the $endflag$ (X), which means that the 0 or 1 on the l^{th} position (depending on whether it is stored under the left (0) or right (1) part of the node) is included. Figure 3.5 shows how the solution 010 is stored in the trie. Another important thing to introduce is the $emptyflag$ (/), which means that a solution, or the whole subtrie, was not yet explored. This leaf is inserted in all places where a single solution (in Figure 3.5, the solution 011), or a whole subtrie (in our case, subtrie 1__ or 00_), is not present in the trie.

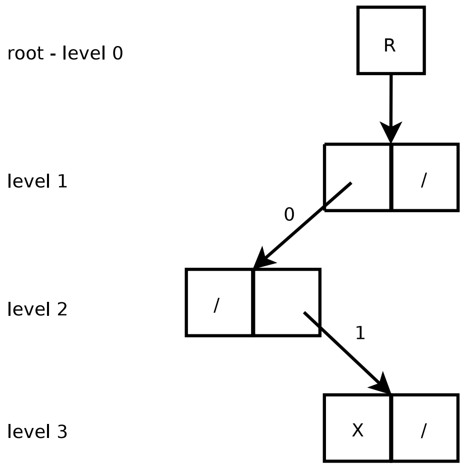


Figure 3.5: 010 solution in the trie

Figure 3.6 displays the same trie as a plane with the solution (010) inserted.

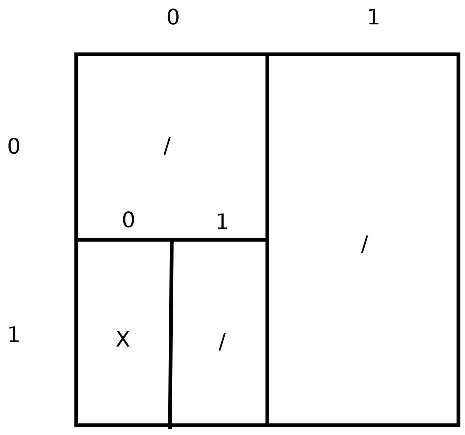


Figure 3.6: 010 solution in the trie - plane

The search algorithm that checks whether a solution is stored in the trie or not works as follows. Starting with the first node, it steps down the trie according to the binary string. On each level n, a check is performed. If the character on the n^{th} position the binary string is 0, the algorithm steps down to the left subtrie and vice versa. The algorithm continues recursively until it comes to the $endflag$ or

$emptyflag$. When we consider the trie without pruning, the $endflag$ can only be on the bottom of it (l^{th} level). Therefore, it needs exactly l steps to reach the solution, which is already in the trie. If the solution is not in the trie, it can happen that the algorithm finds the $emptyflag$ sooner. This means that it needs less then l steps if a whole subtrie is not included in the trie. Written in the O-notation, to find out whether a solution is in the trie or not costs $O(l)$ steps. In Figure 3.7, we see that we need 3 steps to find the 010 solution, but only 2 steps to find the 001 solution, because the whole 00_ subtrie does not exist.

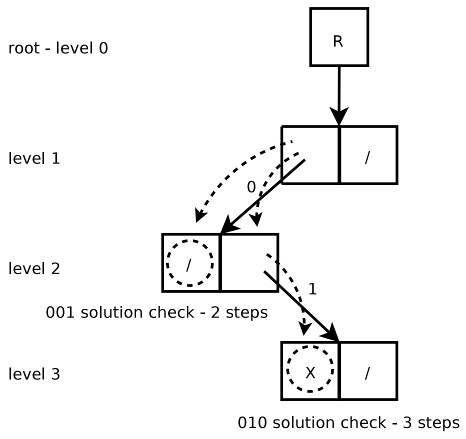


Figure 3.7: search for the 010 and 001 solution

The inserting of a solution into the trie always costs l steps, where l is the length of the inserted binary string. This is because (when we do not consider pruning) all $endflags$ are stored on the bottom of the trie (l^{th} level) and the insertion algorithm needs l steps to get there.

The concrete implemented insertion algorithm enhanced with the pruning mechanism is described in Section 3.4.2.

Using this structure we can store all visited solutions in the memory in a relatively efficient way. Another benefit of the trie is very fast searching of the visited solutions. In the next section we will have closer look at the special properties of a ealib trie for genetic algorithms.

3.4 Specific functionality of the ealib trie

3.4.1 Pruning of the subtries

An important principle which helps us to keep the trie smaller is the pruning of whole subtries. If all solutions under one node are marked as visited, they will be pruned (deleted) and the whole node will be marked as visited. This is possible because we only need the information of whether a solution was visited or not. We do not have to keep specific information for the solutions, such as each solution's fitness. Thus we know if a node is marked as visited, all solutions which belong to this node are visited. This pruning principle can help us to keep the trie smaller and the searching time shorter.

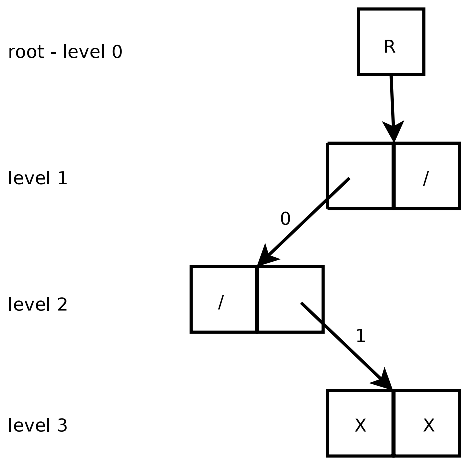


Figure 3.8: 010 and 011 solution in the trie

This is demonstrated in my example Figure 3.8, where the solution 011 is added into the trie.

By backtracking, it can be elicited that the whole 01_ subtrie has been visited. Consequently, the whole subtrie will be marked with $endflag$ (X) (Figure 3.9). The normal node with the 0 and 1 pointers will then be replaced with an $endflag$ (X).

Two benefits of pruning are obvious in this example. You can note that the whole trie now consumes less memory. The second benefit is that instead of 3 steps, we need only 2 to figure out whether solution 010 or 011 is in the trie or not. This helps us to keep the trie access times shorter.

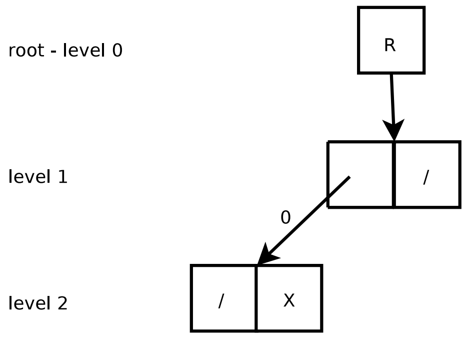


Figure 3.9: 010 and 011 solution in the trie pruned

3.4.2 Algorithm for the inserting of the solution into the trie and deleting of the completely visited parts

First, I will introduce some variables which are used in my algorithms. This is necessary for the further understanding of the algorithms:

- binary string s - solution which is inserted or modified
- pointer p - pointer on a certain node of the trie
- $p.next[2]$ - array of two pointers which point to the left or right subtrie, they can also contain COMPLETED flag or NULL
- integer pos - indicates to which position of the solution the p-pointer is pointing
- COMPLETED - if any pointer points to this flag, it means that a concrete solution, or a whole subtrie, was inserted into the trie (the same as $endflag$ (X))
- NULL - if instead of any pointer there is a NULL stored, it means that the concrete solution or a whole subtrie were not inserted into the trie yet (the same as $emptyflag$ (/))

Now I will describe, how the algorithm for the inserting of a solution into the trie works (Algorithm 2). First, the method *try_insert()* with the new solution as a parameter is called. This steps down the trie progressively, searching for the solution. If a NULL-pointer is found, which means that the solution is not yet in the trie, it is inserted, deriving benefit from the previous search. This is a task for the *insert_with_position_found()* method. The method *try_insert()* returns $true$ afterwards. On the other hand, when the search ends with a revisit, $false$ is returned.

In this case, the solution is changed into an unvisited one and inserted into the trie with the same *insert_with_position_found()* method. Suggesting of unvisited solutions is discussed in Section 3.4.3 and in Section 3.4.4.

Algorithm 2 trie - *try_insert(solution s)* method

```
// checks whether a solution is in the trie already
// calls insert_with_position_found() method and returns $true$ if it is not
// returns $false$ if the solution is in the trie already

$p =$ root pointer;
for $pos$ = 0 to $l$ do
  if $p ==$ NULL then
    create new node;
    $p = p.next[s[pos]];$
  else if $p ==$ COMPLETED then
    return $false;$
  else
    $p = p.next[s[pos]];$
  end if
end for
$insert\_with\_position\_found(pos, p, s);$
return $true;$
```

After the method *insert_with_position_found()* inserts the solutions into the trie it checks whether its neighbour has also been visited. If so, pruning of the whole subtrie takes place (Algorithm 3). The same step is then done recursively with all parent nodes. This pruning mechanism helps us to keep the memory usage of the trie optimal.

3.4.3 Revisits and the suggesting of an unvisited solution

The second most important characteristic of the ealib trie is that you can never insert two of the same solutions in it. If you try to do so, a revisit will occur. The ealib trie then changes the revisited solution automatically into another solution, which has not been visited yet. It is immediately inserted into the trie and returned back to the sender. The suggestion of an unvisited solution can be done in several ways. Discussion about these techniques is done in Section 3.4.4. The main problem is that the unvisited solution, which is nearest in the trie, does not have to be the one

Algorithm 3 trie - *insert_with_position_found(pos, p,s)* method

```
// inserts a solution into the trie
// and prunes completed subtrie of the trie

$p =$ COMPLETED;
$p = p.parent;$

while $pos > 0$ do
  if $p.next[(1 - s[pos])] ==$ COMPLETED then
    $p =$ COMPLETED;
    $p = p.parent;$
  else
    exit for;
  end if
  $pos = pos - 1;$
end while
```

which has the minimal Hamming distance to the revisited solution. However, finding an unvisited solution with a minimal Hamming distance to the revisited solution can cause unnecessary computational costs.

I can illustrate it in following example. Here, we have the same trie as in Figure 3.5. There is only one solution (010) inserted into it. Let us assume that another 010 solution is inserted into the trie. Now a revisit occures, as it is shown in Figure 3.10.

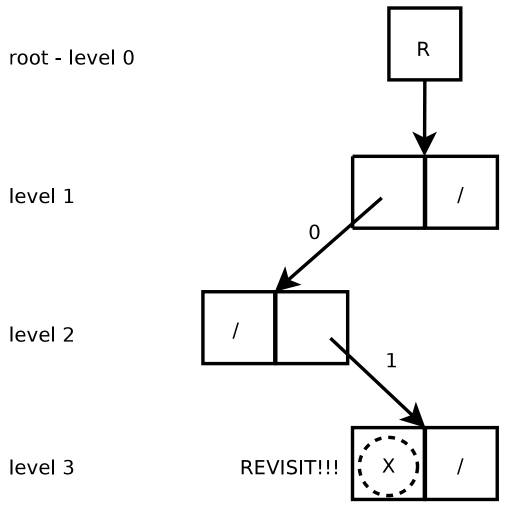


Figure 3.10: attempt to insert the 010 solution into the trie for the 2^{nd} time

A primitive suggestion works in the following way. It finds the next unvisited solution by backtracking. In our case the closest solution is the 011 solution. After that, it is automatically inserted into the trie and returned back to the sender (Figure 3.11). The tree is also pruned immediately, as is shown in Figure 3.9.

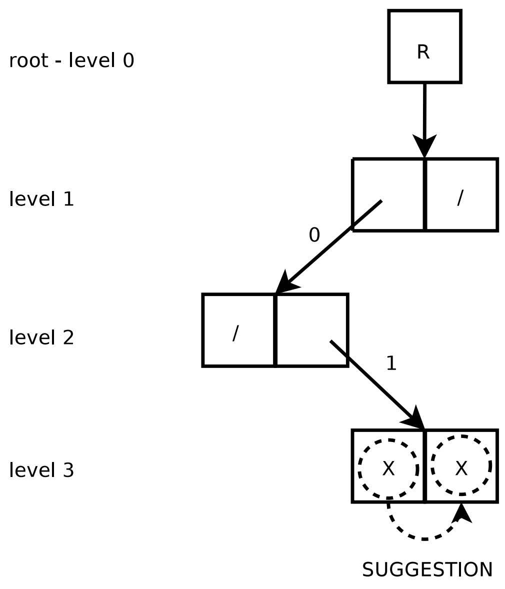


Figure 3.11: 010 solution revisit, suggestion of 011 solution

The aforementioned problem in suggesting solutions can be explained by the following. When we try to insert a 011 solution into the trie shown in Figure 3.9 again, the primitive algorithm finds a revisit and suggests, with the help of backtracking, a 000 solution as the next possible unvisited solution. This has a Hamming distance of 2 to the ancestral solution because it differs from it by 2 places. But, as we see, there are also two unvisited solutions with a Hamming distance of 1 (001 or 111). If the trie held more solutions it could cause some problems in the general performance of the GA.

3.4.4 Algorithms for the suggesting of new solutions

When the GA algorithm inserts certain solution into the trie for the second time, a revisit occurs. The handling of these revisits is crucial for the trie. This avoiding of the revisits brings us two advantages.

First is that the number of visited solutions within a certain number of generations is bigger. Logically, when I explore a new solution instead of exploring a revisited solution, the total amount of explored solutions is higher than in an algorithm with revisits.

The second advantage is in the saving of the computational time needed for evaluation of visited solutions. We do not need to evaluate any solution twice. This issue can bring significant benefits when we have a problem which is very expensive for computation.

After a revisit occurs, the method *change_to_unvisited_fast()* is called. It has two obligatory parameters: s, which stores solutions that should be changed and *modelSol*, which represents a template for the new created solution to be modeled after.

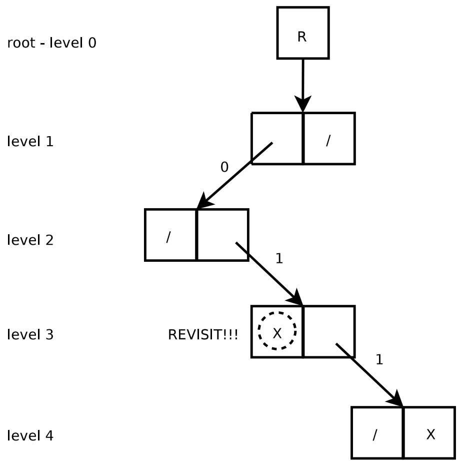


Figure 3.12: The solutions 0100, 0101 and 0111 in the trie, 0101 revisited

To illustrate how the revisits are handled I will introduce some examples. In all of the following examples, the same initial situation will be considered. There is a trie, displayed in Figure 3.12, which stores three solutions (0100, 0101 and 0111). Let us say that the best found solution is the 0111 (this is needed for the transformation algorithm, which uses the best solution). A revisit of the 0101 solution has occured, and the trie transforms it into another yet unvisited solution. This transformation depends on the chosen suggestion algorithm. In my project I implemented these three algorithm types for the transforming of the solution after a revisit into an unvisited one:

- Default suggestion
- Suggestion changed in a random place

- Suggestion using the best solution as a template

Default suggestion

Algorithm 4 trie - default suggestion method

```
var s // solution, which should be changed
var modelSol;

$pos = depthsOfUncompleted.back();$
$p = pointer\ to\ the\ node\ on\ s[pos];$

while $pos < s.length$ do
  if $p.next[modelSol[pos]] == NULL$ then
    create new node;
  else if $p.next[modelSol[pos]] == COMPLETED$ then
    $s[pos] = (1 - modelSol[pos]);$
    $p = p.next[s[pos]];$
  else
    $s[pos] = modelSol[pos];$
  end if
  $p = p.next[s[pos]]$
  $pos + +;$
end while
$insert\_with\_position\_found(pos, p, s);$
return $true;$
```

The default suggestion works in the following way (Algorithm 4). *ModelSol* is the same as solution that is stored in s. This means that the new solution should be as similar to s as possible. All nodes where there is a possibility to find an unvisited solution are stored in the array *depthsOfUncompleted*. In the default suggestion algorithm, the last node of this array is chosen as a deviation point. The solution is mutated in this place. Afterwards, the algorithm tries to follow the pattern of the ancestral solution wherever possible.

A concrete example is shown in Figure 3.13. After a revisit of the 0101 solution has occured, the algorithm searches for the first uncompleted subtrie. The transformation into the 0100 solution is not possible, because it is in the trie already. So, the first transformation is done at the 3^{rd} level. The algorithm now has the 011_ solution and continues to search for an unvisited solution in the 011_ subtrie. First it tries to transform it into the 0111 solution, because this solution is more

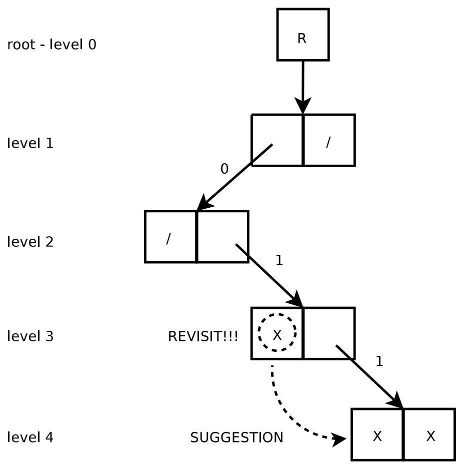


Figure 3.13: Default suggestion

similar to the 0101 solution than the 0110 solution. The 0111 solution differs from the 0101 in only one place, however the 0110 solution differs from it in two places. But the 0101 solution is in the trie already, so the algorithm must take the second possibility and, finally, transform the revisited solution into the 0110 solution.

Suggestion changed in a random place

This suggestion mechanism is very similar to the default suggestion mechanism, but instead of choosing the last node from *depthsOfUncompleted* array as the deviation point, it chooses a random node. Afterwards, the algorithm continues in the same way as the default suggestion algorithm.

In Figure 3.14 the 2^{nd} level node is chosen as the deviation point. The algorithm performs the transformation on the 2^{nd} place of the binary string and steps down the 00__ subtrie. Then it tries to add all the remaining characters from the ancestral solution, which are 0 and 1 in our case. So the revisited 0101 solution will be transformed into the 0001 solution. Note that it differs from the 0101 solution in only one place.

