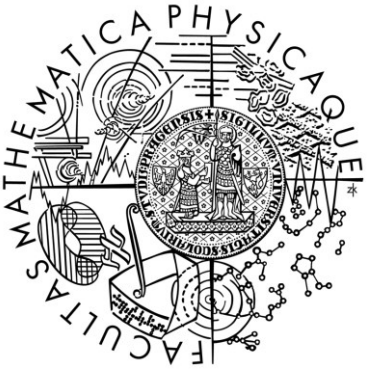
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Štěpán Havránek

Genetic Algorithms driven by MCTS

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: RNDr. Jan Hric

Study programme: Informatics

Specialization: Theoretical Computer Science

Prague 2015

I would like to thank my supervisor RNDr. Jan Hric, as it would be impossible to finish this thesis without his consultations and advices. I would also like to thank the entire MatFyz faculty for being the best computer science school in this republic.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

Název práce: Genetické algoritmy řízené MCTS

Autor: Štěpán Havránek

Katedra / Ústav: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Jan Hric, Katedra teoretické informatiky a matematické logiky

Abstrakt: Evoluční a genetické algoritmy jsou techniky navržené dle přírodní inspirace. Používají se k řešení nejrůznějších úloh, se kterými se neumíme efektivně vypořádat exaktními metodami. Metoda Monte Carlo, potažmo Monte Carlo Tree Search, je založena na vzorkování, a také se uplatňuje tam, kde nelze daný problém držet celý v paměti a úplné prohledávání není možné. Tato práce se zabývá návrhem spojení těchto dvou odlišných přístupů do jedné obecné metody. Tuto metodu ilustruje a implementuje na konkrétním případě: problému obchodního cestujícího (TSP). Součástí práce jsou i nejrůznější experimenty hledající vhodné nastavení parametrů, porovnávající různé varianty metody s klasickým evolučním přístupem k TSP nebo například hladovým algoritmem. Naše metoda se ukázala přinejmenším konkurenceschopná. Nejlepších výsledků potom dosahuje kooperace našeho přístupu s klasickým evolučním řešením TSP. Tato spolupráce dosahuje vyššího výkonu než každá její část samostatně, což považujeme za úspěch naší metody.

Klíčová slova: Monte Carlo, MCTS, UCT, Evoluční algoritmy, TSP

Title: Genetic Algorithms driven by MCTS

Author: Štěpán Havránek

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: RNDr. Jan Hric, Department of Theoretical Computer Science and Mathematical Logic

Abstract: Evolutionary and genetic algorithms are problem-solving methods designed according to a nature inspiration. They are used for solving hard problems that we cannot solve by any efficient specialized algorithm. The Monte Carlo method and its derivation the Monte Carlo Tree Search (MCTS) are based on sampling and are also commonly used for too complex problems, where we are dealing with enormous memory consumption and it is impossible to perform a complete searching. The goal of this thesis is to design a general problem solving method that is built from these two completely different approaches. We explain and implement the new method on one example problem: the Traveling salesman problem (TSP). Second part of this thesis contains various tests and experiments. We compare different settings and parametrizations of our method. The best performing variant is then compared with the classical evolutionary TSP solution or, for example, with greedy algorithms. Our method shows competitive results. The best results were achieved with the cooperation of our method and the classical evolutionary TSP solution. This union shows better results than any of its parts separately, which we find as a great success.

Keywords: Monte Carlo, MCTS, UCT, Evolutionary algorithms, TSP

Contents

Introduction

Computer science, from its beginning, deals with variety of problems and tries to find their computational solutions. Computer scientists develop algorithms and data structures in order to deliver the fastest and the most efficient solving methods that could be put into practice. Since computers play a big role in our everyday lives and the number of their applications is growing every day, the number of problems computer science is facing continues to rise.

Even though the computers' performance increases continuously, there are still a lot of challenging problems and puzzles. There are whole classes of problems that we cannot solve with any efficient algorithm. We often even do not know whether the efficient solution could exist. Another challenge is the case of problems that are not as complicated, but we are dealing with gigantic amount of input data. In the both cases, the computation usually requires enormous time or memory resources. For instance, when a traveling salesman is planning his business trip, he needs to visit 30 customers but he wants to save as much fuel as possible. It is impossible to find out which order of the places is the optimal one, by going thru all the possibilities and selecting the best one. Even if evaluating of one possibility takes one millisecond, the whole enumeration will take about 8.5×10^{21} years.

The very complex or the very large problems gave rise to various approaches that tries to find a practical compromise between the results' quality and the algorithm feasibility. One of the compromise-seeking method is the Monte Carlo Tree Search, which was originally developed for the purpose of playing complex games. Another, and completely different, compromise-seeking method are the Evolutionary algorithms. They were designed according the natural processes and it came as a very general problem solving approach. Both of these different approaches we are going to introduce hereunder.

Goal of this thesis

The core of this paper is to invent a problem-solving method that is a combination of Evolutionary algorithms and Monte Carlo Tree Search. Our method is designed as an add-in[1] for a genetic evolutionary algorithm. Therefore, it should be

[1] Removable additional module.

applicable on every problem that can be solved by the genetic evolutionary algorithm. The method we are developing we also implement and test on one example problem: Traveling salesman problem. All the implementation source codes and experiments results are available on the attached CD (see chap. Attachments).

Outline

In the first two chapters, we review our working background: Evolutionary/Genetic algorithms and Monte Carlo Tree Search. We explain the principles behind these approaches and we focus on the parts which our work is based on. The third chapter introduces the benchmark problem we have chosen: the Traveling salesman problem. The third chapter also contains an overview of genetic algorithm applications and we choose one of them as a baseline of our example implementation.

The fourth chapter is the flagship of this paper. It describes the whole mechanism of our newly proposed method. We divided our technique into three levels according to the complexity of the inner system. On every level, we have prepared two different approaches: *Direct* and *Repaired*. Therefore, each level contains two independent functional modules. Of course, all of the six modules can be variously parametrized or set. The fourth chapter introduces the techniques from the simplest level one to the most complex level three. Every module is explained as an application on the Traveling salesman problem and it fully corresponds with the attached implementation.

In the fifth chapter, we perform variety of measurements in order to find out the best parametrization and setting of our method. We comment and explain the experiments results. Already the first tests show that our method, with the right setting and parameters, is able to converge and can return good results. At the end of the fifth chapter, we select the better performing modules and in the next chapter, we try to improve and extend them. The best improvement turns out to be the cooperation of our method and the classical evolutionary approaches for Traveling salesman problem. This union of two different methods yields better results than any of its parts separately. That we consider as a great success of this thesis.

The last (seventh) chapter tests the best versions of our method with various inputs. This proves that the previous experiments were not only a coincidence. It shows that the results are similar with the special classes of inputs as well. In the performance

experiments, where we test the techniques in a limited time, we prove that the cooperation method is also the most practical version of them all.

At the end of this paper, we write some notes for the potential user of our method. We also present some ideas for the future research.

1. Evolutionary and genetic algorithms

1.1. Inspiration from nature

Evolutionary algorithms are huge class of computer science approaches to various types of problems. As the name suggests, evolutionary algorithms are based on inspiration by nature. Of course, mainly by *Evolution theory* itself and by *Evolution by nature selection* proposed by Charles Darwin [1] in the 19th century.

1.1.1. Evolution theory

In brief, *Evolution theory* describes the development of all living (flora and fauna) on the Earth. It observes that most of the living organisms came into life thanks to their parental organisms. What is more important, the child individual shares many of its biological and physiological properties with its parents.

The division into parents and children induces *generations* as groups of individuals born in the same era. This is very simple and natural principle. Nevertheless, the system of generations is very important in evolutionary algorithms, as we will see few paragraphs bellow.

Sharing the properties between parents and their children is called *heredity*. It provides some kind similarity between an individual and each of its parents (no matter what is the actual number of them). *Heredity* is very important in the *nature selection*. It claims that every generation is made up by individuals who are stronger, smarter and generally better than their parents are (or generally than their predecessors). This should be ensured right through the *nature selection* because only the individuals that have combination of properties good enough to survive, will have children. The principle of *heredity* provides preserving the high quality organism properties.