Another example of the random suggestion algorithm is shown in Figure 3.15. Here we have the same situation, but this time the 1^{st} level node was chosen as the deviation point. The algorithm performs the transformation on the 1^{st} place of the binary string and steps down the 1___ subtrie. In this case, the revisited 0101 solu-

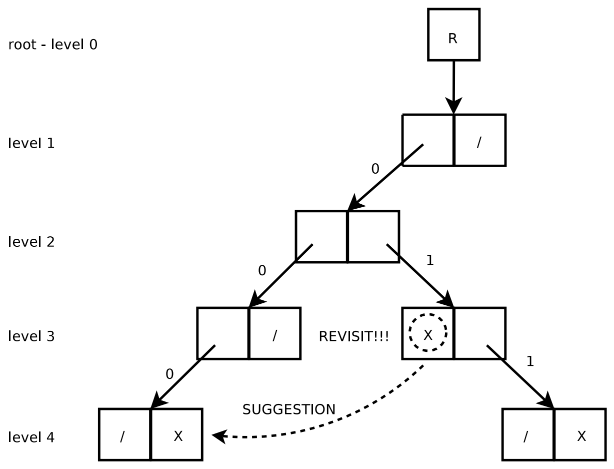


Figure 3.14: Random suggestion

tion will be transformed into the unvisited 1101 solution. Note that it again differs from the 0101 solution in only one place.

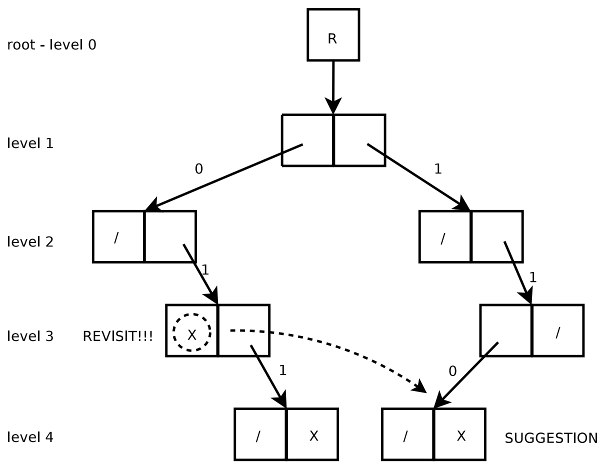


Figure 3.15: Random suggestion 2

Comparing random and default transformation algorithms, the main benefit of the random transformation algorithm is that there is a better chance to find a solution which has a shorter Hamming distance to the ancestral solution.

Suggestion using the best solution

The next suggestion algorithm uses the same principles as the random suggestion algorithm. The only difference is that instead of the most recently found solution, the best found solution is given to the $modelSol$ parameter. This means that the suggested solution tries to be more similar to the best found solution than to the ancestral solution.

An example of this algorithm is displayed in Figure 3.16. The 2^{nd} level node was randomly chosen as the deviation point. Afterwards, the characters 11 were added into the 00__ subtrie, because, as mentioned previously, 0111 was the best found solution.

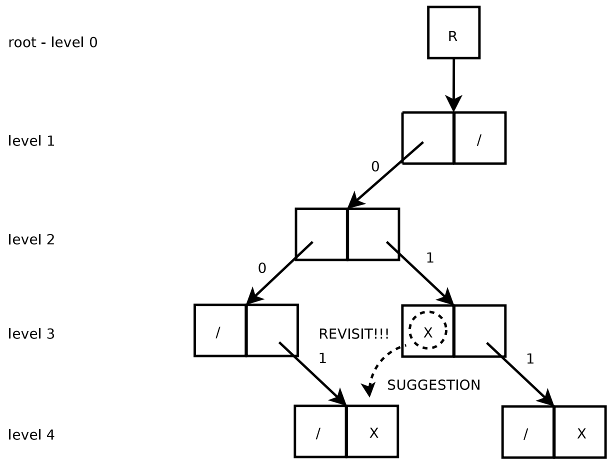


Figure 3.16: Suggestion using the best solution

Both modifications of the default suggestion algorithm (suggestion changed in a random place and suggestion using the best solution) were meant to help the default algorithm to get out of the local optima. In Chapter 6 the presented suggestion mechanisms are evaluated with different test problems.

In the following section I present another mechanism, which helps to avoid becoming stuck in the local optima. However, it is not based on changing the algorithms, but its base consists in altering the trie stuctures.

3.5 Structure implementation

In this section I would like to introduce the concrete structure implementation of the ealib trie. First, I introduce the implementation of a normal structure. In the second subsection we will have a closer look at a randomized ealib trie structure, which helped me to achieve better results in the tests.

3.5.1 Normal trie structure

As I have already described in Section 3.3, the whole ealib trie implements the concept of the classical trie. The basic building unit of the tree is a node named *TrieNode*. Each node has two pointers. These can point to the next node, or they can end in a NULL-pointer or COMPLETED-pointer. NULL-pointer means that a solution or a whole subtrie has not yet been visited. On the other hand, COMPLETED-pointer means that a solution or a whole subtrie has already been visited. Each node on the n^{th} level represents a gene on the n^{th} position.

Using this classical structure, the tests have discovered, in my opinion, a serious weakness. Note that the method *change_to_unvisited_fast()*, which suggests unvisited solutions, has the disadvantage of placing a significant bias on certain genes to be changed more frequently. If the ealib trie encounters a revisit, the first gene which is attempted to be changed is the gene on the last position. This causes the "mutations" of the chromosomes to be done more often on the last positions. This can lead to a case where we can observe rigid structures of the visited solutions in the search space. It is often coupled with solution diversity loss and finally, what is most troubling, loss of GA performance. An example of this rigid structure in the search space is shown in Figure 3.17. All visited solutions of this search space are listed in Table 3.2. We can clearly see that the genes in the positions 6, 7 and 8 are changed more often than the others. The suggestion of new solutions, which follows these rigid structures can therefore also cause the identified unvisited solution to be relatively far away. For instance, a revisit of the 11000000 solution would suggest the 11000111 solution as the next unvisited solution. Note that 11000111 differs from 11000000 in three places. However, there is still an unvisited solution, which differs from it in only one place (11001000). Therefore, more sophisticated techniques were developed. These techniques are described in Section 3.5.2 and Section 3.4.4.

3.5.2 Randomized trie structure

This section describes a special randomized structure of the ealib trie. It defines different geneorder for each chromosome sequence. That means that the n^{th} chromosome gene is no longer stored in the n^{th} level of the trie, but that the order of

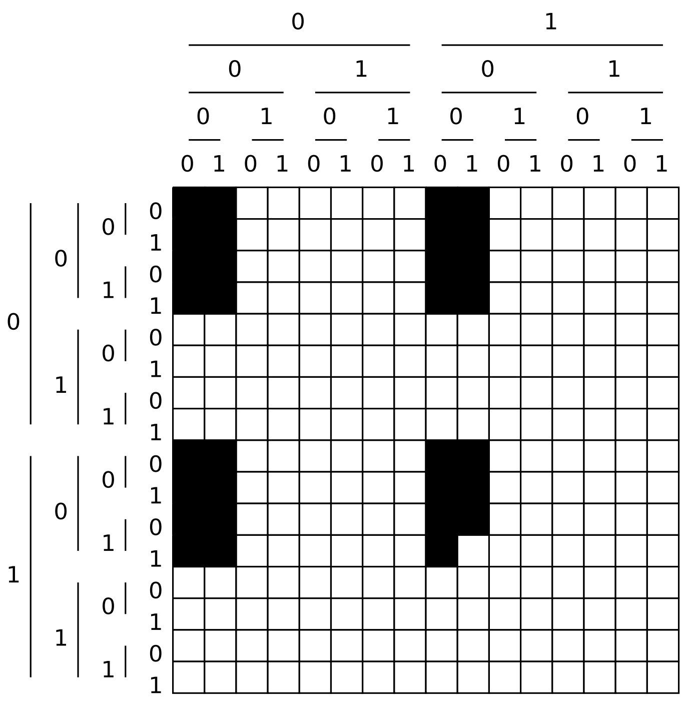


Figure 3.17: rigid structure of visited solutions

Table 3.2: Visited solutions of the searspace shown in Figure 3.17

00000000	01000000	10000000	11000000
00000001	01000001	10000001	11000001
00000010	01000010	10000010	11000010
00000011	01000011	10000011	11000011
00000100	01000100	10000100	11000100
00000101	01000101	10000101	11000101
00000110	01000110	10000110	11000110
00000111	01000111	10000111	

the genes is random. Figure 3.18 displays this condition.

The order of the genes is determined with help of the pseudorandom function *random_intfunc()* and the function *determine_next_gene_for_branching()*. These two functions always return the same randomized order of genes for a given chromosome sequence. There is also another special characteristic. A path to a subtrie always has the same geneorder. Only the subtries are randomized. Concerning our example, we can see the following. The randomization is constructed so that all pseudorandom chromosome sequences stored in the trie displayed in Figure 3.18

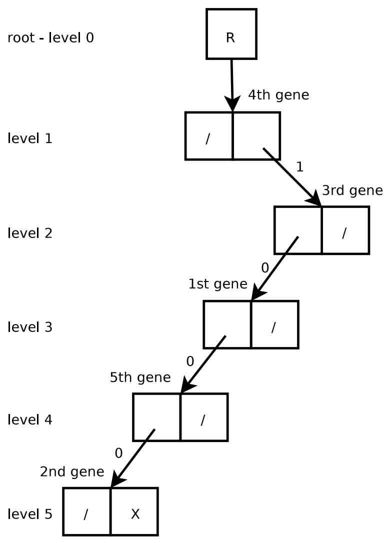


Figure 3.18: 01010 solution in the randomized trie structure

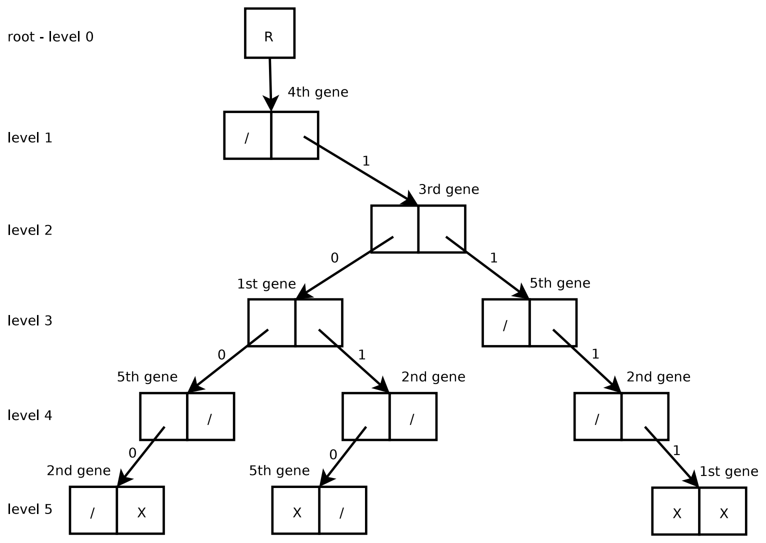


Figure 3.19: solution 01010, 10010, 01111 and 11111 in the randomized trie structure

will begin with the 4^{th} gene. All genes that conform to the 0*01* pattern have the random genorder with a 100** pattern. The subtrie of a 100** geneorder can have the last two genes randomized in any way. To better illustrate this issue, I have added three more solutions into the trie (10010, 01111, 11111). Figure 3.19 shows the randomized order of the genes.

The revisiting of the solutions fills up the lower levels of the trie. Note that in the randomized trie structure, revisiting no longer produces the rigid structure that the normal trie did. Comparing Figure 3.20 to Figure 3.17, we can clearly see that there are no such patterns of visited solutions in the search space as before. This is even more obvious when displaying a search space with longer chromosomes.

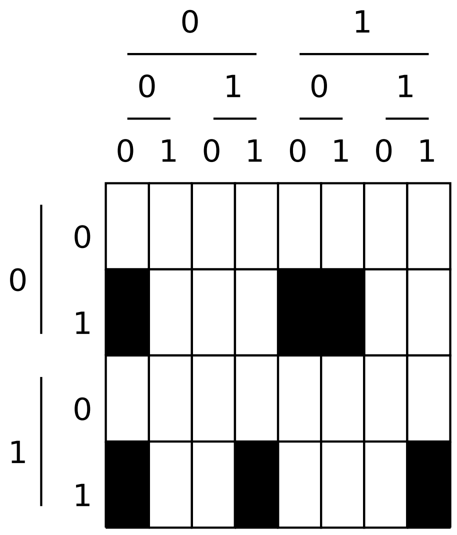


Figure 3.20: nonrigid structure of visited solutions

The implementation of the randomized trie structure adds the following attributes to the normal structure:

- vector *geneorder* – Stores indices of genes in the order they are used on the current search path in the trie. After these genes, all others that are so far unused follow in an arbitrary order.
- vector *rand_order_pattern* – Vector of integers used as an input parameter for the randomized gene ordering.
- unsigned *randorder_seed* – Seed value for random odering of genes.

Using the normal trie structure, the vector *geneorder* contains the default order of genes (12345). The *geneorder* for the solution displayed in Figure 3.18 is (43152). As already mentioned, for each chromosome a unique genorder is initiated. For this purpose there are the following methods in combination with the pseudorandom function *random_intfunc()*:

- *determine_next_gene_for_branching()* – Helper function which determines the index of the next gene to branch over to. The index is returned and geneorder (eventually) updated.

- *geneorder_reset()* – Resets gene indices not yet used, starting from position d to standard order. Used when resetting the search path to a prior position.

Using these methods allows us to have randomized trie structure without having significant performance loss. The initializing of the structure *rand_order_pattern* and the calling of the method *determine_next_gene_for_branching()* is done in the method *try_insert()*, which is described in Section 3.4.2.

3.6 Main algorithmic flows

3.6.1 main() method

The whole program starts with a simple *main()* method shown in Algorithm 5. A set of parameters defines which problem should be solved and which kind of GA will be called. Depending on the solved problem, the specific chromosome class will also be instanced.

Algorithm 5 main() method

```
generate a template chromosome of the problem specific class
generate a population of such chromosomes
generate the GA
run the GA until termination condition
write result and statistics
```

3.6.2 run() method of the steady-state GA

I have used a steady-state GA for my test runs. Thus Algorithm 6 describes the *run()* method of the steady-state GA. It is important that a chromosome method, *locallyImprove()*, is called from *run()* method. This *locallyImprove()* method then communicates then with the ealib trie.

Algorithm 6 steady-state GA - run() method

```
checkPopulation
if NOT terminate condition then
   {create a new solution}
   select solution p1 from the population
   select solution p2 from the population
   perform crossover with p1 and p2 to generate tmpSolution

   {insert the new solution into the trie, respectively change it into an unvisited
   one}
   perform locallyImprove() with tmpSolution
   put the tmpSolution into the population
end if
```

3.6.3 Insertion into the trie

Further, we will have a closer look on the cooperation between the ealib trie and the GA. The locallyImprove() method of the chromosome is called within the *run()* method of the GA, as we have seen it in Algorithm 6. The appropriate trie method *locallyImprove_move_to_unvisited()*, which is shown in Algorithm 7 is called from the chromosome *locallyImprove()* method.

Algorithm 7 trie - locallyImprove_move_to_unvisited() method

```
try to store current solution
if solution is in the trie already then
   change to unvisited solution
   {several strategies of changing of the solution can be used}
   if the whole trie is completed then
      terminate
   end if
end if
```

In the steady-state GA a new solution is created in every generation. This is then forwarded to the ealib trie. If the solution is not in the trie (i.e. it has not yet been visited), it is inserted into the trie normally. If a revisit occurs, the automatic mechanism changes the solution and inserts it into the trie. When all solutions have been visited, the whole program is terminated.

Chapter 4

Test problems

4.1 Overview

To test the new approach I needed to find proper test functions. First, I optimized these functions using the GA with the archive. Afterwards, I optimized the same functions using the same GA, but without the archive. Consequently I compared the results.

I needed to choose the test functions carefully to obtain relevant results. Naturally I searched functions which solutions could be coded in binary strings. The first criterion was that the use of the test function in the area of GAs should be relatively common. The second criterion was practical relevancy and the possibility to parametrize the function in different ways, to obtain more general results.

Considering these requirements I chose following functions for testing:

- Royal Road function
- NK landscapes
- MAX-SAT problem

4.2 Royal Road function

The Royal Road function was specially designed for evaluating GAs by Mitchell, Forrest and Holland [40]. Their intention was to create a specific function which could help to better understand and examine the role of the crossover operator and the building-block hypothesis in GAs. These are the key features which lead the GA to successful performance. Their other goal was to better understand the interference between fitness landscape features and the performance of the GA. Therefore the Royal Road function defined a fitness landscape, on which the GAs should carry out better results than other approaches. An opposite class of functions was defined earlier by Goldberg in [20, 21] and is called GA-deceptive functions. However, to define whether a function is deceptive or non-deceptive for the GA can be complicated. To make it clearer, Watson, Hornby and Pollack explored building-block interdependency in [57].

The Royal Road function is defined for binary strings with fixed length. There is a defined set of schemata $S = s_1, s_2, \ldots, s_n$, which can be indentified in the string and the objective function, which should be maximized. The objective function is defined as

$$f(x) = \sum_{\forall s \in S} c_s \sigma_s(x)$$

where $x \in X$ is a binary string and c_s is the value assigned to schema s. All schemata are stored in the schema table. The binary string belogs to schema s, if it contains several consecutive ones on the defined positions. The amount of necessary ones is equal to the defined *rrbase* parameter or its multiplications with power of 2. An example of defined schemata is given in Table 4.1.

The value of each schema c_s is equal to the number of ones included in the schema s.

The $\sigma_s(x)$ is another function, which practically adds the c_s value to the objective value, if the binary string corresponds to schema s. It is defined as

$$\sigma_s(x) = \begin{cases} 1 \text{ if } x \text{ is an instance of } s \\ \\ 0 \text{ otherwise} \end{cases}$$

Let me illustrate how it is possible to construct the Royal Road function. I defined a set of schemata, which are listed in Table 4.1.