There is one more important element in the evolution, which is also used in the computer science application. Only the combination of the parents' properties sometimes is not enough for succeeding in life. The environment is changing all the time: climatic conditions are changing; surrounding flora and fauna is changing, etc. The second problem is how to build a stronger, smarter and better generation from the finite set of properties, which already turned out as the best constellation (presuming that there is no better combination of the given properties than the actual one). *Evolution theory* has an answer to these problems: *mutation*. *Mutation* is a change of the individual's particular property generally caused by an external, outer or unknown

factor. It brings a chance for individuals to acquire the trait that have not appeared in any generation yet. This new property can help the individual to survive in the changed environment, beat the other or, on the contrary, die sooner.

1.1.2. The genome and the DNA

In the previous subchapter, we have brought the not detailed overview of evolution principle, which was very abstract and did not tell us anything about the actual biological function of the living organisms or about their reproduction. All we understand at this point is that an individual consists of set of its properties, which determine his whole life course and are mutated and somehow recombined during reproduction.

To implement the evolution principle as an algorithm we use another inspiration from nature. The abstract set of properties, which determine the individual's life, is represented by the individual's genome information. The genome contains all the specification of the organism physiological form, look, growth or behavior (consequently). When the organism is developing, every step since the reproduction is influenced by the genome information. The reproduction is the very moment when the genome itself is built.

As it is usual in nature, the genome is not an atomic entity driving the organism's development. Since the 19th century, the biologists have discovered that the genome is made up of parts called chromosomes. The chromosomes we can divide into single genes, which are discrete units of heredity traits and consist of DNA information, which is simply coded by a sequence of nucleobases pairs [2]. The count of the nucleobase variations is a finite number, so it is very analogical to how we encode the information in our computers.

In this very simplified view, which we have presented, is possible to simulate the dynamic evolution just by using the genomes instead of the actual individuals grown based on these genomes. That is exactly the way the computer science is going.

1.2. Evolutionary algorithms

In computer science, there are a lot of problems that we cannot solve by a specialized efficient algorithm. These are either the problems of very high complexity, such as **NP**, **PSPACE**, **#P** or the other even more complex complexity classes [3]. There are also problems for which we do not know any optimal solving algorithm at

all. In these situations, we use suboptimal approximate algorithms or to incomplete heuristic algorithms.

Evolutionary algorithms are member of both of these groups of algorithms for hard-to-solve problems. The base idea is to let the problem solution come up from evolution process just like the nowadays organisms have developed from the prehistoric ones.

In the implementation of evolution process, we work with the candidate solutions of the particular problem – these are our individuals. To make the evolution happen, we need to know, how to reproduce and mutate the individuals. The answer is to create an encoded data structure, which will represent the particular solutions – the individuals. The evolution operations (reproduction and mutation) will be processed just as changes of this code[1]. This approach is the straightforward implementation of the genome inspiration. The algorithms representing possible candidate solutions by the code, which is changing in order to get better and better solution of the problem, are called Genetic algorithms [4]. The term of Genetic algorithms is usually used for Genetic algorithms driven by evolution. In other words, Genetic algorithms are Evolutionary algorithms over genome-represented individuals.

1.2.1. The algorithm

The evolutionary algorithm is simulating population of possible candidate solutions, which should develop thru generations into a good and useful result solution.

Evolutionary algorithm starts with the initial population[2] and begins the loop of life:

1. Parental selection – selects individuals who become parents of the next generation.
2. Reproduction – creating new individuals based on their parents.
3. Mutation – nondeterministic slight changing individuals in the new generation.
4. Environmental selection – fight for survival where usually only the individuals of high quality will outlive.

[1] The code of one concrete individual – its "genome".

[2] The initial population is usually randomly generated.

This loop needs a terminating criterion; otherwise, it would run endlessly. It is up to the concrete implementation whether to stop after given number of generations or to run until an individual better than the given limit appears in the population.

Evolutionary algorithm is actually a stochastic searching algorithm. Reproduction and mutation provide variability and the selections are driving the searching towards the optimal solution [5].

1.2.2. Operator-oriented implementation

There are a lot of various ways how to implement an Evolutionary/Genetic algorithm. The implementation can be very specific, targeted and optimized for solving the one particular problem. In our experiments, which are described few chapters bellow, we have chosen very generic implementation schema: Operator-oriented.

The operator-oriented schema simplifies the evolution life loop into two steps:

1. Apply the operators
2. Environmental selection

Here, the new generation making logic is put into operators, which can implement various mutation, recombination, parental selection, etc. The operators can be more or less problem-specific; nevertheless, all of them have to be compatible with the individual encoding. This approach is very useful for experiments because we can change or mix the operators and build the evolution process like Lego. What is more, we can test the operators separately to find out whether the particular operator helps to find a good solution or not.

The environmental selection, which should manage the "operators' products" to meet the optimal solution, stays unchanged.

1.2.3. Memetic algorithms, Lamarckian evolution, Baldwin effect

Evolutionary algorithms are a general framework for problem solving. There is a lot of papers that are introducing many extensions or improvements. Let us take a look on idea called *Memetic algorithms* [6].

In principle, the *Memetic algorithms* are combination of Evolutionary computing and local search, which is an incomplete solving method itself. The background inspiration is cultural development built thru generations. It should help the individuals to live better life. *Memetic algorithm* lets the individuals to learn

something before comes the environmental selection. A learned individual should then pass the selection better.

The actual learning is made by the local search. An individual that is a product of the evolution operators is then taken as a baseline for searching its vicinity in the problem solutions space. The searching should be fast and simple. It should not substitute the evolution operators, but only upgrade their results. If the quick search does find a better individual than the origin, we have more possible ways how to deal with that.

The *Lamarckian* [5] approach replaces the original individual with the better new searched one. This new specimen is going to fight in the environmental selection and then, if it survives, it will be processed by the operators instead of the original individual.

Another approach avoids interfering the evolution. It leaves the solution quest up to the operators. The only role of the searched better individual is to represent the origin's potential. It is used instead of the origin in the selection; therefore, the original individual has the quality of the best of its "neighbors". However, in the subsequent operators is still used the origin. This change of the individual's potential is called the *Baldwin effect* [5].

The goal of this thesis is to create an approach that will combine Evolutionary algorithms and Monte Carlo Tree Search. Our method, which we are going to introduce hereunder, is very close to the *Lamarckian Memetic evolution* idea.

2. Monte Carlo Tree Search

2.1. The Monte Carlo origins

The concept called *Monte Carlo* is older than the artificial intelligence field of science. *Monte Carlo* method is based on a pseudorandom simulation of a complex process where the pseudorandom decisions respect known particular probability distributions. Samples from this simulation, which has been run many times, are used for determining the process behavior [7].

Monte Carlo method is used widely across mathematics, physics or computer science. From numerical integration in mathematics to, for instance, *Monte Carlo Markov chain* state probability distribution determination in artificial intelligence. It is used mostly for the problems that are too complex or too huge so the exact methods would be very inefficient. The derivation of *Monte Carlo* method, which this thesis is based on, is a searching algorithm *Monte Carlo Tree Search* (MCTS)

2.1.1. Monte Carlo Go

The first step towards the MCTS, which this paper is benefiting from, was the use of the *Monte Carlo* method for creating an artificial player for the game Go [8]. The game Go [9] is known for its too high branching factor for complete searching. Due to this large number of game moves combinations it is very difficult to find out, in a reasonable time, what game results can come up from the current state (position). As a consequence, it is hard to evaluate the particular game states for usage in some heuristics. Bernd Brügmann brought to light an approach based on the *Monte Carlo* idea. He defines the quality of the actual state by the possible game ends that can result from it. It is not possible to efficiently enumerate all the endings. Brügmann deals with this by sampling. He simulates several pseudorandom "playouts" (sequence of steps leading to game end) and from these samples he computes the expected (average) score for the given state. The artificial player then moves to the state with the best-sampled score.