The defined set of schemata has a basic building block of size 4. This can

Table 4.1: Example Royal Road function schemata

#	s	schema	value
1	$s_1 =$	1 1 1 1 * * * * * * * * * * * *	$c_{s_1} = 4$
2	$s_2 =$	* * * * 1 1 1 1 * * * * * * * *	$c_{s_2} = 4$
3	$s_3 =$	* * * * * * * * 1 1 1 1 * * * *	$c_{s_3} = 4$
4	$s_4 =$	* * * * * * * * * * * * 1 1 1 1	$c_{s_4} = 4$
5	$s_5 =$	1 1 1 1 1 1 1 1 * * * * * * * *	$c_{s_5} = 8$
6	$s_6 =$	* * * * * * * * 1 1 1 1 1 1 1 1	$c_{s_6} = 8$
7	$s_7 =$	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	$c_{s_7} = 16$

be clearly seen in the schemata s_1,s_2,s_3 and s_4. During the test run of the GA the size of this block can be defined by the parameter named *rrbase*. The second parameter, which specifies the Royal Road function is *rrmultiplier*. This defines how many sequential repetitions of the basic building block appear in the final combined schema. In the given example, the *rrmultiplier* is 4. There are also intermediate steps on the levels between the basic level and the final level. These always combine two blocks from the lower level into a new schema. Schemata s_5 and s_6 are the intermediate levels in our example. To ensure a proper number of intermediate levels, the *rrmultiplier* parameter must be a number which is a power of 2. In [40] there was a discussion about the contribution of the intermediate levels. Surprisingly, it was proven that the intermediate levels do not speed up the process of finding the optimum. They act oppositley. Since my goal was to compare the GAs with and without the use of the ealib archive, I ignored these discussions and always used the same parameters for the GAs. I did not use other parameters or bits (like introns) either.

For the testing I used Royal Road function with the *rrbase* values 2, 3, 4, 5 and 6 and *rrmultiplier* values 4, 8, 16 and 32. I combined every *rrbase* value with every *rrmultiplier* value and optimized them using the GA with and without the archive. Therefore, the simplest test case was *rrbase* value 2 and *rrmultiplier* value 4 (chromosome length 8). On the other hand, the most complicated testcase was *rrbase* value 6 and *rrmultiplier* value 32 (chromosome length 192).

4.3 NK fitness landscapes

The NK fitness landsape is another model designed for evaluating GAs. It was first introduced by Kauffman in [31]. A special feature of the NK landscapes is that the ruggedness of the landscape is controlled by a single parameter.

NK landscapes work with binary strings. A stochastic fitness function is defined, which assigns a fitness value between zero and one to each binary string ($F(x) : (x_1, x_2, \ldots, x_N) \mapsto [0,1], x_i \in \{0,1\}$). This function has the following mathematical definition:

$$F(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x_i, x_{i_1}, x_{i_2}, \ldots, x_{i_K})$$

Each gene (x_i) combinates a different fitness value (f_i) between zero and one, which is determined by the gene on the i^{th} position itself, and by its K closest neighbor genes $x_{i_1}, x_{i_2}, \ldots, x_{i_K}$ with $i_1, \ldots, i_K \in \{1, N\}$. The total fitness value is then computed as the average of all N fitness values. Theses fitness values are randomly chosen and stored in the stochastic tables. Generally, there are N tables with a size of 2^{K+1}. For each gene there is a fitness table, which stores 2^{K+1} uniformly distributed values. From this table the fitness value f_i is then read out. Figure 4.1 shows a concrete example of the NK landscape structure [3].

Parameter N is set to 7, so the binary string has length 7, paramater K equals 2. That means that the fitness value of each gene depends on two additional neighbor genes. There are also seven fitness value tables, one for each gene. Each table has 8 (= 2^{K+1}) values stored. The corresponding fitness value is picked up for each gene. The average of these values is the final fitness value of the whole chromosome.

There have been several discussions about the computational complexity of this problem [3, 59, 56]. We can say that the complexity depends on parameter K and the choosing method of the K neighbors. It is obvious that with increasing K, the problem becomes more complex. First, the computation of the objective value takes more time, and second, when the value of K is high, coupling between particular genes also rises. This makes the finding of the optimum more complicated.

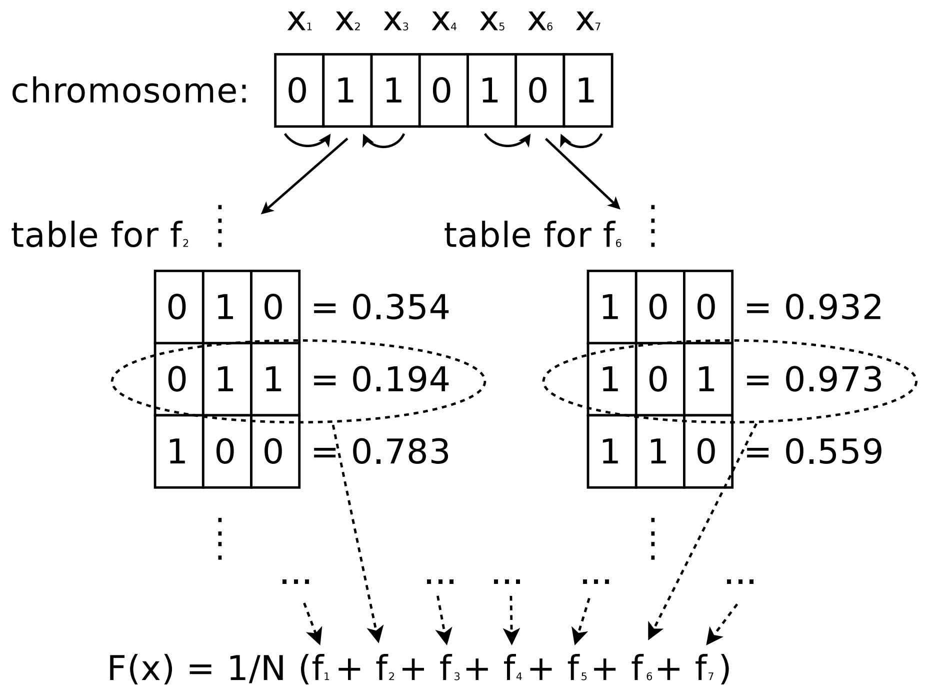


Figure 4.1: NK landscapes (N = 7, K = 2) - Fitness computation

Practice shows that the choosing method for the neighbor genes determines the computational complexity. There are two such methods:

- *adjacent neighbors* – the fitness function for one gene (f_i) is influenced by the nearest K genes
- *random neighbors* – K genes, which influence the fitness value f_i, are chosen randomly

Concerning these properties, the following theorems about computational complexity were presented [3, 59, 56]:

- NK landscape with *adjacent neighbors* – can be solved in polynomial time – $O(2^K N)$ steps
- NK landscape with *random neighbors* and $K \geq 2$ – this is an NP-complete problem
- NK landscape with *random neighbors* and $K = 1$ – can also be solved in polynomial time

The influence of the concrete N and K parameter values on the global optimum height was discussed in [53]. It was shown that

"the expected global optimum value increases with K, despite the fact that the average fitness of the local optima decreases."

It was proven that the highest values are in the NK landscapes, where $K = N - 1$. The increasing ruggedness also makes the finding of the global optimum more difficult.

I compared the performance of the GA with and without use of the arichve on different NK landscapes. For the N parameter I used values 20, 50, 100 and 300, and combined them with K parameter values 1, 2, 5, 6, 7, 8, 9 and 10. The concrete parameter values were not so important because I compared the performances of the GAs on different NK landscapes.

4.4 MAX-SAT problem

Unlike the Royal Road function and the NK landscapes, the Maximum satisfiability (MAX-SAT) problem is more general. The previously described functions were constructed especially to evaluate GAs. The MAX-SAT problem is an optimisation variant of the satisfiability (SAT) problem, which is a one of the main challenges in computer science.

The satisfiability (SAT) problem has alway played an important role in the computer sciences. It was proven that SAT is an NP-complete problem. To describe the SAT problem a conjunctive normal form (CNF) is used. Formally, there is a collection of m clauses, which contain n Boolean variables $(x_1, x_2, \dots, x_n)$. Each clause C_i contains 1 to n literals that are disjuncted. The literal can be any variable or its negation. The clause is satisfied if the disjunction of its literals elevates true. If every clause elevates true, the whole CNF formula is satisfied. So the SAT problem is to find at least one assignment of boolean values to the variables which satisfies all clauses [11].

There are many variations of SAT. MAX-SATis one of them. Its goal is to maximize the number of satisfied clauses or minimize the number of unsatisfied ones. The MAX-SATis also an NP-complete problem [28, 19].

Andjusting the MAX-SAT problem for the GA is not very complicated. Each variable is represented by one gene in the binary string chromosome. The results achieved by pure GAs are not very good. However, when the GA was enhanced by a special meta-heuristic, like local search, the achieved results were better [9, 45, 46, 55].

This was the main impulse for comparing the performances of the GAs with and without the ealib archive for this problem. I implemented the same encoding as it was described. Each variable was stored in one gene of the chromosome. For the tests I used two test sets published on the DIMACS (Discrete Mathematics and Theoretical Computer Science) website:

Asahiro, Iwama and Miyano (AIM) instances[1] were generated with a particular Random-3-SAT instance generator [27]. These instances can be described by three parameters:

- yes- / no- instance (solvable or not)
- number of variables (50, 100 and 200)
- the clause / variable ratio, (1.6, 2.0 for no-instances) and (1.6, 2.0, 3.4 and 6.0 for single-solution yes-instances)

In the whole set there are 4 instances for each parameter combination. So, together there are 72 AIM instances, bud I used only 8 of them. These instances are not considered as intrinsically hard. The complete algorithms show very good performance on these instances, however some local seach algorithms perform poor on the instances with low clause / variable ratio. But, enhancing the local search algorithms with clause weighting appears to make the instances solvable easily [10]. The reason for this could possibly be the occurance of very large plateaus in the search space of the problems with low clause / variable ratios.

For testing, I have picked up 8 of these instances. Four different solvable instances with 100 variables and 1.6 ratio, and four different unsolvable instances with the same parameters.

[1]http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/DIMACS/AIM/descr.html

Inductive inference (II) instances[2] are discribed in [30]. I picked 9 easiest instances of 41 benchmark instances of this problem type. The selected instances contain 66 to 510 variables and 186 to 3065 clauses. These instances appear to be rather easy for local search algorithms. However, complete algorithms had more difficulties to solve these instances, which is quite rare.

[2]http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/DIMACS/II/descr.html

Chapter 5

Implementation

5.1 General description

The concrete implementation of the trie is embedded in the ealib library - version 2.0. This library is intended to be a problem-independent C++ library suitable for the development of efficient metaheuristics for combinatorial optimization problems. Currently, it includes in particular classes for evolutionary algorithms, but it is intended to extend it also to other metaheuristics. The library is in development since 1999 at the Vienna University of Technology, Institute of Computer Graphics and Algorithms, Vienna, Austria.

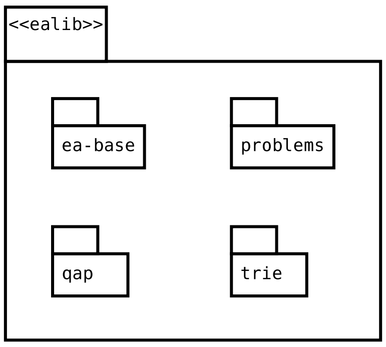


Figure 5.1: general ealib package architecture

The general structure of the ealib is described in Figure 5.1. The package architecture has following design:

- ea-base – a package, which contains all base classes of ealib. All metaheuristic algorithms and basic data structures are implemented in the classes of this package.
- problems – a package, which contains all classes of testproblems, which are used for testing of the algorithms.
- qap – a special package for solving the qap (Quadratic assignment problem).
- trie – package, which contains the classes of the trie structure.

I will ignore the QAP-package, because it does not relate to my work.

5.2 ea-base package - structure

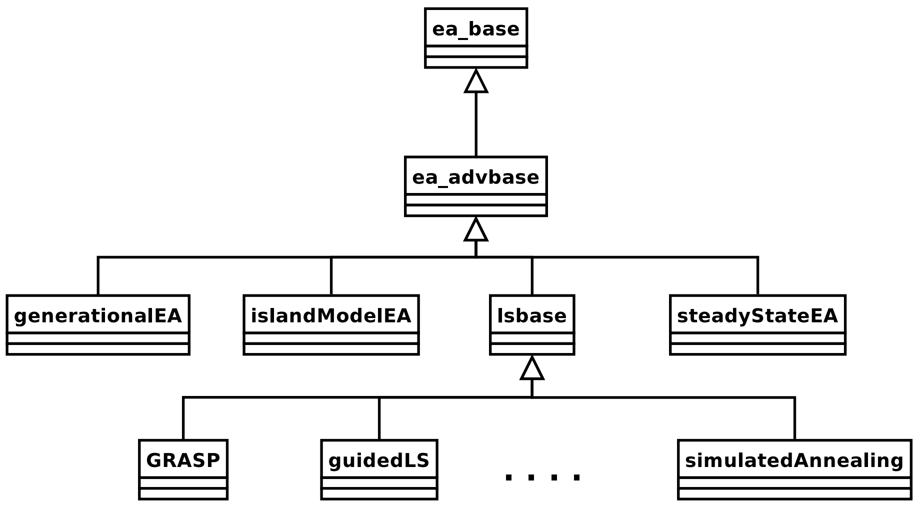


Figure 5.2: general ea-base package architecture

In this section I would like to describe parts of the ea-base package, which are relevant for the ealib trie. The ea-base contains genereally the main structures and algorithms for running programs based on Evolutionary algorithms (EA). This basic module can be parametrized with a set of parameters and extended by new packages, which can have their own packages as well. The relevant classes, which are displayed in Figure 5.2 are:

- *ea_base* – This is the most abstract base class for all EAs. This abstract base contains only the constructor, pure virtual *run()* method parameter attributes that might be needed in any EA and logging object. If a new EA is derived it uses the ea_base as the base class.

- *ea_advbase* – The more advanced abstract base class for EAs. This abstract base contains methods and attributes that are needed also in order to use an EA as sub-EA in an island model, for example. The new EA can be derived using the ea_advbase as the base class also.
- *generationalEA* – This is the base for running the generational EA. During each generation, all solutions are replaced by new ones generated by means of variation operators (crossover and mutation).
- *islandModelEA* – This GA uses the island model and contains several sub-GAs. During each generation, the performGeneration-function is called for each island, afterwards migration between the islands is performed.
- *lsbase* – This is an abstract base class for local search alike algorithms such as, GRASP, guided local search, simulated annealing etc.
- *steadyStateEA* – In the Steady-State EA during each generation, only one new solution is generated by means of variation operators (crossover and mutation). The new solution replaces an existing solution (e.g. the worst of the population). Usually, generated duplicates are discarded using duplicate removal principle (duplicate elimination).

5.3 ea_advbase and steadyStateEA

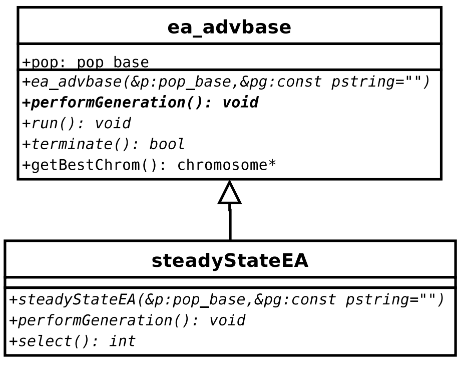

ea_advbase
+pop: pop_base
+ea_advbase(&p:pop_base,&pg:const pstring="")
+performGeneration(): void
+run(): void
+terminate(): bool
+getBestChrom(): chromosome*
steadyStateEA
+steadyStateEA(&p:pop_base,&pg:const pstring="")
+performGeneration(): void
+select(): int

Figure 5.3: attributes and methods of the ea_advbase class

I will describe now the *ea_advbase* and *steadyStateEA* classes in more detailed way. The reason, why I am doing it is, that the trie extension was compared to classical Steady-state EA, but also comparison to other approaches will be done in the future.

The main purpose of the *ea_advbase* class is the running of the EA. It has constructors, which create the generation and insures the right application of the genetic operators such as selection, mutation and crossover. It has also a pure virtual function *performGeneration()*, which is implemented specifically in each non-abstract descendand class.

The *steadyStateEA* class is inherited from the *ea_advbase* class. It implements own *performGeneration()* method. Its implementation corresponds to the principles of steady state EA. It selects two parent structures, recombines them, evaluates them, and adds them to the population, replacing some older. There are many different known strategies for selection, recombination and replacement. Each has its own pros and cons.

In Figure 5.3 we can see some selected attributes and methods of the *ea_advbase* class.

5.4 problems package

Next important structures are the classes that describe the problems, which ealib can solve. Which are more general and describe the structures for storing of the solutions (chromosome, gene, etc.), are still part of the *ea_base* package. On the other hand, the more concrete classes are in the *problems* package already.

The most general structure used in the GA for problem and solution encoding is a chromosome. Every single solution is stored as one chromosome in the population. Each individual contains one chromosome. The basic unit of the chromosome is a gene. Each gene encodes one *property* of the individual (solution). The chromosome, which corresponds with one specific solution, contains a number of genes. In our implementation there is a abstract class called chromosome. Afterwards comes a template class *stringChrom<T>*, which defines the behavior of the chromosome closer. This class is inherited by several subclasses, which are

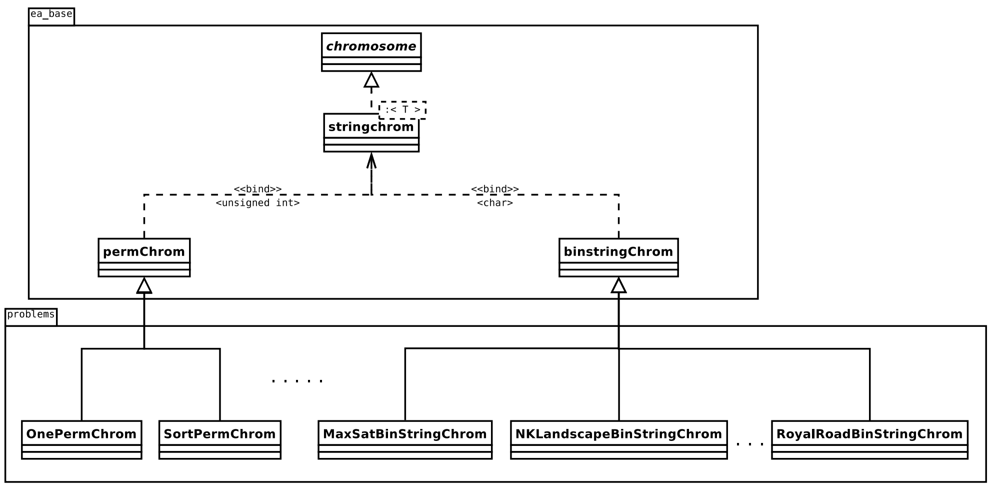


Figure 5.4: chromosome class and its descendants

differenced by the type of the gene. For example the gene type in the class *binstringChrom* is *char*, whereas the class *permChrom* contains *unsigned int* genes.

The concrete problem classes in the package *problems* only define the evaluation (objective) function and some other utility functions for the problems. These are for example *SortPermChrom*, *MaxSatBinStringChrom*, *RoyalRoadBinStringChrom* etc. These classes also know, how to decode the chromosome and what the genes mean.

The basic architecture of chromosome class and its descendants is displayed in Figure 5.4.

Detailed view of chromosome architecture is shown in Figure 5.5. It displays the important methods, dependencies and inheritances.

The abstract class *chromosome* defines the basic chromosome behavior and its related utilities. Typical utility functions are the functions *isBetter()* and *isWorse()*. Very important are procedures *mutate()*, *crossover()* and the function *objetive()*. In the *chromosome* class they are pure virtual.

The template class *stringChrom<T>* represents next level of the hierarchy. Methods *mutate()* and *crossover()* are implemented here. Depending on the set parameters they can call different mutation or crossover modifications, like *mutate_flip()*, *mutate_exchange*, *crossover_1point()* or *crossover_multipoint()*. Also the

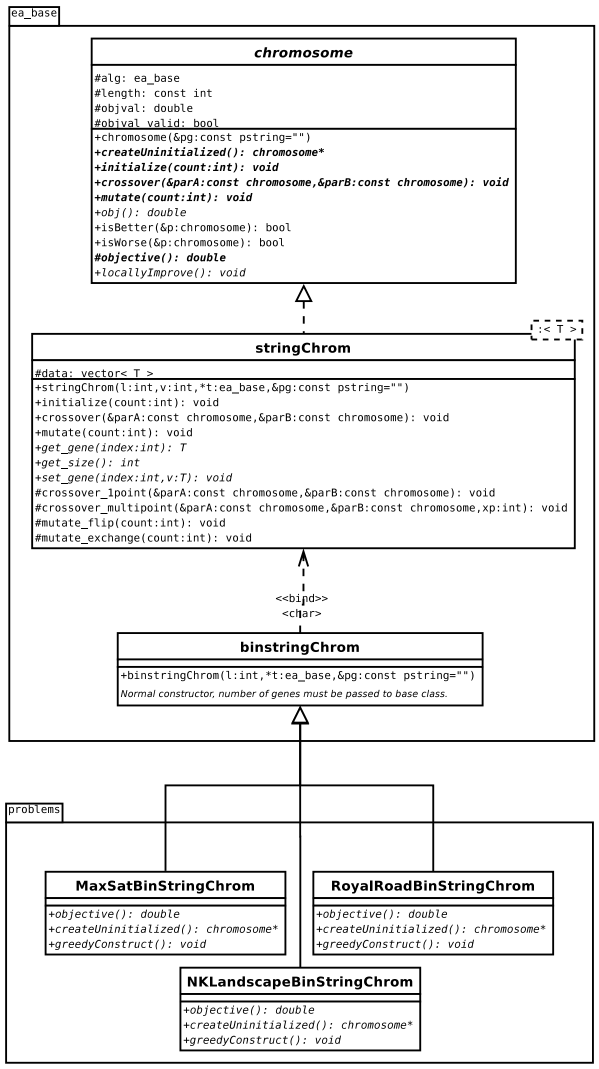

ea_base
chromosome
#alg: ea_base
#length: const int
#objval: double
#objval_valid: bool
+chromosome(&pg:const pstring="")
+createUninitialized(): chromosome*
+initialize(count:int): void
+crossover(&parA:const chromosome,&parB:const chromosome): void
+mutate(count:int): void
+obj(): double
+isBetter(&p:chromosome): bool
+isWorse(&p:chromosome): bool
#objective(): double
+locallyImprove(): void
:< T >
stringChrom
#data: vector< T >
+stringChrom(l:int,v:int,*t:ea_base,&pg:const pstring="")
+initialize(count:int): void
+crossover(&parA:const chromosome,&parB:const chromosome): void
+mutate(count:int): void
+get_gene(index:int): T
+get_size(): int
+set_gene(index:int,v:T): void
#crossover_1point(&parA:const chromosome,&parB:const chromosome): void
#crossover_multipoint(&parA:const chromosome,&parB:const chromosome,xp:int): void
#mutate_flip(count:int): void
#mutate_exchange(count:int): void
<<bind>>
<char>
binstringChrom
+binstringChrom(l:int,*t:ea_base,&pg:const pstring="")
Normal constructor, number of genes must be passed to base class.
problems
MaxSatBinStringChrom
+objective(): double
+createUninitialized(): chromosome*
+greedyConstruct(): void
RoyalRoadBinStringChrom
+objective(): double
+createUninitialized(): chromosome*
+greedyConstruct(): void
NKLandscapeBinStringChrom
+objective(): double
+createUninitialized(): chromosome*
+greedyConstruct(): void

Figure 5.5: detailed view of chromosome architecture

basic gene handling is defined in this class. As it use a generic type, it can define it for all subclasses. The genes are stored in the *data* vector and handled by procedures *set_gene()*, *get_gene()* or *get_size()*.

In the example I use, there is a class *binStringChrom*, which is used to instantiate the template class *stringChrom<T>* with a concrete type (char). For each concrete problem there is always even more specific derived class. This class is mostly, but not completely independent of the used EA.

The concrete implementation classes for each specific problem can be found in the *problems* package. Every class in the *problems* package implements the *ob-*

jective() function, which defines, how to evaluate the objective value of the chromosome. For my work I have defined three specific problem classes (*RoyalRoadBinStringChrom*, *MaxSatBinStringChrom* and *NKLandscapesBinStringChrom*). Additionally there can be some utility functions, which encode or decode the chromosome genes like *getNumOfClauses()* or *getNumOfVariables()* in the *MaxSatBinStringChrom* class for instance.

5.5 trie package

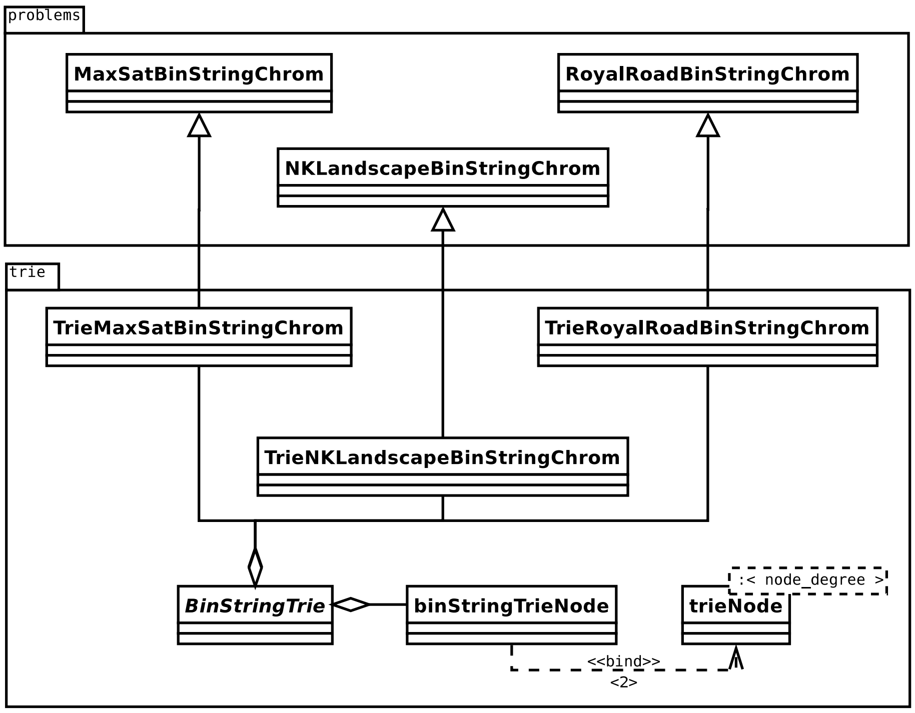
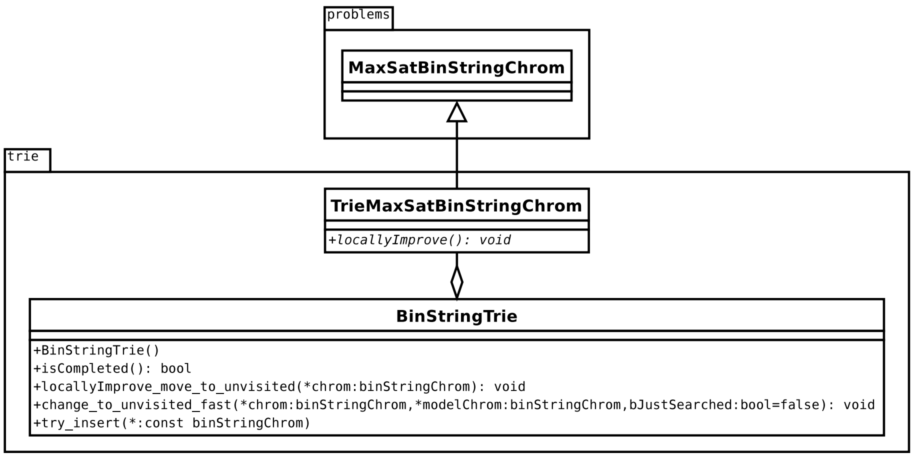


Figure 5.6: structure and dependencies in the trie package

The general structure of the trie package is displayed in Figure 5.6. The package is divided into two parts. First part comprises classes, which define the trie structure. These are *TrieNode* and *BinStringTrie*. Other classes (second part of the trie package) are extensions of the problem-classes from the problems package. The extension is mandatory, because the chromosome classes have to communicate with the trie somehow. The responsibility for the communication lies in the method *locallyImprove()*, which is implemented in the trie-problem classes.

However, it defined in the *chromosome* class already and used by other local improvement strategies. Detailed view of the important methods of the trie structure is shown in Figure 5.7.



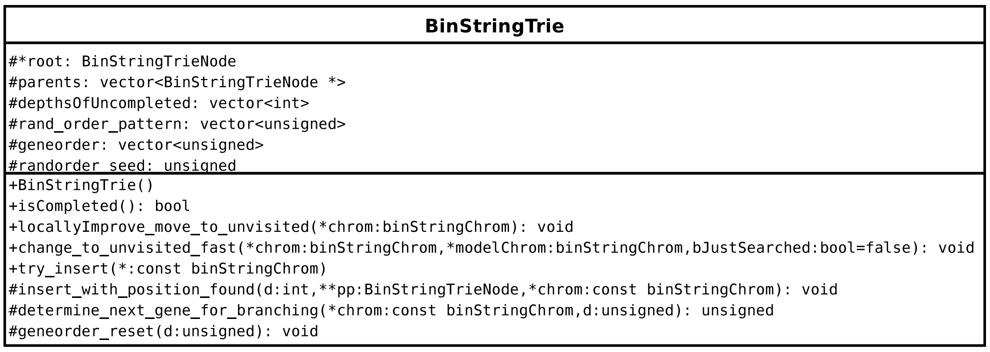
```
+locallyImprove(): void
```

```
+BinStringTrie()
+isCompleted(): bool
+locallyImprove_move_to_unvisited(*chrom:binStringChrom): void
+change_to_unvisited_fast(*chrom:binStringChrom,*modelChrom:binStringChrom,bJustSearched:bool=false): void
+try_insert(*:const binStringChrom)
```

Figure 5.7: important methods and functions in the trie package

5.6 BinStringTrie and TrieNode

As Figure 5.6 displays, there are two basic classes used by the trie. *BinStringTrie* class, shown in Figure 5.8, and trienode class, shown in Figure 5.9.



```
#*root: BinStringTrieNode
#parents: vector<BinStringTrieNode *>
#depthsOfUncompleted: vector<int>
#rand_order_pattern: vector<unsigned>
#geneorder: vector<unsigned>
#randorder_seed: unsigned
```

```
+BinStringTrie()
+isCompleted(): bool
+locallyImprove_move_to_unvisited(*chrom:binStringChrom): void
+change_to_unvisited_fast(*chrom:binStringChrom,*modelChrom:binStringChrom,bJustSearched:bool=false): void
+try_insert(*:const binStringChrom)
#insert_with_position_found(d:int,**pp:BinStringTrieNode,*chrom:const binStringChrom): void
#determine_next_gene_for_branching(*chrom:const binStringChrom,d:unsigned): unsigned
#geneorder_reset(d:unsigned): void
```

Figure 5.8: BinStringTrie class - attributes and methods

The *binStringTrie* class contains following basic attributes:

- *BinStringTrieNode *root* – Root pointer of trie.

- *vector<BinStringTrieNode *> parents* – Stores pointers to all nodes on the path of the current (last) search.
- *vector<int> depthsOfUncompleted* – Stores depths (index) of nodes in parents vector, for which the alternative next pointer (i.e. not the one traversed in the last search) is not yet completed.

The *root points on the first TrieNode of the trie. Vectors parents and depthsOfUncompleted are two helper vectors used to speed up basic trie operations (insertion of a new solution and suggesting of an unvisited solution).

The basic *binstringtrie* class methods are:

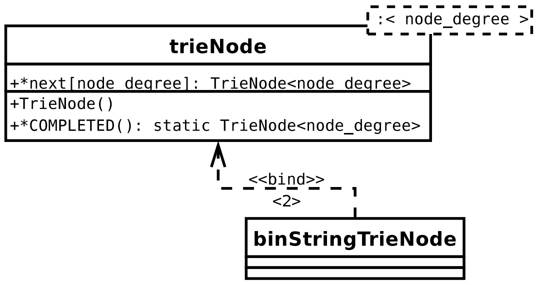

:< node_degree >
trieNode
+*next[node_degree]: TrieNode<node_degree>
+TrieNode()
+*COMPLETED(): static TrieNode<node_degree>
<<bind>>
<2>
binStringTrieNode

Figure 5.9: trienode class - attributes and methods

- *BinStringTrie()* – Constructor. Initializes empty trie.
- *bool isCompleted() const* – Returns true if the whole trie has been completed, i.e. the whole search space has been evaluated.
- *void locallyImprove_move_to_unvisited(binStringChrom *chrom)* – Local improvement function to be called within locallyImprove() of the specific chromosome. It includes the mechanism of using the trie, eventually modifying the solution.
- *bool try_insert(const binStringChrom *)* – Try to insert a new chromosome. Returns false if the chromosome is already in the trie, i.e. the solution has already been visited. It keeps the information about the search path to the solution in vectors parents and deptshOfUncompleted.
- *void insert_with_position_found(int d,BinStringTrieNode **pp, const binStringChrom *chrom)* – Helper function for inserting a new solution in the trie when its final pointer has already been identified. It marks the final next-pointer corresponding to the new solution as completed and then goes up the trie

propagating the completion of a subtrie and removing any nodes in which both subtrees are completed.

- *void change_to_unvisited_fast(binStringChrom *chrom, const binStringChrom *modelChrom, bool bJustSearched=false)* – Changes given solution into a similar to a modelChrom but yet unvisited solution. If bJustSearched is true, the given chromosome has just been searched in the trie, i.e., the search path is stored in vector parents. This version of this operator is the fast and simply way, in which we go back to the last uncompleted parent node and then down at the alternative (still uncompleted) side, following further the given chromosome as far as possible but avoiding running into a completed subtrie.

Chapter 6

Experiments

In Chapter 3 I introduced the ealib trie structure, which can be added to any GA helping them to find better solutions by storing the visited ones and avoiding revisits. To study the performance of the GA with and without the use of the trie I performed several tests. First, I chose the test functions, which are described in Chapter 4, and then I made test runs using different sets of parameters. Afterwards, I statistically evaluated the acquired results.

In this Chapter, I describe the GA parameters first. These are the parameters which I use in all test runs independent from the test problem. Afterwards, I describe the performed experiments. There are also some special parameters for each test problem. These are described in the corresponding test problem's section.

6.1 Parameters for test problems

The parameter types described in this section can be divided into two groups. The first group is the set of parameters which can be used by each GA, and the second group is the parameter set related to the ealib trie use.

Here is a brief description of relevant GA parameters:

- popsize – The population size. Depending on the problem solved, I used the values 20, 50, 100, 200, 500 and 1000.
- tselk – Group size for tournament selection. I used the values 2, 4 and 10.

- tgen – The number of generations until termination. These parameter values were set to 1000, 10000 or 100000, depending from the problem solved.
- pcross – The crossover rate. This parameter was always set to the value of 1.
- pmut – The mutation rate for new chromosomes. This parameter was always set to the value of 1 per chromosome.
- dupelim – Default duplicate elimination. This parameter has effect only for the GA without the use of the trie and was always set to 1.

The following is a description of the parameters that influence the ealib trie functioning:

- trie_use – Tells the GA to use the ealib trie to store the visited solutions. Except for the tests using the standard GA, it was always set to 1.
- randomDepthOfUncompleted – If set to 0, the deepest alternative is always used for moving to an unvisited solution. If set to 1, a random choosing algorithm is used, described in Section 3.4.4.
- randomizedGeneOrder – When set to 1, is organizes the structure of the trie randomly according to Section 3.5.2. Otherwise, the trie structure is built up as described in Section 3.5.1.