This approach is very successful in the game Go; however, the most interesting benefit of this method is that the sampling, which substitutes here the game positions space searching, does not need any problem specific heuristics at all. Despite the fact that the playout is driven by random generator, the result samples can produce useful information.

2.2. Tree searching

The system described above aims to game positions valuating. If we look at the problem of artificial Go player more generally, the strategy of evaluating possible positions and taking the best-looking result is not very smart. As every greedy algorithm, it does not use any complex problem overview.

Solution of this problem was brought by an upgraded algorithm that is trying to benefit more from the promising states and searches for the game steps sequence as a whole. This algorithm searches the tree representing the game states space in these steps:

1. Go from the root node and by selecting always the best child find the most promising leaf node.
2. If this leaf node was already a few times attended, expand it (create child nodes).
3. Do some playouts ("simulations") from the current node.
4. Based on the playouts' results, update the quality information of the nodes in the tree that precede the playouts ("backpropagation").

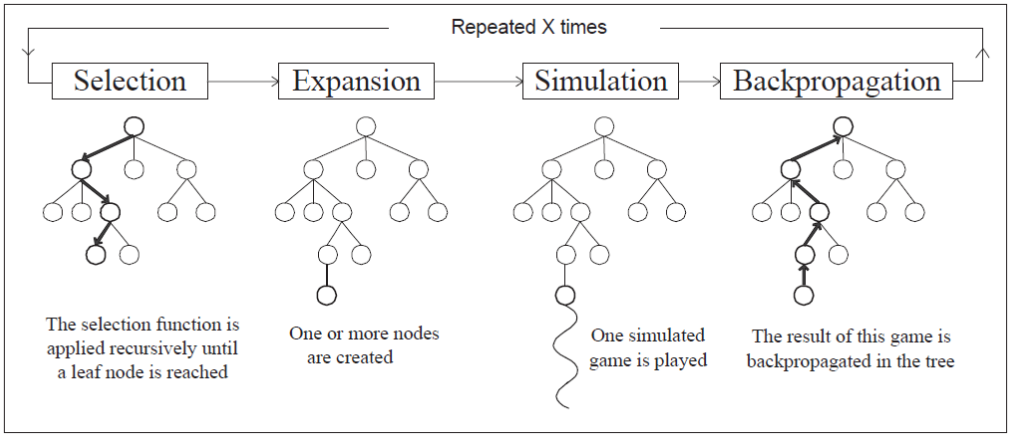


Figure 1 Monte Carlo Tree Search outline from [10]

As this cycle repeats, the algorithm is building an asymmetric tree, in which the most promising game strategy is growing deeper [10].

2.3. Upper confidence bound for trees

The MCTS algorithm as described above is a best-first searching algorithm. It tries to get maximum of the most promising branch it knows. This is exactly the core problem of many AI disciplines. The question is how to exploit the best-known option

if we do not have a complete model of the searched space. It is obvious that the exploitation should be preceded by an exploration mode, which would create as such corresponding model as possible. Since the searched space is too large to be stored in computer memory, it yields a question when to stop the exploration part and start the exploitation.

2.3.1. K-armed bandit problem

One of the problems dealing with the *exploration/exploitation* dilemma is the *K-armed bandit* problem, which is a plaything from the mathematical game theory [11]. *K-armed bandit* is a set of K slot machines, where every slot machine takes one coin for a play. After inserting the coin into the machine, the slot machine returns random reward from its inner distribution. The task is to maximize the reward sum by deciding which slot machine to play and in what order.

In 2002 was introduced a deterministic algorithm for dealing with the *K-armed bandit* called UCB (later UCB1) [12]. The UCB algorithm counts the total number of tries N, the number of tries at the particular arms n_k and the average gain of the particular arms μ_k. Once the UCB tries every arm ($\forall_{k \in \{1,\dots,K\}}\, n_k = 1$), it chooses the arm that maximizes the expression

$$\underset{k}{\operatorname{argmax}} \left(\mu_k + c \sqrt{\frac{\log N}{n_k}} \right)$$

where the c is exploration/exploitation constant originally set to $\sqrt{2}$ in the UCB algorithm.

This strategy provides very good balance between the exploration and exploitation modes. The maximized expression gives even bad options a chance when the good ones were tried many times. It was also proved, that the regret[1] of this method is a logarithmical function [12].

2.3.2. UCB for trees

In 2006, the UCB was applied as the solution of *exploration/exploitation* dilemma in MCTS [13]. The application is very straightforward. In every tree node (that is already represented in memory) is located a bandit and each arm represents one

[1] The average difference between the total gain of the optimal strategy and the total gain of our strategy.

possible move. During the MCTS cycle, the node's successor is chosen using the UCB strategy and the back propagated score is registered as the just obtained chosen arm gain.

What is more, the authors of this idea proved that if the nodes successors' evaluating is completely independent[1], this approach provides asymptotically the same results as the non-heuristic minimax algorithm [14].

[1] In the game Go the evaluating is, of course, not independent.

3. Model problem

In this thesis, we are trying to invent a new concept for solving the problems that are too complex or too large. Our approach should be applicable on every problem that is solvable by genetic algorithm. As you will see in the next chapter, all we need is to represent the individual from the population as a vector (chromosome) of finite dimension where every component (allele) is defined by a finite domain. We are also able to handle a mutual exclusion of values among the components, which we will demonstrate by our front-end problem specific implementation.

Although we want to bring a very general method for problem solution, we need an instance, where the principles can be shown. Therefore, we have decided to pick one particular sample problem. The explanation will be based on this example. The accompanying implementation or furthermore the research measuring and tests will be also proceeded on the instance problem. The sample problem is the Traveling salesman problem (TSP).

3.1. Traveling salesman problem

TSP is very well known problem, which is very simple to explain, but very hard to solve. Not only is TSP **NP-complete** [3], it also is **strongly NP-complete** [3]. What is more, TSP cannot be generally solved by any poly-time approximation algorithm with constant ratiometric error (unless **P** = **NP** of course) [3]. This makes TSP a very tough and challenging benchmark.

3.2. Genetic solutions of TSP

As we declared above, to use our method for solving TSP, we need a genetic representation for it. Since we are using the operator-oriented evolution, we need to decide the individual representation and prepare background for the environmental selection.

3.2.1. Fitness and environmental selection

The environmental selection in most of the EA[1] implementations is based on *fitness function*. The *fitness function* is an indicator which tells us, how good the particular individual actually is (in the problem solution meaning). In the TSP, there

[1] Evolutionary algorithm

is a very clear fitness indicator: weight of the Hamiltonian cycle represented by the actual individual. When implementing the environmental selection, we should not forget that the lower value is the better one. This kind of fitness function produces nominal numeric outputs. Nevertheless, we should go with the fact, that the outputs are mutually comparable, because we use the simple *Tournament selection* [5].

3.2.2. Representation of an individual

The individual representation needs an arbitrary decision to be made. There are a lot of various papers dealing with genetic TSP solution, which are using various representation:

- Steps representation is the most straightforward. The chromosome is simply a vector containing N numbers, which represents the cities exactly as the follow in the traveling salesman journey. Hence, the domain of every allele is the set of the cities (numbers $1 \dots N$). However, the numbers must be unique – a permutation.
- Edge/adjacency representation [15] also uses permutations of length N, but the meaning is different. The number i at the position j means that there is the edge $< j, i >$ used in the result Hamiltonian cycle. It is important to work carefully with this representation because not every permutation represents a Hamiltonian cycle. After selecting edges specified by the chromosome, there could occur several separated cycles in the graph.
- Buffer/ordinal representation [15] is the trickiest one. It keeps the cities ordered in imaginary buffer. The chromosome is also a long N and it also represents the cities as they follow in the result journey. The first number in the chromosome points into the buffer and selects the first city of the journey. The city is then removed from the buffer. The remaining cities are shifted in order to fill the empty gap. The buffer actually works as a "random access stack". The second number from the chromosome again points into the buffer and this repeats until there is at least one city in the buffer. As a consequence, the domain of the ith allele are the numbers $1 \dots (N - i + 1)$. Furthermore, every set of number satisfying the domain conditions are a valid representation of Hamiltonian cycle.