For better human readability and orientation in the parameters of each test run, I have defined some abbreviations. Each abbreviation defines how the parameters *trie_use*, *randomizedGeneOrder* and *randomDepthOfUncompleted* are set. Table 6.1 displays the parameter settings.

Table 6.1: Possibilities of the ealib trie parameter settings

abbreviation	trie_use	randomizedGeneOrder	randomDepthOfUncompleted
std	do not use (0)	—	—
tnd	use (1)	normal (0)	default (0)
tnr	use (1)	normal (0)	random (1)
trd	use (1)	random (1)	default (0)
trr	use (1)	random (1)	random (1)

6.2 Experiments with Royal Road function

First, I used the Royal Road function for the tests. This is a function that was specially designed for evaluating GAs. Closer description of this function is written in Section 4.2.

6.2.1 Paramaters for Royal Road function

For Royal Road function there are two special parameters defined:

- *rrbase* – defines the size of the basic Royal Road function building block
- *rrmultiplier* – defines how many multiplications of the basic building block there are in the final schema

6.2.2 Test results - Royal Road function

Test 1 - comparison of the quality of the solutions

My first goal was to examine the fitness value achieved by the algorithms within 1000 generations. Table 6.2 lists which values I used for each parameter. 100 test runs were then performed for each possible parameter combination.

Table 6.2: Royal Road function test - parameter settings

paramater name	possible values
algorithm type	std, tnd, tnr, trd, trr
tgen	1000
tselk	10
popsize	50, 100, 200, 500
rrbase	2, 3, 4, 5, 6
rrmultiplier	4, 8, 16

The achieved results (Table 6.3) are separated according to algorithm type (std, tnd, tnr, trd, trr) and maximal achievable fitness value ($= rrbase * rrmultiplier * \log_2(rrmultiplier) + 1$). For every algorithm type there are two values; mean value, and stadard deviance value.

Table 6.3: Test results: Royal Road function, 1000 generations (std, tnd, tnr, trd, trr)

			std		tnd		tnr		trd		trr	
rrb	rrm	max	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
2	4	24	24.00	0.00	24.00	0.00	24.00	0.00	24.00	0.00	24.00	0.00
3	4	36	36.00	0.00	36.00	0.00	36.00	0.00	36.00	0.00	36.00	0.00
4	4	48	48.00	0.00	48.00	0.00	48.00	0.00	48.00	0.00	48.00	0.00
5	4	60	59.48	4.25	59.83	2.47	59.65	3.48	59.65	3.48	**60.00**	0.00
2	8	64	64.00	0.00	64.00	0.00	64.00	0.00	64.00	0.00	64.00	0.00
6	4	72	63.00	17.54	64.50	16.34	**66.21**	14.71	62.76	17.72	64.95	16.29
3	8	96	95.78	3.17	95.78	3.17	95.78	3.17	**96.00**	0.00	**96.00**	0.00
4	8	128	116.88	23.34	119.88	20.56	119.12	21.62	115.88	24.26	**121.38**	18.83
2	16	160	156.27	14.76	157.19	12.95	**159.07**	7.54	158.14	10.58	158.45	9.68
5	8	160	97.63	40.17	105.13	42.19	102.58	41.32	98.78	40.91	**105.85**	43.71
6	8	192	69.87	29.83	74.43	34.56	74.10	35.87	62.57	40.49	**75.30**	34.59
3	16	240	187.07	51.30	188.40	52.07	**200.15**	50.47	189.09	51.24	196.80	51.71
4	16	320	140.48	45.52	140.18	51.54	147.44	59.65	**149.56**	55.90	143.56	58.38
5	16	400	100.95	32.83	95.00	33.47	**101.13**	35.76	99.08	33.92	98.85	35.52
6	16	480	74.43	28.93	73.41	31.26	71.43	26.38	73.13	28.06	**76.32**	34.09

The same results are displayed in Figure 6.1. The Y-axis represents the percentage values. The results are ordered according to the maximum possible fitness value. Results for the parameter setting *rrbase* = 2 and *rrmultiplier* = 4 (maximum fitness value 24) are represented by the values on the leftmost side of the graph, and results for the parameter setting *rrbase* = 6 and *rrmultiplier* = 16 (maximal fitness value 480) are represented by the values on the rightmost side of the graph.

For example, for parameter values *rrbase* = 2 and *rrmultiplier* = 4, all algorithms reached the maximal fitness value (24) within 1000 generations in each run. So, the percentage for this setting is 1.0 for each algorithm type.

For parameter values *rrbase* = 3 and *rrmultiplier* = 16 (maximal fitness value = 240) the average result of the std algorithm was 187.1 (0.78%) and the average result of the trr algorithm was 196.8 (0.82%). Therefore the trr-line is higher then the std-line in this case.

Furthermore, I compared the difference between the average achieved results for each combination of the parameters *rrbase* and *rrmultiplier* and algorithm type to the the average value of all algorithm types (Figure 6.2). I expressed these differences in percents again.

From these results we can see that except in the most trivial cases the std algorithm never achieved the best result among all the other algorithms. Another important thing to notice is that the standard GA parameters influenced the

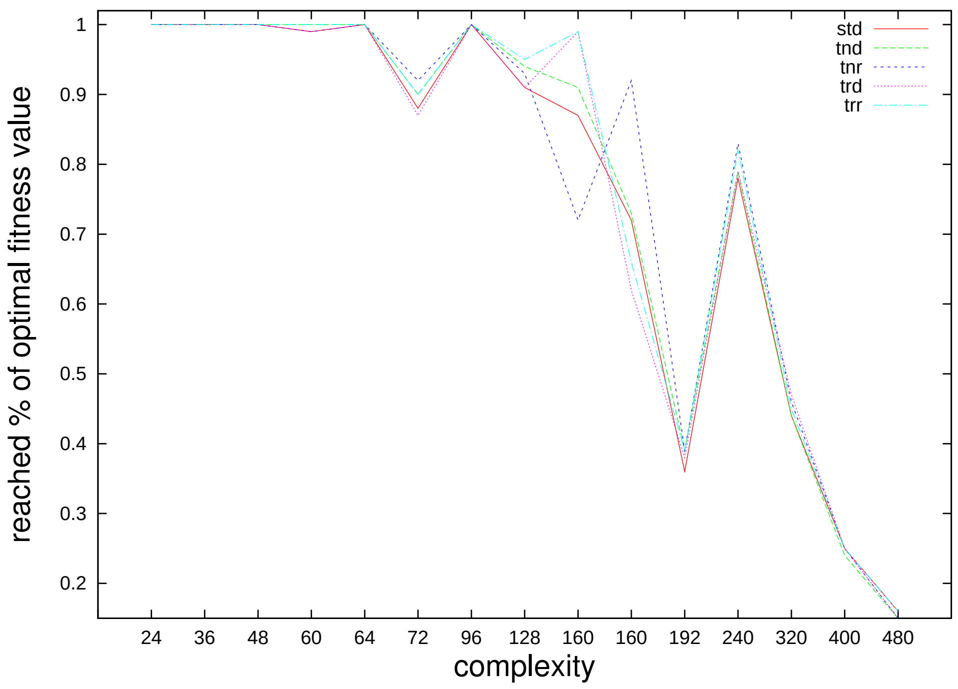


Figure 6.1: RR, 1000 generations - comparison of fitness value means for each RR configuration

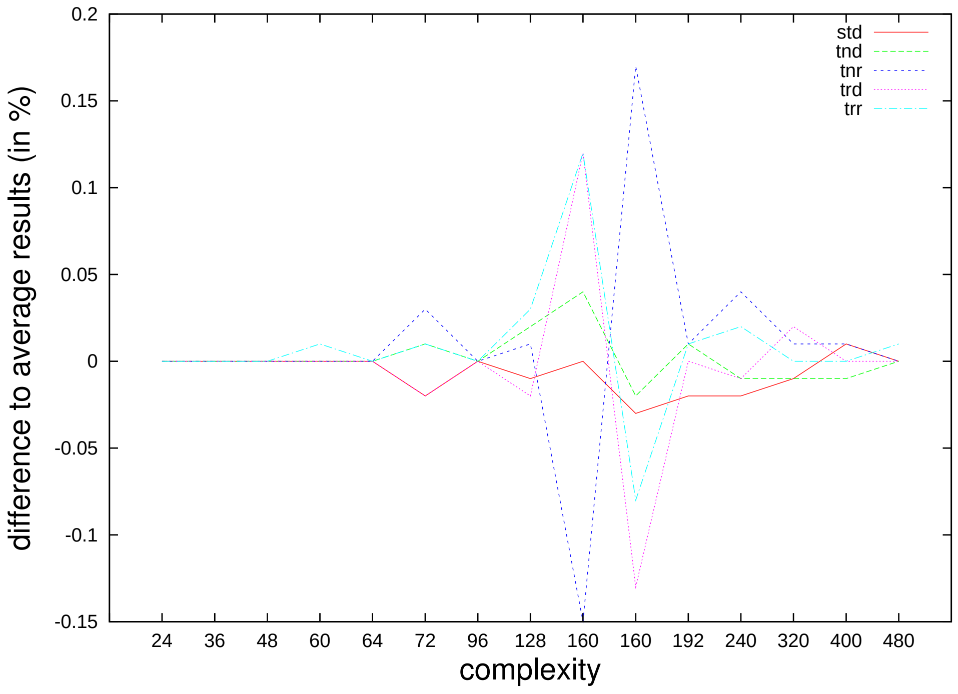


Figure 6.2: RR, 1000 generations - difference between mean and average mean value of all algorithms

achieved results much more than the use of the archive. But this was not a problem, because my goal was to compare the runs with or without the use of the trie. From the graphs it is not obvious which algorithm achieved the best results in the summary, or which algorithm has better performance than the others. To examine this, I have put all achieved results for all possible parameter combinations into one table. After this, I compared each type of algorithm with the others using the Wilcoxon test. Table 6.4 shows the results (p-values) of the comparison.

Table 6.4: Wilcoxon test - fitness value comparison: Royal Road function 1000 generations

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	88.921	51.337	0.02			0.8551	0.9997	0.7696	0.9996
tnd	89.714	52.288	0.03	272	0.1449		0.9909	0.3316	0.9943
tnr	91.243	54.763	0.03	271	0.0003	0.0091		0.0078	0.4971
trd	89.776	53.466	0.05	262	0.2304	0.6685	0.9922		0.9955
trr	91.297	54.326	0.05	268	0.0004	0.0057	0.5029	0.0045	

Looking at this comparison we can see that all algorithms with the use of the archive have achieved better results than the standard GA without the use of the archive. However, significantly better results were achieved only by the algorithms which were using the *suggestion changed in a random place algorithm* (tnr and trr). This result depends probably on the structure of the Royal Road function. It is easier to build a block of ones when this block builds one subtrie of the trie.

The main advantage of the ealib trie is that it visits more solutions within same number of generations. The GA can profit out of this property when the ratio between the number of visited solutions and solutions in the search space is relevant. The second advantage of the ealib trie archive is higher mutation rate in the most visited parts of the search space. The GA can take advantage out of this when the mutation has higher chances to influence the fitness value of the solution. This can be influence can observed in the test result with lower *rrbase* parameter.

Test 2 - comparison of the number of generations needed to reach the optimum

Table 6.5 displays the parameter settings for Test 2. In this test, I left out the most difficult *rrbase* and *rrmultiplier* combinations and increased the number of generations in each run, because for this comparison it was always necessary for all algorithms to achieve the maximum fitness value.

Table 6.5: Royal Road function Test 2 - parameter settings

paramater name	possible values
algorithm type	std, tnd, tnr, trd, trr
tgen	10000
tselk	10
popsize	50, 100, 200, 500
rrbase	2, 3, 4, 5
rrmultiplier	4, 8, 16

After running 100 testruns for each possible parameter combination, I compared for each in which generation the optimum was reached. Figure 6.3 displays the number of generations needed to reach the optimum for each algorithm type. The maximum possible fitness value rises from left to right.

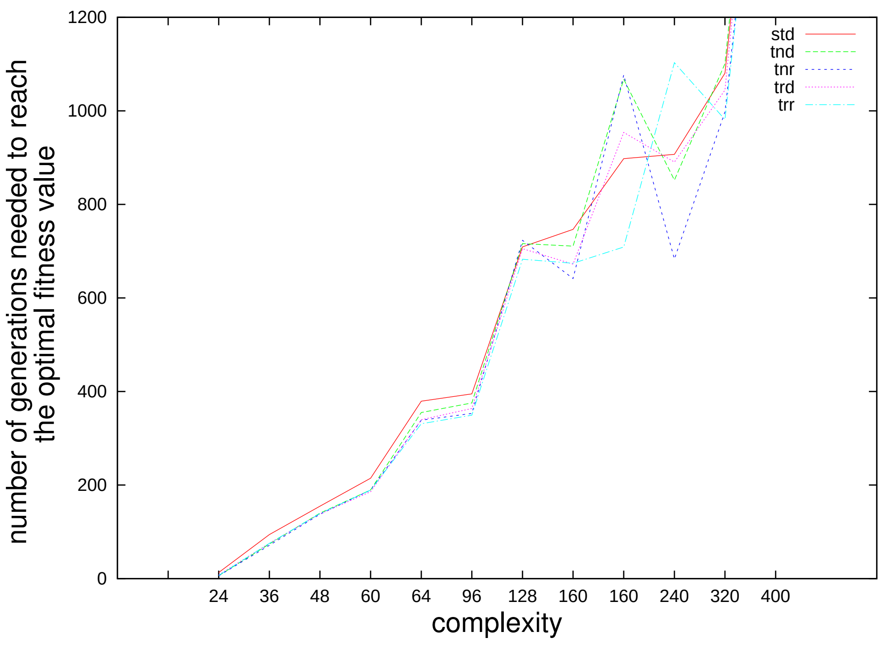


Figure 6.3: RR 10000 generations - means of the generations needed to reach the optimum

In Figure 6.4 there is another comparison of the results. The value 1 represents the maximum number of generations of all algorithm types. All other algorithm type values are computed proportional to this maximum value. The maximum possible fitness value rises again from left to right.

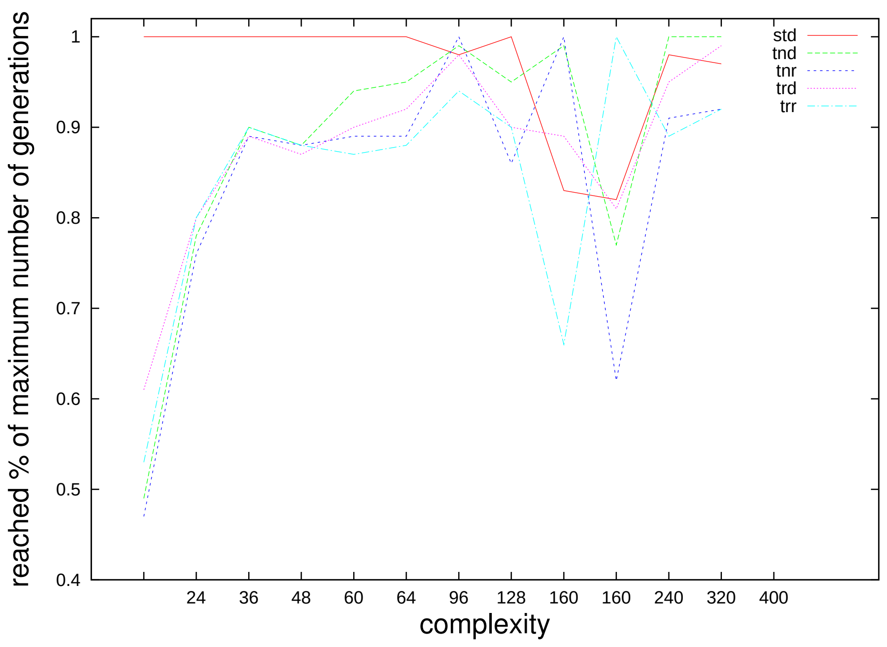


Figure 6.4: RR, 10000 generations - comparison of the number of generations needed to find the optimal fitness value

Then I summarized all the results and performed the Wilcoxon test again (Table 6.7). This test showed that all algorithms using the archive found the optimum significantly earlier than the standard algorithm without the use of the archive. From the table we can also read that the performance of the algorithm which used the archive without any improvement (tnd) was significantly worse than the performance of other algorithms with the archive. Other comparisons have not shown significant differences, but the results of the archives with the *suggestion changed in a random place algorithm* were better than the results of the other ones.

Table 6.6: Test results: Royal Road function, 10000 generations (std, tnd, tnr, trd, trr)

			std		tnd		tnr		trd		trr	
rrb	rrm	max	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
2	4	24	12.8	19.9	6.3	10.5	6.0	10.3	7.9	14.2	**6.8**	10.6
2	8	64	154.8	103.2	139.9	78.4	**137.5**	86.9	138.2	88.9	139.1	83.9
2	16	160	522.4	249.9	488.1	240.3	**461.2**	242.5	484.9	236.2	463.6	252.1
3	4	36	94.2	78.9	73.8	55.3	**71.1**	57.6	75.2	54.1	75.1	54.3
3	8	96	379.2	187.0	355.0	173.3	338.6	185.8	339.7	184.4	**331.0**	190.9
3	16	240	1080.7	483.9	1100.8	431.1	998.0	441.8	1044.5	435.6	**981.5**	458.3
4	4	48	214.6	118.5	189.4	114.6	**189.1**	110.5	185.8	111.8	189.2	119.9
4	8	128	746.6	347.7	710.7	316.5	**641.5**	325.1	672.2	311.0	674.2	332.6
4	16	320	2097.5	817.8	2154.9	756.0	1991.2	741.4	2126.1	796.1	**1978.1**	832.8
5	4	60	394.8	215.8	375.5	213.6	353.1	191.7	363.7	185.1	**349.3**	184.3
5	8	160	**1282.4**	558.8	1431.4	763.0	1298.6	695.5	1359.7	680.3	1348.3	690.0
5	16	400	**4496.3**	1701.8	4949.9	1885.0	4828.1	1950.6	4688.6	2066.9	4665.5	1959.5

These results also confirmed my conclusions from the Test 1. The main advantage of the ealib trie is in the count of visited solutions within the same number of generations and higher mutation rate in the most visited parts of the search space. We see that with raising size of the search space and higher *rrbase* parameter achieves the ealib trie even worse results. (see Table 6.6, *rrbase* = *5*).

Table 6.7: Wilcoxon test - number of generations comparison: Royal Road function 10000 generations

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	956.36	1356.43	0.05			0.3104	0.0000	0.0000	0.0000
tnd	997.97	1491.25	0.08	5180	0.6896		0.0000	0.0004	0.0000
tnr	942.83	1458.21	0.07	5328	1.0000	1.0000		0.9613	0.4904
trd	957.21	1450.72	0.11	5100	1.0000	0.9996	0.0387		0.0093
trr	933.48	1427.85	0.11	5298	1.0000	1.0000	0.5096	0.9907	

6.3 Experiments with NK landscapes problem

For the next test function I used the NK fitness landscape function. A closer description of this function is written in Section 4.3. This function was also created to examine and prove the features of GAs.

6.3.1 Paramaters for NK landscapes problem

For NK landscapes there are three special parameters defined:

- nk_n – defines the length of the landscape and of the chromosome
- nk_k – defines by how many other genes the gene value is influenced
- nk_seed – seed used from the random function generator, determines the structure of the NK landscape

6.3.2 Test results - NK landscapes problem

Test 1

My first goal was to examine the fitness value achieved by the algorithms within 10000 generations. Table 6.8 lists which values I used for each parameter. 50 test runs were then performed for each possible parameter combination, each run with a different *nk_seed* value. Every landscape is determined by the combination of three nk parameters (*nk_n*, *nk_k* and *nk_seed*). Thus each algorithm ran on each of the 1600 landscapes 4 times (once with every popsize).

Table 6.8: NK landscapes Test 1 - parameter settings

paramater name	possible values
algorithm type	std, tnd, tnr, trd, trr
tgen	100000
tselk	10
popsize	50, 100, 200, 500
nk_n	20, 50, 100, 300
nk_k	1, 2, 5, 6, 7, 8, 9, 10
nk_seed	1 - 50

Figure 6.5 displays the results of the first test run. Each point represents the average achieved fitness value over all *nk_seed* values for one popsize, *nk_n* and *nk_k*. The results are ordered from left to right by *nk_k*, then *nk_n*, and then popsize. In the displayed graph it can be seen that the std algorithm achieved the lowest fitness value in most cases. Detailed results are displayed in Table 6.9 and in Table 6.10. Both tables display the same results, but Table 6.9 summarizes all *nk_n* values for each *nk_k* value. The column % max shows the ratio between each mean and maximum mean among all.

Table 6.9: Test results: NK landscapes, 100000 generations, tselk 10

	std		tnd		tnr		trd		trr	
k	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
1	0.7624	0.0242	0.7635	0.0251	0.7650	0.0241	0.7646	0.0240	**0.7656**	0.0239
2	0.7603	0.0262	0.7620	0.0266	0.7637	0.0253	0.7630	0.0254	**0.7647**	0.0253
5	0.7413	0.0310	0.7420	0.0308	0.7444	0.0313	0.7435	0.0311	**0.7445**	0.0312
6	0.7557	0.0272	0.7567	0.0273	**0.7599**	0.0266	0.7578	0.0258	0.7596	0.0261
7	0.7518	0.0278	0.7550	0.0283	0.7567	0.0279	0.7564	0.0275	**0.7580**	0.0268
8	0.7504	0.0284	0.7531	0.0304	0.7538	0.0297	0.7530	0.0295	**0.7551**	0.0301
9	0.7221	0.0219	0.7224	0.0217	0.7236	0.0212	0.7231	0.0213	**0.7238**	0.0212
10	0.7494	0.0283	0.7502	0.0279	0.7511	0.0277	0.7508	0.0276	**0.7513**	0.0274

To display the differences between the algorithms, I created the next two figures. Figure 6.6 displays averaged results achieved by each algorithm. The values are ordered in the same way as in the previous figure, but each next value is added to all previous values, and the result is averaged. Thus, each added value contributes to the previous computed average. This average is computed by the following formula:

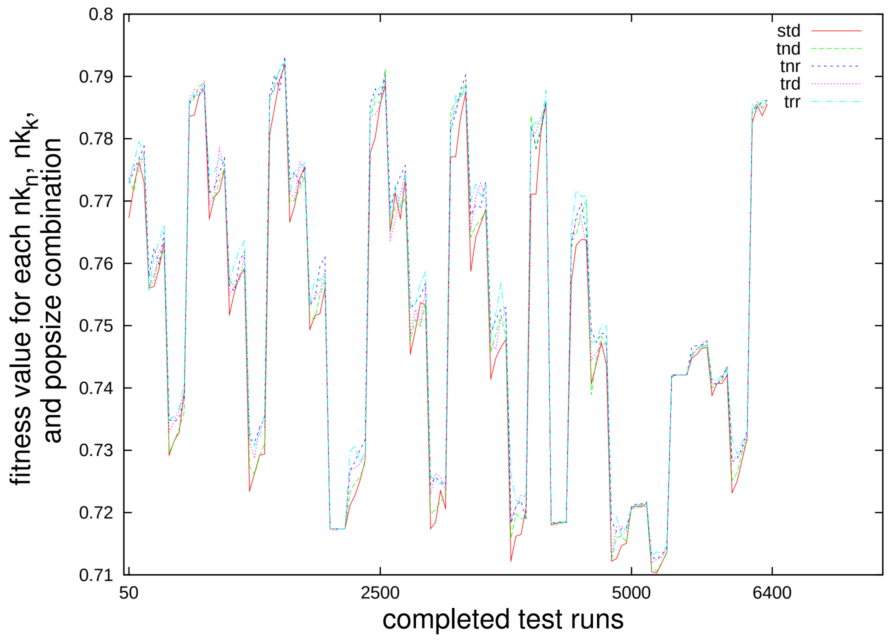


Figure 6.5: NK, 100000 generations, tselk 10 - fitness values for each combination of popsize, *nk_n* and *nk_k*

$$F(i) = \frac{1}{i}\sum_{j=1}^{i} f_{j(nk\_n_j,\ nk\_k_j,\ popsize,\ all\ nk\_seed\ values)}$$

The next figure (Figure 6.7) was also created to demonstrate the differences between the algorithm types. I ordered all achieved result from least to greatest and put them into the graph.

Figure 6.6 and Figure 6.7 show that there is a certain gap between different algorithm types. They show that the use of the ealib trie influences the achieved results in a positive way. Table 6.11 displays the results of the Wilcoxon test, which compared all achieved results. Using the results of this table we can order all algorithms by the quality of achieved results from best to worst (1. trr, 2. tnr, 3. trd, 4. tnd, 5. std). Again we can see that the implemented ealib trie improvements have brought us significant betterment of the results.

The main reason for it is higher relevance of mutation rate in this problem. Even in large search spaces a mutation of a single gene causes the change of the fitness value. That makes the finding of a better solution easier. This explains also the differences between the algorithms which use ealib archive. Random-

Table 6.10: Test results: NK landscapes, 100000 generations, tselk 10

		std		tnd		tnr		trd		trr	
k	n	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
20	1	0.7723	0.0151	0.7746	0.0155	0.7759	0.0162	0.7755	0.0152	**0.7764**	0.0152
50	1	0.7587	0.0135	0.7592	0.0135	0.7613	0.0132	0.7595	0.0127	**0.7616**	0.0141
100	1	0.7332	0.0090	0.7327	0.0093	0.7358	0.0082	0.7358	0.0085	**0.7369**	0.0084
300	1	0.7856	0.0187	0.7874	0.0186	0.7870	0.0188	**0.7876**	0.0182	0.7873	0.0190
20	2	0.7711	0.0141	0.7725	0.0144	0.7737	0.0149	0.7735	0.0146	**0.7754**	0.0140
50	2	0.7561	0.0123	0.7576	0.0124	0.7586	0.0119	0.7575	0.0126	**0.7607**	0.0129
100	2	0.7271	0.0083	0.7284	0.0095	**0.7330**	0.0087	0.7319	0.0077	**0.7330**	0.0084
300	2	0.7868	0.0198	0.7895	0.0191	0.7896	0.0195	0.7891	0.0191	**0.7897**	0.0194
20	5	0.7713	0.0152	0.7715	0.0150	0.7741	0.0141	0.7732	0.0142	**0.7744**	0.0141
50	5	0.7522	0.0132	0.7536	0.0121	**0.7571**	0.0129	0.7556	0.0122	0.7566	0.0127
100	5	**0.7174**	0.0384	**0.7174**	0.0384	**0.7174**	0.0384	0.7173	0.0385	**0.7174**	0.0384
300	5	0.7244	0.0087	0.7255	0.0088	0.7292	0.0086	0.7277	0.0082	**0.7297**	0.0077
20	6	0.7830	0.0147	0.7866	0.0140	**0.7876**	0.0131	0.7856	0.0136	0.7870	0.0140
50	6	0.7692	0.0161	0.7685	0.0140	**0.7726**	0.0153	0.7692	0.0151	0.7709	0.0151
100	6	0.7504	0.0135	0.7506	0.0132	0.7545	0.0129	0.7519	0.0122	**0.7554**	0.0120
300	6	0.7200	0.0086	0.7212	0.0089	0.7249	0.0085	0.7246	0.0085	**0.7252**	0.0083
20	7	0.7812	0.0160	**0.7865**	0.0143	0.7859	0.0153	0.7854	0.0146	0.7856	0.0144
50	7	0.7645	0.0155	0.7664	0.0155	0.7698	0.0148	0.7698	0.0158	**0.7715**	0.0144
100	7	0.7452	0.0134	0.7485	0.0131	0.7514	0.0136	0.7492	0.0124	**0.7516**	0.0136
300	7	0.7164	0.0093	0.7185	0.0093	0.7200	0.0086	0.7212	0.0096	**0.7232**	0.0084
20	8	0.7773	0.0166	0.7826	0.0182	0.7820	0.0171	0.7826	0.0159	**0.7837**	0.0166
50	8	0.7183	0.0246	0.7183	0.0246	**0.7184**	0.0245	0.7183	0.0246	**0.7184**	0.0245
100	8	0.7619	0.0152	**0.7669**	0.0155	0.7662	0.0154	0.7651	0.0135	0.7696	0.0147
300	8	0.7439	0.0132	0.7447	0.0136	0.7485	0.0127	0.7460	0.0125	**0.7487**	0.0110
20	9	0.7136	0.0083	0.7150	0.0081	0.7176	0.0084	0.7164	0.0078	**0.7182**	0.0088
50	9	0.7211	0.0180	0.7209	0.0178	**0.7213**	0.0175	0.7211	0.0177	**0.7213**	0.0176
100	9	0.7116	0.0091	0.7117	0.0089	0.7132	0.0090	0.7129	0.0089	**0.7137**	0.0090
300	9	**0.7421**	0.0293	0.7420	0.0293	0.7421	0.0292	**0.7421**	0.0292	**0.7421**	0.0292
20	10	0.7457	0.0202	0.7462	0.0200	**0.7469**	0.0199	0.7465	0.0197	0.7466	0.0202
50	10	0.7405	0.0171	0.7412	0.0168	0.7417	0.0169	0.7415	0.0172	**0.7419**	0.0169
100	10	0.7271	0.0105	0.7282	0.0095	0.7301	0.0102	0.7300	0.0097	**0.7313**	0.0094
300	10	0.7843	0.0242	0.7852	0.0232	**0.7855**	0.0234	0.7853	0.0234	**0.7855**	0.0234

Table 6.11: Wilcoxon test - fitness value comparison: NK landscapes 100000 generations, tselk 10

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	0.7492	0.0295	10.90			1.0000	1.0000	1.0000	1.0000
tnd	0.7506	0.0301	13.22	56407	0.0000		1.0000	1.0000	1.0000
tnr	0.7523	0.0297	14.14	59074	0.0000	0.0000		0.0000	0.9999
trd	0.7515	0.0294	18.11	56477	0.0000	0.0000	1.0000		1.0000
trr	0.7528	0.0295	20.07	58754	0.0000	0.0000	0.0001	0.0000	

ized structures and solution suggestions cause mutations on different places of the chromosome and make the mutation more efficient. Also the count of visited solutions within the same number of generations influence the results in a positive way.

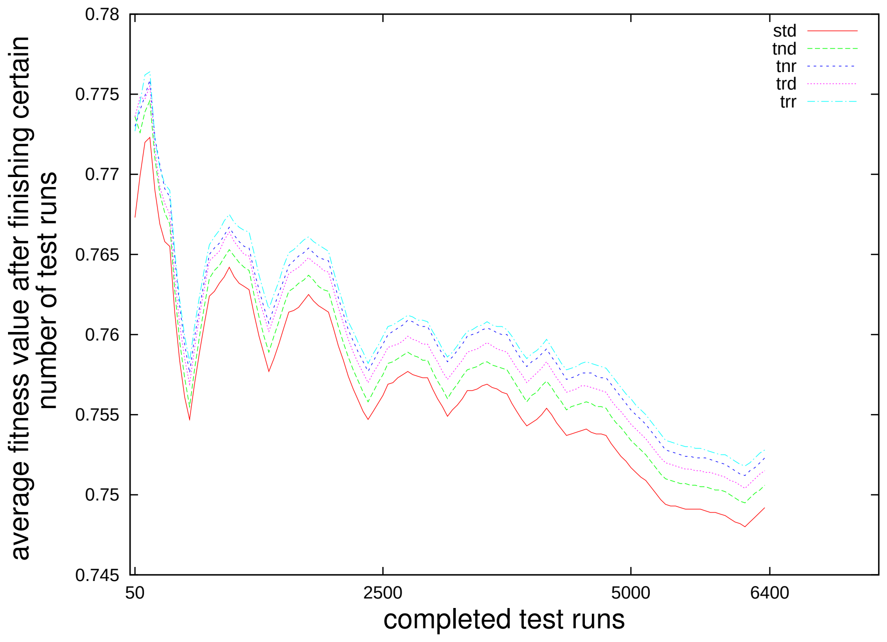


Figure 6.6: NK, 100000 generations, tselk 10 - average fitness value after finishing certain number of test runs

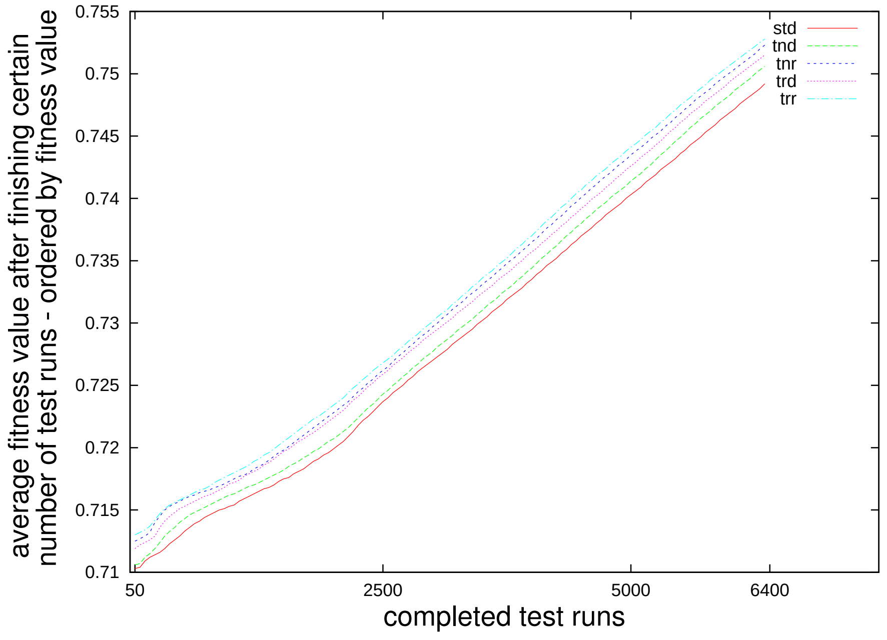


Figure 6.7: NK, 100000 generations, tselk 10 - average fitness value after finishing certain number of test runs -ordered by fitness value

Test 2

To study the influence of the GA parameters on the test results I performed the same tests again with only the *tselk* parameter changed. Instead of the value 10, I used the value 2. All other parameters were same as before. With the obtained results I did the same comparisons, plotting the same graph types. In these graphs there can also be seen the difference between the particular algorithm types. Figure 6.8 displays the achieved results, Figure 6.9 diplays their averaged values, and Figure 6.10 displays these averaged values ordered from least to greatest.

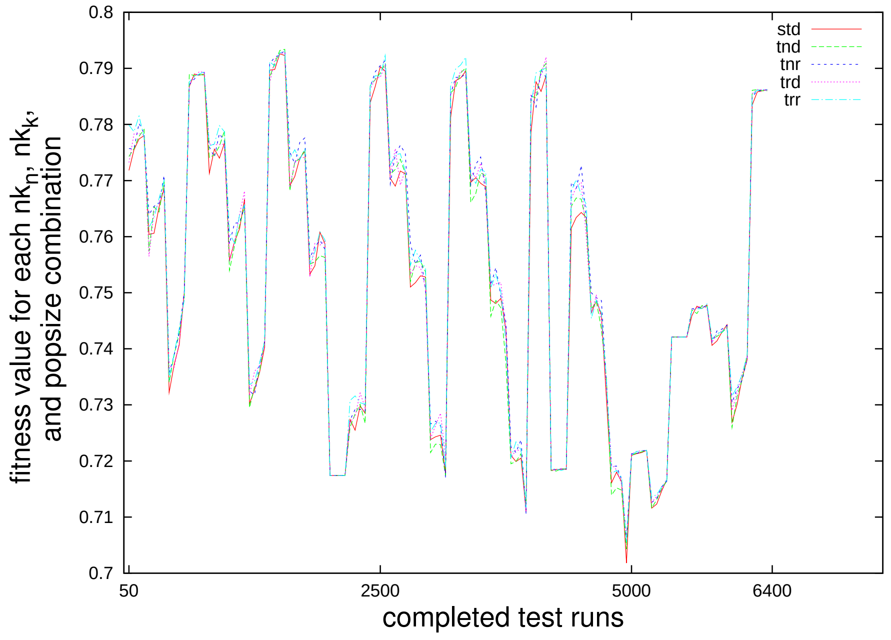


Figure 6.8: NK, 100000 generations, tselk 2 - fitness values for each combination of popsize, *nk_n* and *nk_k*

Detailed results of this test are displayed in Table 6.12 and in Table 6.13. Both tables display the same results, but Table 6.12 summarizes all *nk_n* values for each *nk_k* value. The columns display mean values and standard deviations of the results.