Even though all the representations have their pros and cons for using them in genetic algorithms, we have chosen the edge representation. The edge representation

is the best analogy to the nature genetics. That is because every allele has its own meaning (selects the edge that will be used for continuing from the corresponding vertex). On the contrary, steps or buffer representations does not provide any particular meaning to the allele until the whole chromosome is known. Hence, these representations would necessitate very large knowledge of the context for deciding the one allele. This could require creating very complex combinations, which could not be very friendly for genetic algorithm or MCTS.

4. MCTS operators for genetic algorithms

The aim of this paper is to create an approach that is combining two algorithms from different AI[1] fields of computer science. To do that we have decided to use the operator-oriented EA[2] design as a framework. The entire logic inspired by MCTS will be always encapsulated in several operators, which are modules for usage in the framework. Therefore, our operators can be combined with each other and, what is more important, with the traditional operators.

Our operators fall within the category of Memetic algorithms. We use the Lamarckian evolution idea and try to create very smart operators that do not use any problem-specific heuristic, but benefit from the MCTS research results.

We are going to present three levels of our idea. Each level will contain two implemented approaches to dealing with the chromosome inner system[3]:

1. *Direct* operator restricts its inner logic to keep the chromosome valid.
2. *Repaired* operator has free hand to manipulate the chromosome, as it wants to. Afterwards the chromosome is repaired to be valid.

4.1. UCB Selector

4.1.1. UCB Selector black box implementation

The *UCB selector* is a simple unit implementing exactly the *Upper Confidence Bound* selecting method. Each instance of this class is parametrized by the "exploration" constant and by the number of options that are available to select from. The *UCB selector* is used as a "black box" providing these functions:

- `Select()`, which simply evaluates the formula (the single components are explained in the chapter 2.3.1)

$$\underset{k}{argmax}\left(\mu_k + c\sqrt{\frac{\log N}{n_k}}\right)$$

and returns the maximal argument k, which represents one of the available options. The actual implementation also contains even more overloads of this method allowing us to specify a particular subset of available options to be

[1] Artificial intelligence

[2] Evolutionary algorithm

[3] Unique permutation representing one Hamiltonian cycle

selected from. In that case the $argmax$ formula chooses the k only from the given subset. In the situation that at least one of the available options was not selected yet ($n_k = 0$), this unselected option will be returned instead of evaluation of the formula. If there is more than one not-selected option, one of them is selected randomly.

- `RegisterGain(k, gain)` updates the internal records. It increments the stored number of attempts of the kth option n_k and updates its average gain μ_k. It also increments the global number of all attempts N. The registered gain is always a number from the interval $[0,1]$.

4.1.2. Alternative UCB selector initialization

The *UCB selector* implementation described above brings a mechanism, which automatically balances between exploration and exploitation and tries to identify the best option from the available set. However, this mechanism has a special beginning. Due to the not-selected options protection, there is performed a "select-each" loop across all the options at the beginning. That is because all the option counters are zeros at the beginning, and the *UCB selector* refuses to perform the UCB-selection until all the counters are at least one.

The *UCB selector* itself is the base and key building block. The *UCB selector* will be present in all the operators that we are going to present in this chapter. We are little bit afraid that the "select-each" loop, which is needed to be accomplished before the UCB-selecting begins, will confuse, break or slow down the mechanisms we are planning to implement. Therefore, we present an alternative way of the *UCB selector* implementation. The difference is only in the inner data structures initialization. In the original version, the option counters are initialized with zeros and the average option gains are initialized right at the moment when the first gain is registered for the particular option. While in the alternative version, the counters are initialized with ones and the average gains are all equally initialized with a constant g.

This alternative constant initialization omits the need of the "select-each" loop and still does not break the selector principle (the average gains still converge to their true values).

What of the two *UCB selection* initialization is better and what value for the constant g should be used we are going to find out experimentally in the chapter 5.

4.2. The used individual representation

Even though the TSP solution can be represented variously, all the below described UCB/MCTS inspired operators use the edge representation. This representation is a simple array of immediate successors of the vertices. Number i at the jth position in the array means that edge $< j, i >$ is used in the TSP solution.

4.3. Single allele operators

The level one operators are based on the idea of searching zero-deep trees rooted in the alleles (where the allele is one particular position in the successors array and the chromosome is the whole array). We present two different implementations of this principle. Both of them use the same base structure.

The operator is initialized with an array full of instances of the *UCB selector* units. The length of this array is exactly the same as the number of vertices in the TSP graph (number of towns). Hence, this array of *UCB selectors* has exactly the same length as the successors array coding the TSP solution. Each selector corresponds to the particular vertex in the graph and selects the vertex's successor in the Hamiltonian cycle. That is why all the selectors in the array are initialized with the same number of given options – the number of vertices in the TSP graph.

Not every combination of options returned by the selectors is a valid solution of TSP. Furthermore, the returned vector might not be a valid permutation at all, because values can repeat. Therefore, we present the two specific implementations based on this idea but providing valid solutions of TSP – Hamiltonian cycles. There will always be a *direct* and a *repaired* version of a chromosome filling mechanism. These two versions we are going to introduce in the following subchapters.

4.3.1. Direct single allele selecting operator

The first implementation of the level one operator is the *direct single allele selecting operator*. This operator works with the selector's *allowed options sets* to directly provide a vector which will be a valid solution itself.

During the solution creating process, the operator builds the Hamiltonian cycle step by step. It starts at a randomly chosen vertex. All vertices except this starting vertex are now in the *allowed options set*. We use the starting vertex's selector to tell us which one of the *allowed options* will be the selected succeeding vertex. The result successor is removed from the *allowed options set* and we repeat this procedure, now

with the result successor instead of the starting vertex. This process repeats until the *allowed options set* is empty. At this point, the only possible successor will be exactly the starting vertex chosen at the beginning.

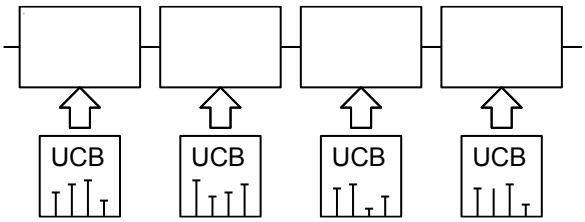


Figure 2 *Single allele selecting operator* schema. Every allele has its own corresponding *UCB selector* which determines the allele evaluation.

4.3.2. Repaired single allele selecting operator

The second implementation of the level one operator works a little bit differently. The TSP solution creating routine works in three steps:

1. Use the *UCB selectors* to generate an arbitrary array of vertices.
2. Repair this array to be a valid permutation (even with more cycles).
3. Decompose the permutation into separated cycles and join them into one Hamiltonian cycle.

While the first step is clear, there are more than one possible ways how to perform the second step. The task is to convert the list of numbers with repetition into list of unique numbers.

In the array, we identify the numbers (vertex successors) that are appearing more than once. These repeating successors we remove from the array. It is not necessary to remove all the occurrences of the particular repeating number. The number has to appear exactly once in the final array; therefore, we leave unchanged always one of the appearances of each repeating number. Of course, we do not know the optimal position where the number occurrence shall stay, thus we choose it randomly. The rest positions from where the repeating successors were taken out represent the vertices that have no selected successor at this time. We will call them the *empty predecessors*. And finally, there could be numbers which didn't appear in the former array at all. These are called the *currently unused successors*.

The *empty predecessors* and the *unused successors* should be now somehow connected together to provide a valid permutation – the array of successors where every vertex has a unique successor. We present two possible approaches how to do that:

a) Connect them randomly. For every *empty predecessor* will be the successor chosen randomly. Of course without repeating.
b) Use the *UCB selectors* again. Every *empty predecessor* still has its own corresponding selector. The *unused successors* become now the *allowed options set*. Going across the *empty predecessors* in random order, we let the selector to choose the successor from the available options and then exclude the chosen successor from the available options for the next iteration.