Table 6.12: Test results: NK landscapes, 100000 generations, tselk 2

	std		tnd		tnr		trd		trr	
k	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
1	0.7670	0.0230	0.7678	0.0227	0.7686	0.0225	0.7676	0.0228	**0.7688**	0.0227
2	0.7652	0.0251	0.7656	0.0252	0.7666	0.0252	0.7664	0.0246	**0.7671**	0.0247
5	0.7436	0.0309	0.7435	0.0308	0.7449	0.0315	0.7441	0.0308	**0.7452**	0.0312
6	0.7582	0.0272	0.7590	0.0279	**0.7607**	0.0271	0.7596	0.0265	**0.7607**	0.0272
7	0.7554	0.0287	0.7550	0.0290	0.7571	0.0284	0.7569	0.0291	**0.7573**	0.0291
8	0.7525	0.0304	0.7537	0.0314	**0.7548**	0.0311	0.7541	0.0316	0.7545	0.0314
9	0.7226	0.0218	0.7224	0.0218	**0.7233**	0.0216	**0.7233**	0.0214	**0.7233**	0.0214
10	0.7518	0.0272	0.7522	0.0270	0.7525	0.0266	0.7524	0.0269	**0.7527**	0.0266

I summarized all achieved results again and performed a Wilcoxon test to compare them. Table 6.14 displays the results of the comparison. In this case, we can also order all algorithms by the quality of achieved results from best to worst

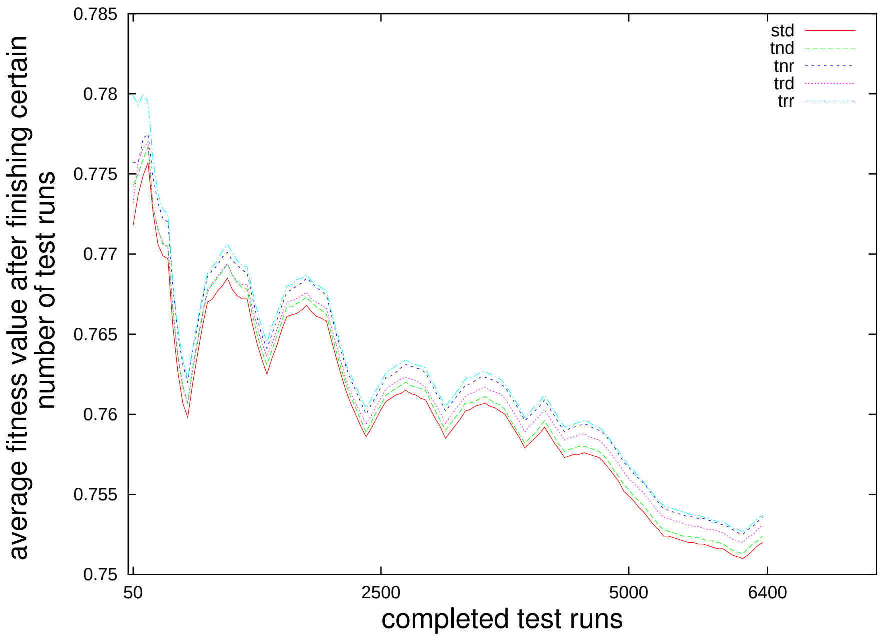


Figure 6.9: NK, 100000 generations, tselk 2 - average fitness value after finishing certain number of test runs

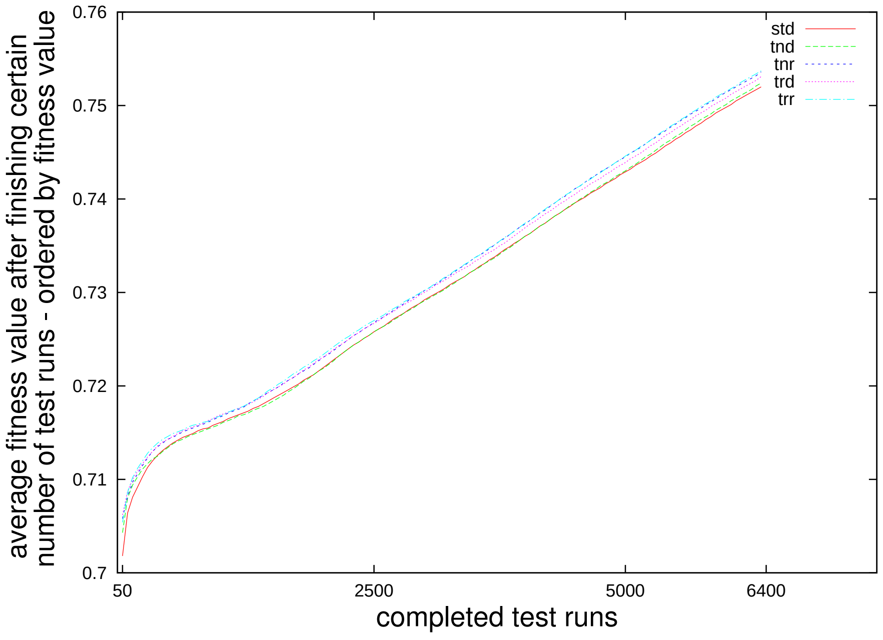


Figure 6.10: NK, 100000 generations, tselk 2 - average fitness value after finishing certain number of test runs -ordered by fitness value

Table 6.13: Test results: NK landscapes, 100000 generations, tselk 2

		std		tnd		tnr		trd		trr	
k	n	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
1	20	0.7757	0.0158	0.7767	0.0146	0.7775	0.0151	0.7770	0.0159	**0.7795**	0.0147
1	50	0.7638	0.0122	0.7642	0.0134	**0.7665**	0.0134	0.7638	0.0129	0.7655	0.0132
1	100	0.7400	0.0104	0.7412	0.0104	**0.7419**	0.0101	0.7412	0.0099	0.7416	0.0095
1	300	0.7884	0.0181	**0.7889**	0.0179	0.7884	0.0181	0.7886	0.0180	0.7886	0.0179
2	20	0.7744	0.0144	0.7759	0.0136	0.7764	0.0142	0.7765	0.0131	**0.7778**	0.0133
2	50	0.7607	0.0123	0.7600	0.0126	**0.7623**	0.0123	0.7617	0.0128	**0.7623**	0.0117
2	100	0.7348	0.0091	0.7351	0.0091	0.7357	0.0088	0.7363	0.0087	**0.7367**	0.0095
2	300	0.7911	0.0194	0.7915	0.0193	**0.7921**	0.0195	0.7911	0.0193	0.7919	0.0191
5	20	0.7722	0.0138	0.7727	0.0136	**0.7756**	0.0128	0.7726	0.0137	0.7747	0.0132
5	50	0.7569	0.0116	0.7559	0.0113	0.7578	0.0114	0.7573	0.0115	**0.7582**	0.0128
5	100	0.7174	0.0384	0.7174	0.0384	0.7174	0.0384	0.7174	0.0384	0.7174	0.0384
5	300	0.7278	0.0088	0.7278	0.0091	0.7286	0.0083	0.7291	0.0088	**0.7307**	0.0083
6	20	0.7876	0.0135	0.7888	0.0134	**0.7895**	0.0136	0.7884	0.0136	0.7891	0.0133
6	50	0.7706	0.0148	0.7720	0.0135	**0.7738**	0.0135	0.7719	0.0136	0.7734	0.0141
6	100	0.7521	0.0125	0.7542	0.0127	0.7553	0.0110	0.7533	0.0122	**0.7556**	0.0136
6	300	0.7226	0.0088	0.7212	0.0090	0.7243	0.0092	**0.7249**	0.0083	0.7244	0.0089
7	20	0.7867	0.0141	0.7879	0.0138	0.7878	0.0139	0.7882	0.0146	**0.7899**	0.0139
7	50	0.7697	0.0146	0.7690	0.0133	0.7710	0.0137	**0.7712**	0.0151	0.7710	0.0137
7	100	0.7473	0.0127	0.7448	0.0132	**0.7500**	0.0122	0.7495	0.0130	0.7489	0.0125
7	300	0.7181	0.0095	0.7182	0.0089	**0.7196**	0.0097	0.7188	0.0098	0.7194	0.0092
8	20	0.7852	0.0162	0.7876	0.0170	0.7866	0.0163	**0.7880**	0.0167	**0.7880**	0.0160
8	50	0.7185	0.0245	0.7184	0.0246	0.7185	0.0245	0.7185	0.0245	0.7185	0.0245
8	100	0.7632	0.0140	0.7654	0.0151	**0.7685**	0.0136	0.7663	0.0151	0.7679	0.0144
8	300	0.7430	0.0145	0.7434	0.0138	**0.7454**	0.0143	0.7435	0.0142	0.7438	0.0132
9	20	0.7130	0.0103	0.7121	0.0094	**0.7151**	0.0106	0.7150	0.0093	0.7146	0.0095
9	50	0.7214	0.0177	0.7215	0.0178	0.7215	0.0176	**0.7217**	0.0175	0.7216	0.0176
9	100	0.7138	0.0091	0.7140	0.0090	0.7144	0.0090	0.7145	0.0091	**0.7149**	0.0088
9	300	0.7421	0.0292	0.7421	0.0292	0.7421	0.0292	0.7421	0.0292	0.7421	0.0292
10	20	0.7471	0.0197	0.7471	0.0193	**0.7474**	0.0193	0.7472	0.0194	0.7473	0.0198
10	50	0.7423	0.0170	0.7427	0.0162	**0.7430**	0.0166	0.7427	0.0170	0.7429	0.0166
10	100	0.7325	0.0103	0.7327	0.0100	**0.7340**	0.0091	0.7337	0.0097	0.7348	0.0092
10	300	0.7854	0.0235	0.7861	0.0230	0.7858	0.0233	**0.7860**	0.0232	0.7859	0.0230

(1. trr, 2. tnr, 3. trd, 4. tnd, 5. std). However, the difference between the trr and tnr algorithm is no more significant, but the trr algorithm still achieves the better results. In this case, it seems that the randomized structure begins to loose its advantage.

Table 6.14: Wilcoxon test - fitness value comparison: NK landscapes 100000 generations, tselk 2

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	0.7520	0.0300	11.29			0.9799	1.0000	1.0000	1.0000
tnd	0.7524	0.0303	13.84	41658	0.0201		1.0000	1.0000	1.0000
tnr	0.7536	0.0301	14.45	43072	0.0000	0.0000		0.0003	0.8542
trd	0.7531	0.0300	18.97	41343	0.0000	0.0000	0.9997		1.0000
trr	0.7537	0.0302	20.30	42766	0.0000	0.0000	0.1458	0.0000	

The tests have shown relevance of mutation rate also for this parameter settings. However, the contribution of the ealib trie is no more so high. This is probably caused by the lower selection pressure (*tselk* = 2 instead of 10). Lower selection pressure makes the revisits more rare. Therefore also the count of visited solutions within the same number of generations influence cannot influence the results with the same relevance.

Looking at the mean values of the results, I have realized that there is a significant difference compared to Test 1. Because of this, I have put results of both test runs into same graphs to be able to compare the differences between them.

Test comparison

First, I compared the achieved results of both *std* algorithm types. This is displayed in Figure 6.11. In this figure we can see that the std algorithm with *tselk* 2 achieved better results in most test runs.

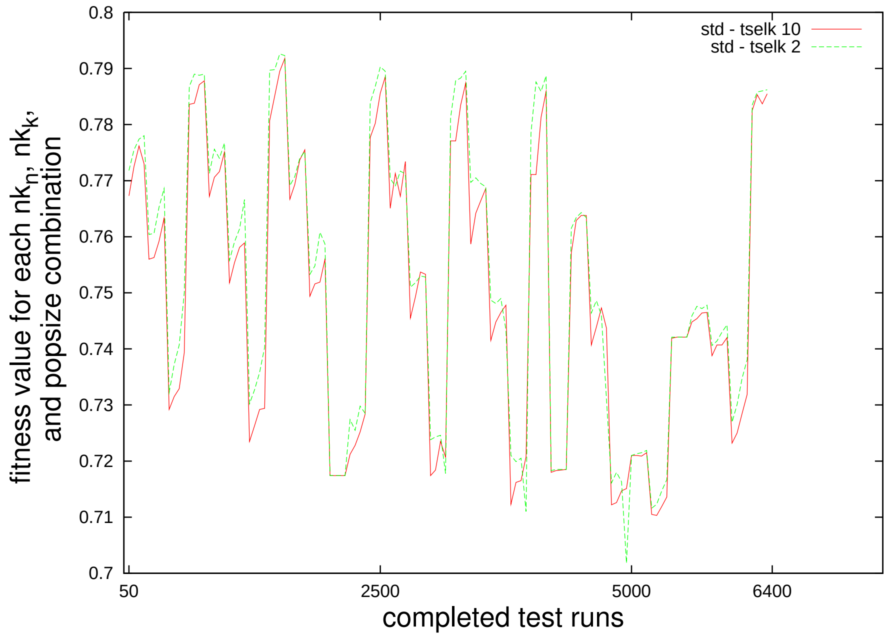


Figure 6.11: NK std, 100000 generations, comparison of *tselk* 2 and *tselk* 10 - fitness values for each combination of popsize, *nk_n* and *nk_k*

Figure 6.12 displays the comparison of two trr algorithm types. In this figure we can also see that the algorithm with *tselk* 2 achieved better results in most test runs. In this case the difference between the results is smaller, but still significant.

In the next step (Figure 6.13) I compared the averaged results in the same way, as in previous sections. This comparison shows the dominance of the algo-

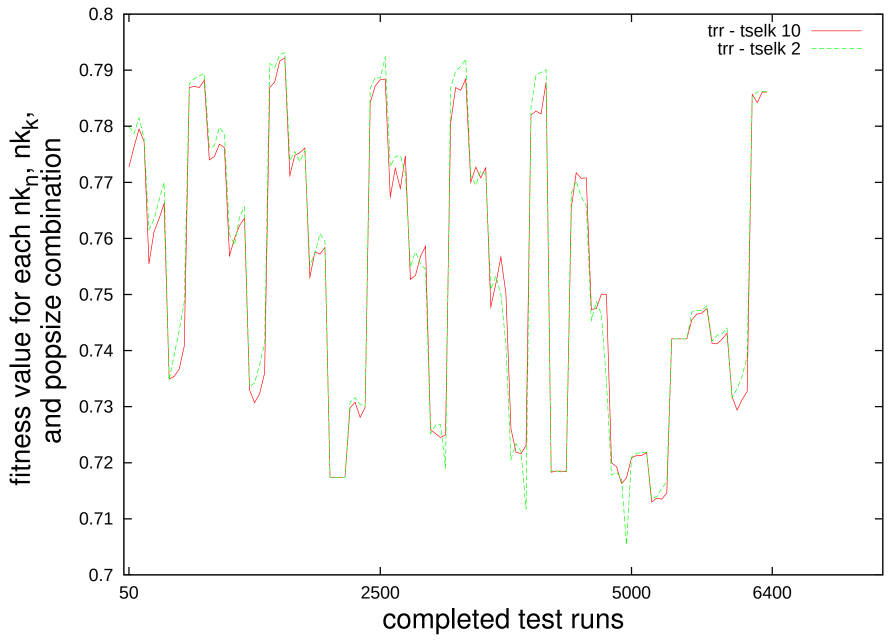


Figure 6.12: NK trr, 100000 generations, comparison of *tselk* 2 and *tselk* 10 - fitness values for each combination of popsize, *nk_n* and *nk_k*

rithms with *tselk* = 2 setting. The *std* algorithm with *tselk* = 2 type also achieved better results than the *trr* with *tselk* = 10.

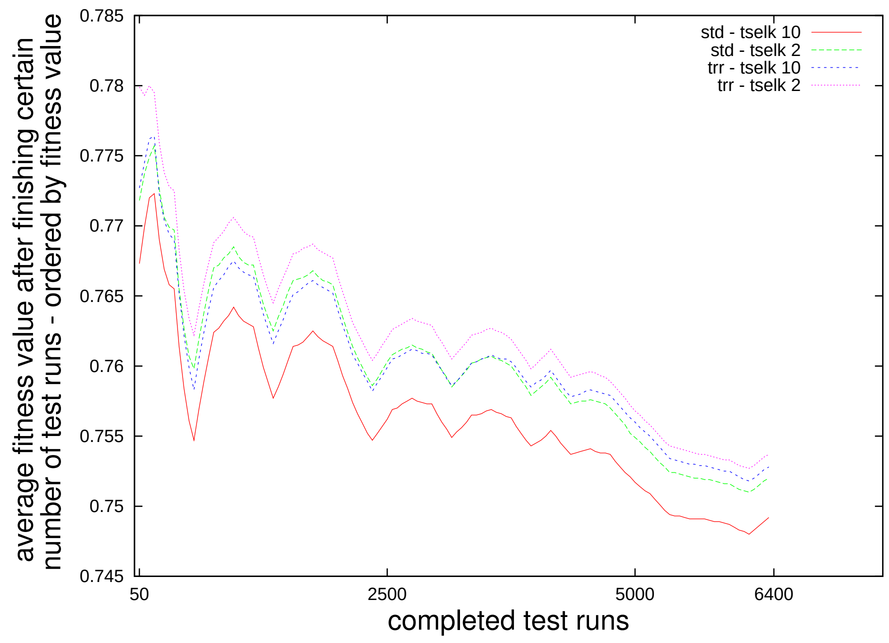


Figure 6.13: NK std, trr, 100000 generations, comparison of *tselk* 2 and *tselk* 10 - average fitness values after finishing certain number of test runs

This indeed brings us to an important conclusion. Using the archive can positively influence the results achieved by the GA, but the standard GA parameters still have the major influence on GA performance.