After choosing one or the other strategy, we have a valid permutation and the reparation step two is complete.

4.3.3. Updating the inner data structures

The level one operator would have no chance of success without continual updating its inner data structures – without learning. No matter if it is the *direct* or the *repaired* implementation, there is the array of *UCB selectors* inside. These selectors need to be updated to provide better results next time.

The gain that will be showed to the selectors (via `RegisterGain` function) is universal for all the *UCB selectors* stored in the array. That is because only a fully filled array of successors, representing Hamiltonian cycle, generates a particular solution of TSP. We cannot rate one used edge separately because it is not obvious whether using this edge leads to the optimal solution or not.

The gain of one particular chromosome, which is solution of TSP, is actually its fitness value. Since the *UCB selector* allows only number from interval $[0,1]$, the fitness value has to be transformed. The fitness value of the TSP solution is simply the weight of the result path. The lower fitness is better. The gain value is a different case: greater value is better.

Fortunately, the fitness value can be converted into a gain value relatively straightly using a linear transformation:

$$gain = 1 - \frac{fitness}{u}$$

where the u is simple the upper bound estimate for the fitness function. This estimate expresses weight of the worst possible solution of the TSP. We simply use sum of weights of the N heavier edges in the graph, where N is the number of vertices in the graph.

The redistribution of the gain value is quite intuitive. We go through the chromosome (successors array). For each allele, we register the gain for the option that

is used. After this process, every *UCB selector* has exactly one more registered attempt.

The truth is that in case of the *repaired* operator some selectors might register a different option than they have returned as a recommended selection initially. In other words, the *UCB selector* selects option k, but the gain is registered for option l, which has replaced the option k during the repairing process. Although this is not a standard usage of the UCB principle, it will not violate the selector's principle. The option k could not be used in the result solution. On the other hand, keeping in secret the gain of the finally chosen option l would not improve the operator at all. The registered gain helps to rate this particular option l and changes its probability of being selected next time.

4.4. Conditional operators

The level two operators – the *conditional operators* – introduce the first attempt to bring the idea of dependency between alleles in the chromosome. They are based on searching trees of constant depth.

The level one operators were based on the principle that every vertex has its own selector, which is trying to choose the best succeeding vertex in the result Hamiltonian cycle. Whereas the *conditional* idea tells us that the decision of the particular selector may be better if the selector considers the result selection of another (generally) selector. This brings some context into the selector's decision.

Naturally, you can imagine various approaches how to implement this idea. We present our two different implementations.

4.4.1. Direct conditional operator

Like the *direct* level one operator, the *direct conditional operator* produces a valid TSP solution literally directly. The main goal of the implementation is providing the dependency between the selectors and their choices. This operator, like the level one operators, goes step by step and builds the Hamiltonian cycle. In contrast with the level one operators, while deciding the successor for a particular vertex, the *conditional* operator considers even the predecessor of the current particular vertex.

The *direct conditional operator* contains an array full of *UCB selector* collections. The length of this array again equals the number of the vertices in the graph (N). Each collection in the array can contain up to N *UCB selectors* too. Every

record (every *UCB selector*) in the collection is marked by a vertex in order to provide the selector conditionally. Altogether, while deciding the next step of the Hamiltonian cycle, the vertex's successor will be selected by a selector that will be chosen from the collection corresponding to the current vertex and the key for choosing from the collection will be the predecessor of the current vertex.

Of course, the first decided vertex cannot be solved conditionally because its predecessor – the last selected vertex – is unknown at the moment. Therefore, the starting vertex is fixed and it has only one general corresponding selector instead of collection of selectors.

This operator uses the available options set in the exactly same way as the level one operator did. Thus, the result chromosome is array of vertex successors coding a valid TSP solution. The gain value of the result, computed from the fitness value equally to the level one operator, is registered again N times. For each vertex, only the used selector from the corresponding collection will register the gain value and it will be credited to the option that was actually selected – the chosen successor.

4.4.2. Repaired conditional operator

The *repaired conditional operator* contains similar inner data structures as the *direct* one. However, it works with the *conditional* selecting more abstractly and generally. This operator creates the dependencies between alleles in chromosome based on the position in chromosome. The allele is influenced by its neighbor one. To be concrete, every allele affects the allele on right side. Except the last one.

This *repaired* operator works in the same three steps as the level one *repaired* operator. In the first step, it builds the array of numbers – potential array of successors. It goes sequentially thru the array from the left to the right and every position is filled by the value chosen by the selector. At this phase, there is no restriction like "available option set" used – all options are available. The selector for the particular position is chosen from the corresponding collection and the key, used for the choice, is the value filled in the left neighbor allele.

The second – repairing – step can be proceed in the random way, which is exactly the same as in the level one implementation. However, we can take the advantage of the selectors again. After creating the list of *empty predecessors* and list of *unused successors*, we take the *empty predecessors* from the list in the increasing order (increasing order of chromosome array indices). It is obvious that the allele left

from the first *empty predecessor* is already filled. If it would not be, it would be the first *empty predecessor* itself. Hence, we can choose the right *UCB selector* from the corresponding collection for the allele that is the first *empty predecessor*. This selector is used to select a successor for the current *empty predecessor*, but only from the list of *unused successors*. The selected successor is removed from the *unused successors* list and we can continue repeating this procedure with the next *empty predecessor*. At the end of this loop we have got a valid permutation stored in the vertex successor array. The third step is again exactly the same as in the level one operator.

Figure 3 shows the schematic arrangement of the *repaired conditional operator*. Every allele has more corresponding *UCB selectors*. The previous allele chooses which one is used (green arrows). In the case of the *direct* variant, the green causality arrows would not always go to the right neighbor. They would respect the order determined by the constructed Hamiltonian cycle.

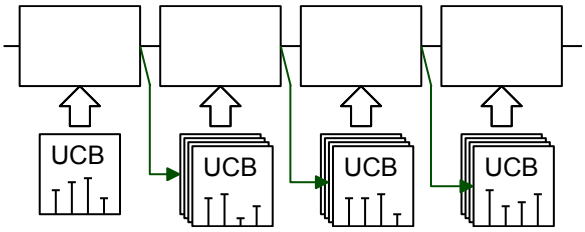


Figure 3 *Repaired conditional operator* schema.

4.5. Local trees searching operators

The idea of allele value selection that is based on the previously selected value can be even more generalized. While deciding the value for the current allele, we can consider more than one previously chosen value. The number of considered values should not be constant. Therefore, the mechanism can fluently grow while learning from the gain feedbacks. This brings us from the trees of constant depth to the trees of dynamic depth.

4.5.1. The actual trees

Since the depth of the trees has to be dynamic, we should start with a very small tree depth. The lowest depth of tree is, in general, one – only the root node and no edges at all. Using the trees with only a root node, we actually represent the same principle as the *single allele selecting operators* – the level one technique. Every allele has exactly one corresponding *UCB selector*, which is responsible for the values. The

only difference is that in the level three operators the *UCB selector* is encapsulated into a tree node, which is also a root node of the tree at this moment.

After the process of chromosome evaluation (all alleles have proper values), there is performed as well the phase when the *UCB selectors* – the trees – receive the actual gain of the particular chromosome. At this moment, the selected option counters are incremented. After this increment, it is time for the tree expansion, which is applied sequentially to all the trees. According to the used tree expansion policy, we choose the concrete tree nodes and the concrete options that should be expanded right now. We create new edges from the current node (containing the expanded option) into newly built nodes representing new *UCB selectors*, which are going to correspond with the next alleles. Moreover, these new *UCB selectors* are used only conditionally. They depend on the values selected for the previous alleles – by nodes above in the tree (nodes lying on the path from the root node into the current node).

The tree expansion described in the previous paragraph brings the analogy with the level two operators – the *conditional* ones. More than that, we can build the dependency chains as long as we want. Every dependency chain of *UCB selectors* is actually the path from the root node into a leaf node in one of our *local trees*.

4.5.2. Chromosome evaluation

In the previous text, we intentionally skipped the part when the actual chromosome is filled by concrete values. This procedure is not difficult at all if the inner data structures in the operator are clearly described. Inside the operator, there are the *local trees*, which everyone's root corresponds exactly to one allele in the chromosome. Some of the trees are only single root nodes without any additional subtrees, and the other trees are partially expanded but potentially asymmetric.

We start the chromosome evaluation process at the first allele. The corresponding *local tree* is asked not only for one value, but also for a sequence of values. The sequence of values is generated by going thru the tree from the root node to one of the leaf nodes. Every entered tree node contains an *UCB selector* inside. This selector is asked for the preferred (selected) option, which will be the actual next value in the result sequence. Then we look if the current node contains an expanded subtree for the selected option. If there is an edge expanding this option, then the process continues recursively by going along this edge. Otherwise, the selected option is the last value in the sequence and the query ends.

The generated sequence of values is the result of a decision process based on the UCT principle. Every value in the sequence was selected with regard to the preceding values. The first value from the sequence is filled into the allele that corresponds to the root node where the query started. The next value in the sequence we put into the next allele and analogically we fill the other succeeding alleles until we spend all the values in the sequence.

If there still remains any farther unfilled allele, we just take the tree corresponding to this allele and query it for another sequence of values. By repeating this procedure, we certainly fill the entire chromosome even whether all the *local trees* are only single root nodes, or whether there is a fully expanded tree, which would fill all the alleles by one query.

It is obvious that not all of the *local trees* are used for the chromosome evaluation. However, the alleles that were not filled by the corresponding root node were actually filled by a deeper and more specialized node, which takes more account of the context.

4.5.3. Trees expansion

Until this time, we brought to light how the trees are used and when the tree node should be expanded. Nevertheless, the expansion itself was not fully clarified yet.

First, we need to decide when a particular selectable option in a particular tree node should get its own edge leading to a new node. To determine this we use a simple mechanism of *maturity threshold*, which is commonly used in the Monte Carlo Tree Search. The *maturity threshold* is a constant number. We define that the option whose counter inside the *UCB selector* exceeds the *maturity threshold* shall be expanded.

When the decision of option expansion is made, a new tree node is created. The simplest way to create a new node is to instantiate a clear new *UCB selector* and put it inside the node. This fresh new node should learn all the information about the gains of various available options. However, we already have another node that already contains some learned knowledge about the options and their average gains for the current allele. This versed node is the root node of the *local tree* that corresponds to the current allele. This knowledge is also general – it is independent on the chromosome context. Therefore, we can reuse this knowledge in our new context-dependent node. To do that, we simply copy the inner data from the original *UCB selector* in the root node (of the tree corresponding to the current allele) and use it as

a base set in the new *UCB selector*. As a consequence, the newly created node already knows the gain distribution of its options – this general knowledge is taken from the original root node. Since this moment, the newly created tree node will be reshaping this general knowledge into a context specific, which might be different.

Copying the *UCB selector* inner data is not the only thing that is done while creating new node during the expansion process. The newly created node will as well keep a reference (pointer) to its origin node. This reference will allow us to perform a better expansion on this node in the future. When an option i in this node with reference is going to be expanded, the future child node will not be created as a clone of some root node. The origin for the copying will be the child node, of the referenced node, which is denoted by the same option i. Of course, this child node does not need to exist. In that case, the corresponding root node will be used as an origin instead.

The above described strategy, which tell us how to choose the origin node during the expansion, will let us to exploit the best information that is currently available for the current allele. Not only do we copy the already learned statistics for this allele, but this origin node also depends on the context which is actually a shorter version of our current context. By cloning this origin node, we actually prolong the context, which will produce a more specialized decision node.

4.5.4. Tree size

The previously described mechanism gives us a set of trees, which every one of them gradually grows. Theoretically, every tree can expand into a full size and symmetric form. The fully expanded tree describes and evaluates all the possible chromosome variations. This full expansion would of course spend exponential memory space for each tree. Since we are developing fast incomplete heuristic method, we have to avoid huge trees. To satisfy this requirement, we will use tree pruning and stricter expansion policy.

To perform an expansion of a particular option inside some tree node, we have needed this option's counter to exceed a given limit, which is called *maturity threshold*. This was the only condition for expansion. To keep reasonable tree sizes but still let the trees to expand the successful branches, we add one more condition that has to be satisfied. The candidate option's average gain (inside the *UCB Selector*) has to be greater or equal than a third quartile value of all the average gains stored in the

selector. In other words, for expansion, the option has to be tried and it has to show good results.

Despite the strict expansion policy, the trees can still grow larger than we want. For instance, an option that seemed to be good before is not actually good at the moment. Nevertheless, the option has been already expanded. The whole subtree under this option will be probably never used already. It should be cut off.

To know when to do the pruning, we prescribe a tree size limit. This simply will be the maximal number of the nodes in one *local tree*. Every time this limit is exceeded, one whole subtree is going to be cut off. The dropped subtree should be rooted in the worst rated node in the tree. We could seek the tree for the node with the lowest gain of all. However, this would take very long time and the whole tree should be searched. Instead of the systematic searching, we rather use an incomplete heuristic to quickly find a bad node.

Our implementation of the tree limit compliance is inspired by the SMA* algorithm [16], which deals with a very similar task – it searches a graph using a limited set of expanded vertices. Because the SMA* algorithm needs to add only one more node every iteration, it gets by with cutting only one tree leaf. It drops the leaf with the worst utility function value. Dropping the leaf means that the SMA* algorithm omits the paths that are begging with the prefix represented by the leaf. That is exactly what we want to do: drop off the node which represents an option sequence prefix of very poor quality.

The poor sequence prefix we seek greedily. First, we calculate the number of nodes that have to be cut off to fall below the size limit (ΔN). We start at the root node of the tree. From the root node, we go deep into the tree choosing always the worst expanded option. Using these steps, we locate a node that is larger than the needed size ΔN and its worst child subtree is not large enough. Finally, this located node is cut off.

The reader has certainly made an observation during the previous paragraph: there can be no subtree lying under the worst option in the root node that satisfies the ΔN size condition. In this case, the worst subtree of the root node is simply cut off and the searching process is repeated again.

4.5.5. Local trees operators implementation

We have described yet an abstract mechanism of *local trees*, which are filling a generic chromosome and are consuming its quality feedback information. What has

been not introduced until now is how to use this principle to produce solutions for our prototype case – the TSP. As well as in the level one and level two operators, we introduce two different approaches: the *repaired* operator and the *direct* one.

The main part that was not explained in the *local trees* mechanism is the alleles ordering. Nevertheless, the actual ordering was already explicitly used by expressions like 'the next allele' or 'the first allele'. In other words, the actual *local trees operator* needs some linear ordering of the chromosome alleles. When we select an option for a particular allele and continue in the tree to the next node annotated by this option, the linear ordering is telling us to which allele the succeeding node corresponds.

4.5.6. Repaired local trees operator

The *repaired* operator implements the *local trees* principle quite straightforwardly. It uses the allele ordering exactly as they are located in the chromosome. The first allele is the first item in the chromosome array and the next allele is always the left neighbor one. This approach is very analogous to the *conditional repaired operator*.

The rest implementation of the *repaired local trees operator* is similar to the other *repaired* operators from previous chapters. It very freely fills the chromosome in the first step and then the result is repaired into a valid TSP solution. Like in the previous *repaired* operators, the *local trees* can be used for the repairs as well. The *UCB selector* can select from restricted *allowed options set* even when it is inside some node in a tree.

Figure 4 visualizes the schematic arrangement of the *repaired local trees operator*. The green arrows are the inner tree edges, which determine the concrete context-specific *UCB selector* for the particular allele. The blue arrows show the origin node used for the new node creation during the expansion.