6.4 Experiments with MAX-SAT problem

The next set of tests were performed with MAX-SAT problem instances. A closer description of the MAX-SAT problem is written in Section 4.4. Unlike the other test functions, this problem is not intended to be solved with genetic algorithms. The tests were performed on two types of test instances:

- AIM instances - instances solved better with complete algorithms, but harder for local search algorithms because of big plateaus
- II instances - instances more difficult for complete algorithms, but easier for local search algorithms

6.4.1 Paramaters for MAX-SAT problem

There are three no special parameters defined for the MAX-SAT problem. The only uncommon parameter is the instance type, which should be solved.

6.4.2 Test results - MAX-SAT problem

Test 1 - AIM instances

Table 6.15: MAX-SAT test - AIM instances - parameter settings

paramater name	possible values
algorithm type	std, tnd, tnr, trd, trr
tgen	1000, 5000
tselk	4
popsize	200
instance type	AIM
instance count	8
runs per instance	50

The testing of the algorithms on the AIM instances was done with the parameters listed in Table 6.15. I performed 50 test runs with parameter on each of the eight AIM instances which are described in Section 4.4. I performed two different tests. In the first, each test run lasted 1000 generations. The second test was performed with 5000 generations. Every solution gained fitness value according to the number of unsatisfied instances. This problem should be minimized. That means that the better results are the lower ones. Therefore, the *p-value* of the Wilcoxon test needs to be interpreted inversely.

Table 6.16: Wilcoxon test - fitness value comparison: MAX-SAT - AIM instances, 1000 generations

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	5.7350	1.3581	0.07			0.0438	0.2150	0.0304	0.4898
tnd	5.5625	1.4039	0.11	4.55	0.9562		0.9347	0.4122	0.9730
tnr	5.6775	1.3958	0.11	4.40	0.7852	0.0654		0.0569	0.7705
trd	5.5600	1.3513	0.17	4.55	0.9696	0.5881	0.9431		0.9775
trr	5.7350	1.4141	0.17	4.28	0.5104	0.0270	0.2297	0.0225	

Table 6.17: Wilcoxon test - fitness value comparison: MAX-SAT - AIM instances, 5000 generations

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	1.7725	0.8200	0.16			0.4684	0.2800	0.4567	0.4817
tnd	1.7675	0.8247	0.24	60.10	0.5321		0.2985	0.4969	0.5097
tnr	1.7450	0.8010	0.24	64.08	0.7204	0.7020		0.7796	0.7245
trd	1.7700	0.7639	0.39	58.78	0.5438	0.5037	0.2209		0.5103
trr	1.7725	0.7889	0.38	60.69	0.5189	0.4909	0.2759	0.4903	

Table 6.16 and Table 6.17 display the test results. For this problem, the GA with the use of the archive does not achieve significantly better results in most cases. This is probably caused by the AIM instances structure. Wide plateaus make finding of the optimal solution harder and 100 variables are too many for the GA with the use of the archive to gain the significant advantage. The number of revisits in the case of MAX-SAT problem is very low, probably because many solutions have the same fitness value. The large search space and the low efficiency of mutation are the main reasons for bad success of ealib archive in this test.

Test 2 - II instances

The testing of the algorithms on the II instances was done with the parameters listed in Table 6.18. I performed 50 test runs with parameter on each of the eight II instances which are described in Section 4.4. I made two different tests. In the first,

Table 6.18: MAX-SAT test - II instances - parameter settings

paramater name	possible values
algorithm type	std, tnd, tnr, trd, trr
tgen	1000, 5000
tselk	4
popsize	200
instance type	II
instance count	8
runs per instance	50

each test run lasted 1000 generations. The second test was performed with 5000 generations. Again, every solution gained fitness value according to the number of unsatisfied instances.

Table 6.19: Wilcoxon test - fitness value comparison: MAX-SAT - II instances, 1000 generations

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	35.0475	32.8319	0.31			0.0243	0.0188	0.0136	0.0076
tnd	34.3225	31.4474	0.41	4.48	0.9757		0.1440	0.1516	0.2984
tnr	33.6250	31.0560	0.41	4.42	0.9812	0.8561		0.4627	0.4581
trd	33.6800	30.8307	0.58	4.50	0.9864	0.8486	0.5375		0.5941
trr	33.9325	31.4162	0.58	4.39	0.9924	0.7018	0.5421	0.4061	

Table 6.20: Wilcoxon test - fitness value comparison: MAX-SAT - II instances, 5000 generations

alg	mean	sd	time	revisits	w_p-vs.std	w_p.vs.tnd	w_p-vs.tnr	w_p-vs.trd	w_p.vs.trr
std	6.8500	5.0736	0.91			0.0024	0.0707	0.1110	0.5470
tnd	6.5025	4.6931	1.19	76.68	0.9976		0.7771	0.8686	0.9740
tnr	6.6275	4.8794	1.21	73.70	0.9295	0.2233		0.6674	0.9262
trd	6.6750	4.9193	1.70	74.64	0.8893	0.1317	0.3331		0.8606
trr	6.8025	4.9921	1.69	73.47	0.4535	0.0260	0.0740	0.1396	

Table 6.19 and Table 6.20 display the average test results for all 50 runs for each algorithm type and their Wilcoxon test comparison. The results show that the performance of the GA with the archive is significantly better than without the GA. However, in the test with 5000 generations we can see that the archives, which use randomized trie structure have worse performance than the other archives. Also, the *trr* archive achieves only the results of the *std* algorithm. This is probably only a coincidence, because the number of revisits in comparison to number of visited solutions is very low. Searching in the "promissing" parts of the search space brings better results than small search allover in the search space. But this turns out to be impossible in this case, because there are wide plateaus of solutions

with the same fitness value in the sarch space. Therefore the GA does not provide the needed selection pressure.

6.5 Evaluation

The tests have shown dominance of the GAs with the use of the ealib archive in most test cases. The archive has brought significant contributions especially when it was used for the solving of GA-specific problems (Royal Road function and NK landscapes problem).

GAs with the use of the trie brought equivalent or worse results in the cases where the discovery of the better solutions was more or less random (Royal Road function with higher *rrbase* parameter) or where the number of revisits was too small in comparison to the search space size (MAX-SAT problem with a very large amount of variables).

The main advantage of the ealib trie is that it visits more solutions within same number of generations. The GA can profit out of this property when the ratio between the number of visited solutions and solutions in the search space is relevant. The second advantage of the ealib trie archive is higher mutation rate in the most visited parts of the search space. The GA can take advantage out of this when the mutation has higher chances to influence the fitness value of the solution. Therefore, the algorithm with the use of the ealib archive achieves better results easier in the cases with higher selection pressure.

As expected, the use of the ealib archive insures not that the GA finds the best solutions in general for the given problem, but that the use of it can lead only to a betterment of the solutions found by the standard GA. The standard GA parameters like population size, tournament size, number of generations, etc. still have a major influence on the fitness values of the found solutions.

Even though the ealib archive does not give us the general insurance of finding the best possible solution, its use can significantly increase the chances of finding it. Therefore, the use of this archive should at least be considered for every application of the GA.

Chapter 7

Conclusion

In this work, I have presented a complete archive for GAs, named ealib trie. It is an archive based on the trie structure, which can be used to enhance any GA. Its purpose is to store all visited solutions, avoid revisits of the solutions by suggesting unvisited ones, and in this manner improve the ability to find a better solution for each GA.

After giving a short introduction to the problem in Chapter 1, I described the previous work (Chapter 2). There, I described approaches which dealt with the problem of archiving the visited solutions or avoiding the revisits.

In Chapter 3, I have introduced and discussed the implemented structure of the archive. Based on a the short comparison, I chose the trie data structure as the most fitting structure for this purpose. Further, I have introduced how the visited solutions are stored in the most efficient way. Also the mechanism of the handling of revisits was presented. In this part I also introduced special properties and the functionality of the ealib archive. These were randomized structure and special algorithms for the handling of the revisits.

The test problems, which I used for performance comparison of the GA with and without use of archive, have been described in Chapter 4.

A description of the implementation of the ealib trie archive has been given in Chapter 5. Here, the basic packages and object structures have been described.

Finally, the performed tests and their results were discussed in Chapter 6. In this chapter, I have compared the results achieved by the standard GA to the results achieved by GAs which have used the ealib trie archive.

The tests have shown dominance of the GAs with the use of the ealib archive in most test cases. The archive had brought significant contributions especially when it was used for the solving of GA-specific problems (Royal Road function and NK landscapes problem). In general the use of the ealib trie can be recommended. The achieved improvement depends also on other GA parameters. For some constellations it can bring a betterment with higher probability. For example, when the selection pressure is higher, then the using of the trie has a higher contribution to the overall result.

The main advantage of the ealib trie is that it visits more solutions within same number of generations. The GA can profit out of this property when the ratio between the number of visited solutions and solutions in the search space is relevant. The second advantage of the ealib trie archive is higher mutation rate in the most visited parts of the search space. The GA can take advantage out of this when the mutation has higher chances to influence the fitness value of the solution.

The achieved test results also prove that the implemented improvements (randomized trie structure, suggestion changed in a random place) reach better results than the standard ealib archive without use of these improvements.

Some results have also shown that the contribution of the ealib archive does not always have positive effects. Tests on the MAX-SAT problem revealed that the betterment of the GA cannot always be ensured. The main reason for it are huge plateaus in the search space that contain solutions with the same fitness value. Because of these plateaus the revisits are very rare and the mutation operator has only little influence on the quality of the solution.

It could be interesting to test the archive for GAs which are combined with another metaheuristic in the future. For example, to use the ealib archive for GAs enhanced by local search algorithm [64].

Appendix A

Bibliography

[1] M. Abrash. *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash*. Coriolis Group Books, Scottsdale, AZ, USA, 1997.

[2] G. M. Adelson-Velskii and E. Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

[3] L. Altenberg. B2.7.2. nk fitness landscapes. In Bäck et al. [6].

[4] M. E. Aydin and T. C. Fogarty. A distributed evolutionary simulated annealing algorithm for combinatorial optimisation problems. *Journal of Heuristics*, 10(3):269–292, 2004.

[5] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Jan. 1996.

[6] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Computational Intelligence Library. Oxford University Press in cooperation with the Institute of Physics Publishing, Bristol, New York, ring-bound edition, Apr. 1997.

[7] R. Battiti. Reactive search: Toward self–tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, pages 61–83. John Wiley & Sons Ltd., Chichester, 1996.

[8] Bentley and Sedgewick. Fast algorithms for sorting and searching strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[9] D. Boughaci, H. Drias, and B. Benhamou. Solving max-sat problems using a memetic evolutionary meta-heuristic. In *Proceedings of the IEEE International conference on Cybernetics and Intelligent Systems, CIS-2004*, pages 480 – 484, December 2004.

[10] B. Cha and K. Iwama. Adding new clauses for faster local search. In *AAAI/IAAI, Vol. 1*, pages 332–337, 1996.

[11] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (2nd Ed.)*. MIT Press, Cambridge, MA, 2001.

[13] R. C. Eberhart, Y. Shi, and J. Kennedy. *Swarm Intelligence*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, March 2001.

[14] A. E. Eiben, Z. Michalewicz, M. Schoenauer, and J. E. Smith. Parameter control in evolutionary algorithms. In Lobo et al. [35], pages 19–46.

[15] J. E. Fieldsend, R. M. Everson, and S. Singh. Using unconstrained elite archives for multiobjective optimization. *IEEE Trans. Evolutionary Computation*, 7(3):305–323, 2003.

[16] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, USA, Oct. 1966.

[17] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[18] V. B. Gantovnik, C. M. Anderson-Cook, Z. Gürdal, and L. T. Watson. A genetic algorithm with memory for mixed discrete-continuous design optimization. *Computers & Structures*, 81:2003 – 2009, August 2003.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[20] D. E. Goldberg. Genetic algorithms and walsh functions: Part II, deception and its analysis. In *Complex systems 3*, pages 153–171, 1989.

[21] D. E. Goldberg. Construction of high-order deceptive functions using low-order walsh coeficients. IlliGAL Report 90002, Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illi-

nois, Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, IL 61801, USA, 1990.

[22] J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern.*, 16(1):122–128, 1986.

[23] P. Hartono, S. Hashimoto, and M. Wahde. Labeled-ga with adaptive mutation rate. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1851–1858. IEEE Press, 2004.

[24] J. H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM*, 9(3):297–314, 1962.

[25] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975. Reprinted by MIT Press, April 1992.

[26] B. D. Hughes. *Random Walks and Random Environments: Volume 1: Random Walks*. Oxford University Press, USA, May 16, 1995.

[27] K. Iwama and K. Hino. Random generation of test instances for logic optimizers. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 430–434, New York, NY, USA, 1994. ACM.

[28] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[29] H. A. Kamal. Optimal control of fed batch fermentation processes using adaptive genetic algorithm. In *Proceedings of the Twentieth National Radio Science Conference. NRSC 2003*, volume C1, pages 1–11, Cairo, Egypt, 2003.

[30] A. Kamath, N. Karmarkar, K. Ramakrishnan, and M. Resende. A continuous approach to inductive inference. *Mathematical Programming*, 57:215–238, 1992.

[31] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, May 1993.

[32] J. Knowles and D. Corne. The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 98–105, Mayflower Hotel, Washington D.C., USA, June-September 1999. IEEE Press.

[33] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

[34] A. H. Land and Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[35] F. G. Lobo, C. F. Lima, and Z. Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*. Springer, 2007.

[36] S. J. Louis and G. Li. Combining robot control strategies using genetic algorithms with memory. In P. J. Angeline, R. G. Reynolds, J. R. McDonnell, and R. Eberhart, editors, *Evolutionary Programming VI*, pages 431–441, Berlin, 1997. Springer.

[37] M. L. Mauldin. Maintaining diversity in genetic search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 247–250, 1984.

[38] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, second, revised and extended edition, Dec. 2004.

[39] B. L. Miller and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. IlliGAL Report 95006, Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois, Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, IL 61801, USA, 1995.

[40] M. Mitchell, S. Forrest, and J. H. Holland. The Royal road for genetic algorithms: Fitness landscapes and GA performance. In *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 245–254, 1991.

[41] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical report, Caltech Concurrent Computation Program, 1989.

[42] A. Neumaier. Global optimization and constraint satisfaction. In I. Bomze, I. Emiris, A. Neumaier, and L. Wolsey, editors, *Proceedings of GICOLAG workshop (of the research project Global Optimization, Integrating Convexity, Optimization, Logic Programming and Computational Algebraic Geometry)*, Dec. 2006.

[43] P. M. Pardalos, N. V. Thoai, and R. Horst. *Introduction to Global Optimization*. Nonconvex Optimization and Its Applications. Springer, second edition, Dec. 31, 2000. First edition: June 30, 1995, ISBN: 978-0792335566.

[44] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. The Addison-Wesley series in artificial intelligence. Addison-Wesley Pub (Sd), Apr. 1984.

[45] S. Rana. *Examining the role of local optima and schema processing in genetic search*. PhD thesis, Fort Collins, CO, USA, 1999. Adviser-Darrell Whitley.

[46] S. Rana and D. Whitley. Genetic algorithm behavior in the maxsat domain. In *In Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, pages 785–794. Springer, 1998.

[47] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, Dec. 1996.

[48] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart, 1973. his dissertation from 1970.

[49] S. Ronald. Preventing diversity loss in a routing genetic algorithm with hash tagging. In R. Stonier and X. H. Yu, editors, *Complex Systems: Mechanism of Adaption*, pages 133–140, Amsterdam, 1994. IOS Press.

[50] S. Ronald. Duplicate genotypes in a genetic algorithm. In D. B. Fogel, H.-P. Schwefel, T. Bäck, and X. Yao, editors, *IEEE World Congress on Computational Intelligence (WCCI'98)*, pages 793–798, Piscataway NJ, 1998. IEEE Press.

[51] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, Dec. 2002.

[52] H.-P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley and Sons Ltd, New York, NY, USA, June 17, 1981.

[53] B. Skellett, B. Cairns, N. Geard, B. Tonkes, and J. Wiles. Maximally rugged nk landscapes contain the highest peaks. In *In Genetic and Evolutionary Computation Conference*, pages 579–584, 2005.

[54] J. Smith and T. C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *International Conference on Evolutionary Computation*, pages 318–323, 1996.

[55] T. Stützle, H. Hoos, and A. Roli. A review of the literature on local search algorithms for MAX-SAT. Technical report, Intellectics Group, Darmstadt University of Technology, Germany, Feb. 2001.

[56] R. K. Thompson and A. H. Wright. Additively decomposable fitness functions. Technical report, Computer Science Department, The University of Montana, Missoula, MT 59812-1008, USA, 1996.

[57] R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In *In Parallel Problem Solving from Nature - PPSN V*, pages 97–106. Springer, 1998.

[58] B. Weinberg and E.-G. Talbi. Nfl theorem is unusable on structured classes of problems. *Evolutionary Computation, 2004. CEC2004. Congress on*, 1:220–226 Vol.1, June 2004.

[59] E. D. Weinberger. NP Completeness of Kauffman's N-k model, a tuneable rugged fitness landscape. Working Papers 96-02-003, Santa Fe Institute, Feb. 1996.

[60] T. Weise. *Global Optimization Algorithms - Theory and Application*. Thomas Weise, second edition, Aug. 29, 2008. Online available at http://www.it-weise.de/.

[61] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[62] J. R. Woodward and J. R. Neil. No free lunch, program induction and combinatorial problems. In C. Ryan, T. Soule, M. Keijzer, E. P. K. Tsang, R. Poli, and E. Costa, editors, *EuroGP*, volume 2610 of *Lecture Notes in Computer Science*, pages 475–484. Springer, 2003.

[63] S. Y. Yuen and C. K. Chow. A non-revisiting genetic algorithm. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 4583–4590, 2007.

[64] Y. Zeng and Y.-P. Wang. A new genetic algorithm with local search method for degree-constrained minimum spanning tree problem. In *ICCIMA '03: Proceedings of the 5th International Conference on Computational Intelligence and Multimedia Applications*, page 218, Washington, DC, USA, 2003. IEEE Computer Society.