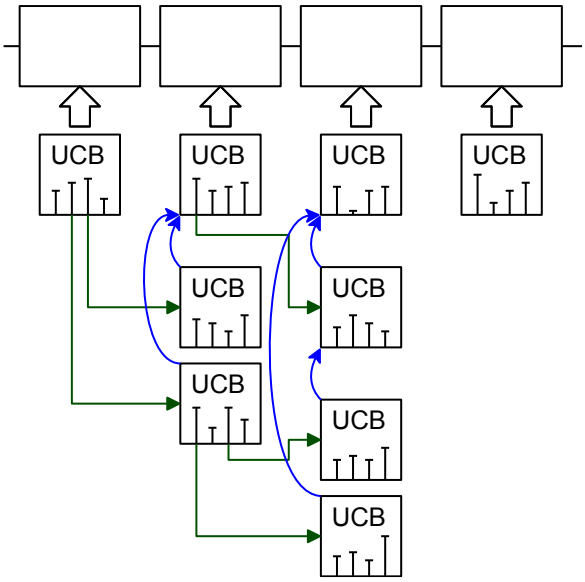


Figure 4 *Repaired local trees operator* schema.

4.5.7. Direct local trees operator

The *direct* operator extends the *direct conditional operator* from the level two implementation. The actual allele ordering depends on the values filled in the chromosome – on the selected options. The first allele is again the most left allele in the chromosome array. However, the next allele is determined by the option selected for the current allele. The next allele will be the allele that is representing the vertex succeeding the vertex that is represented by the current allele. As a consequence, the chromosome is filled by values in the exact order as the result Hamiltonian cycle goes thru the TSP graph. This is, again, very analogous to the other *direct* operators implementations.

The main difference between the *repaired local trees operator* and this *direct* one is that in the *repaired* operator all the child nodes of some node do correspond to the same allele. On the contrary, in the *direct* operator each child node corresponds to another allele.

4.6. Summary

In this chapter, we have introduced six variants of the MCTS-inspired operators for solving the TSP in Evolutionary algorithm. The core element of all of the operators is a gadget called *UCB selector*. Every *UCB selector* corresponds with one and only one particular allele in the chromosome. On the other hand, one allele can have more than one corresponding *UCB selectors*. When it is asked to, the *UCB selector* chooses the right option for the particular allele. The six variants of the operators we divide

into three levels by the complexity of usage of the *UCB selectors*. Each level then contains two approaches for solving the inner chromosome constraints[1]. To clarify the terminology, the following table brings the overview of our operators:

Level	**Constrains solving variant**	**Name**	**Maximal number of UCB selectors**	**Minimal number of UCB selectors**	**Max. context length**
One	Direct	Single Allele Selecting	N	N	0
	Repaired				
Two	Direct	Conditional	$N^2 - N + 1$	$N^2 - N + 1$	1
	Repaired				
Three	Direct	Local Trees	$N \times t$	N	$t - 1$
	Repaired				

Table 1 MCTS-inspired operators summary

N... input graph size

t.... the chosen tree size limit

All the *repaired* operators can be also parametrized by the chosen repairing strategy. The variants are: the *UCB-repaired* and the *randomly repaired* (both explained in chapter 4.3.2).

[1] The chromosome has to represent only valid Hamiltonian cycle.

5. The tests and measurements

Our MCTS operators, as we have introduced them, do have plenty of various parameters and settings. In this chapter, we are going to compare behavior and qualitative results according to the different parametrizations. Than we are going to measure the operators' performance in producing the TSP solutions.

5.1. Methodology

5.1.1. The algorithm run and results recording

Each experiment will be executed as an evolutionary algorithm. There will always be the population of individuals (chromosomes), which will be within every generation affected by the used operators. At the end of the generation there can be performed some type of an environmental selection. For the operator characteristic measuring we want to see only the development made by the operator, so if the selection is not mentioned in the measuring specification, there is no selection performed. The particular setting of the evolutionary algorithm will differ in various experiments. During the evolutionary algorithm run, all the fitness values of each individual will be recorded.

The recorded fitness values will be reported in a various graphical charts. The used types of charts, which visualize one evolutionary algorithm run, are going to be these:

- **Box plot**. The box plot chart shows the statistical information about every generation. Every box plot record shows the maximum and the minimum, the first and the third quartile and the mean and median values.
- **Lines of the best-found solution**. The continuous line expresses the progress of the best solution found yet.
- **Lines of the best in generation**. This chart shows the best fitness value in each generation record.

All these three types visualize the fitness value (vertical axis) depending on the generation number (horizontal axis).

For the purpose of comparing the performance of different solving methods are going to be executed multiple (at least five) runs of the evolutionary algorithm, whose records are going to be averaged and visualized as:

- **Averaged best histogram**. This chart shows all the fitness values of the best individuals in every generation sorted from the worst fitness to the best fitness. The vertical axis again shows the fitness value. The horizontal axis shows the number of averaged samples, which is equal to the generations count, but the order may not be the same.

Each chart figure is going to have its title where the particular type of the chart is denoted.

In the case that there is a large set of measured records and the chart space for one data point representing one generation would be too small, only subset of all the data points is shown. This is made by simple uniform sampling. For instance, every tenth generation is drawn. This data sampling will be maintained mainly for the box plot charts because they need more space for figuring one generation.

From each measuring, only one or a few charts will appear in this text. The rest of the experiments' outputs can be found on the attached CD.

5.1.2. Input data

For the purpose of testing we have developed several TSP instances generators. Each of the generator is able to create TSP input (a complete graph) of the desired size N and other parameters. The generators produce these graph categories:

- **Random graph** consists of N vertices and the distance between every two vertices is chosen randomly from the desired interval.
- **Triangle unequal graph** also contain N vertices. The vertices are put into the 2D space and the distance between them are computed by Euclidean metric. Therefore, theses graphs satisfy the triangle inequality condition.
- **Grid graph** is generated from points in 2D space too. These points are situated on a regular square grid of desired width and height ($w \times h = N$). The distances are also calculated using Euclidean metric.

All the generated graphs that will be used in the following experiments will be saved and attached to this thesis on the CD. Thanks to this, our measuring will be potentially repeatable.

The random graph is the most general input of TSP. Hence, we are going to use this type of graph for the operators' behavior and parametrization tests. The rest types of input we are going to use in the performance and verification measurements.

In the first tests (behavior and parametrization), we are going to intentionally use very large number of generations (10000) in every run. The reason is that we do not know the best operators' settings and we want to observe their characteristics even if the convergence will appear very late or not at all.

The graph size N will be used very frequently in measuring specifications or the following text. It is necessary to remember that this parameter determines a lot of mechanisms in our operators. The N is: the graph vertices count, the individual's chromosome length, the potential size of one allele domain, the number of *UCB selectors* in the *single allele operator*, the number of *local trees* in the *local trees operator* or the number of potential options in any *UCB selector*.

5.2. Basic UCB principle settings

The very first thing, we have decided to measure, is the basic settings of the actual UCB principle. The UCB principle occurs in every type and every level of our operators. To get the best performance from our operators, we have to tune the basic shared parameters first.

The impact of the basic shared parameters will be measured on the simplest level of our operators – the level one, the *single allele selecting operators* represented in chapter 4.3.

5.2.1. Repairing strategy

At all the levels of the implementation that we have introduced always contain two different approaches for solving the TSP. They are the *direct* and the *repaired* operators. Let us focus on the *repaired* approach at this moment. The *repaired* operator fills the chromosome in two phases. It absolutely freely uses the UCB principle and then it performs some repairs to provide a valid TSP solution. We already proposed two types of repairing strategies: the random one and the UCB one, which again uses the technique of the particular level.

We will compare these two approaches on a real TSP instance. The better repairing strategy we will then use in the following tests.

Measuring 1 Repairing strategies

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constant: $\sqrt{2}$
> Operators: Repaired single allele selecting
> Repairing strategies: UCB, Random

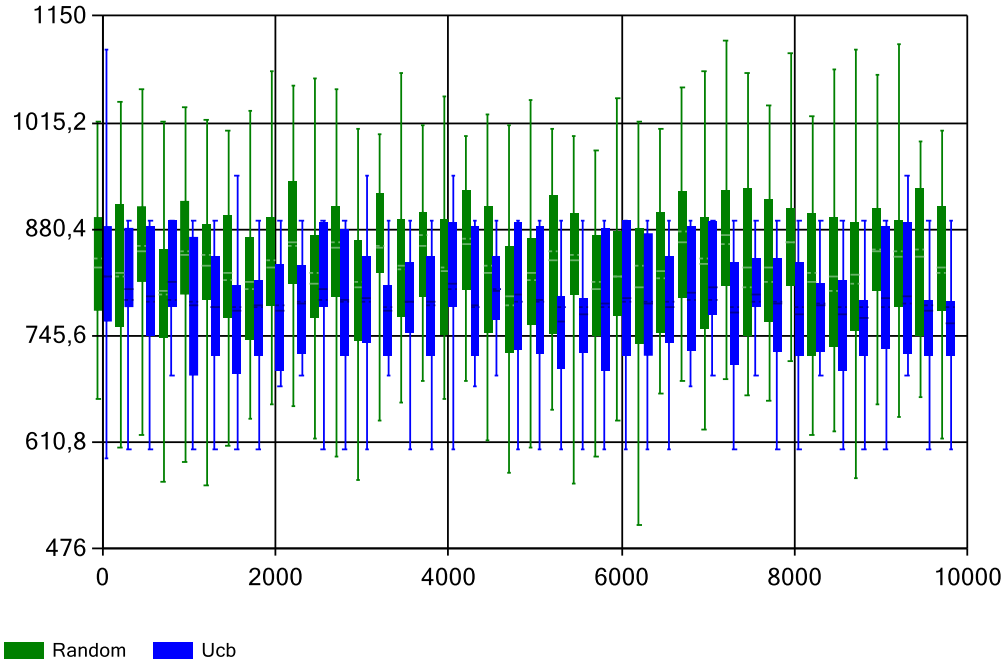


Figure 5 Measuring 1 Repairing strategies – Box plot

The results of the repairing strategies test show that the random repairing provides much wider variance of the fitness values in one generation. On the other hand, the UCB repairing keeps out of the actually bad values and generally produces less variance.

This result is not surprising at all. The fact that the random postprocessing would generate the bad solutions as well as the good solutions is evident. However, it seems like the UCB repairing is too conservative in this configuration. It finally is not able to create a better solution than the random repairing. In Figure 5 there is observable that the UCB repairing sticks at the same best solution which it has found, and does not explore the solution space enough to find a better chromosome. As a consequence, the random approach did meet better solutions than the UCB one. That is also obvious from the following figure showing the best-found solutions in the same experiment run as is shown in the Figure 5.

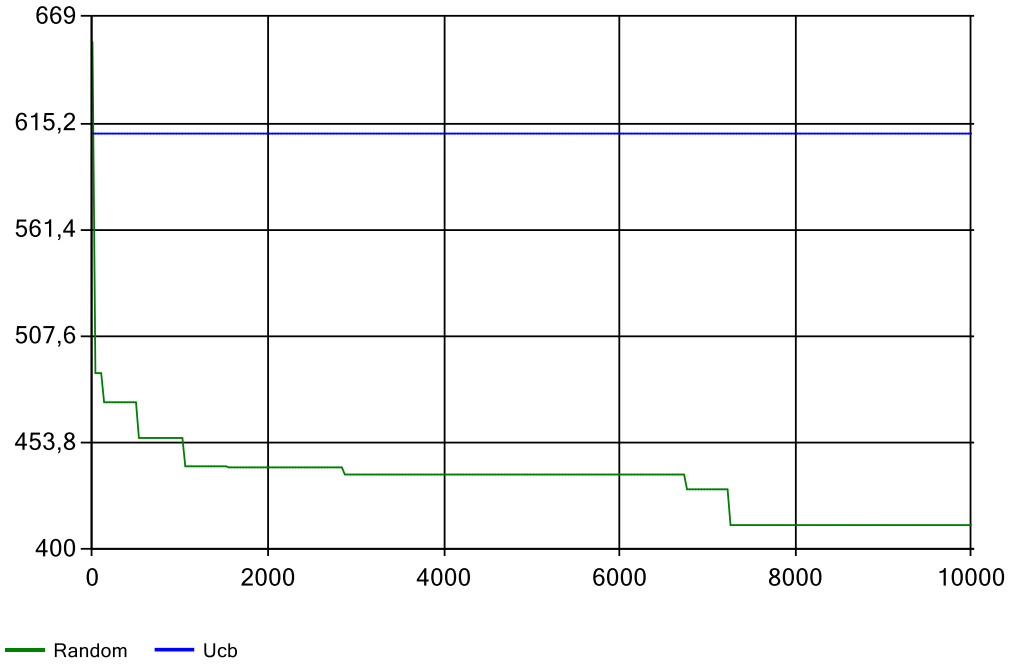


Figure 6 Measuring 1 Repairing strategies – The best found solution

Using this base configuration, the randomly *repaired* approach seems to be generally better. This will be demonstrated by the histogram chart made from multiple experiments of the same configuration.

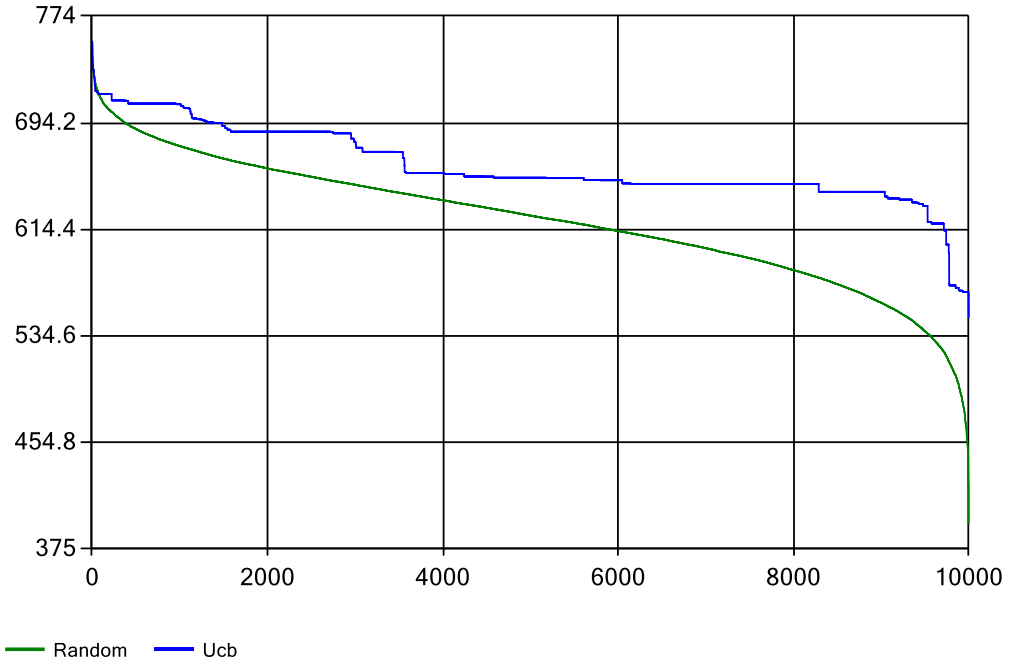


Figure 7 Measuring 1 Repairing strategies – Avg. histogram of the best in generation

5.2.2. Exploration constant

The core of all the operators we have introduced is the UCB principle. The $argmax$ expression always selects the option with the best optimistic perspective. Inside the $argmax$formula, there is a parameter c, which determines the ratio between exploration and exploitation (higher values cause more exploration). Let us see how this parameter inside the selecting expression impacts the seeking for solution.

We are going to do this experiment using the level one operators. However, there could occur a misinformation caused by the difference between the *direct* and the *repaired* operators. To prevent this side effect, we will test the various values of the exploration constant on the *direct* operators only. Then we will test the impact of cooperation of the selected exploration constants with the two types of the operators.

Measuring 2 Exploration constant in direct operators

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constants: 0.10, 0.50, 1.00, 1.41, 2.00, 4.00, 10.00, 100.00
> Operator: Direct single allele selecting

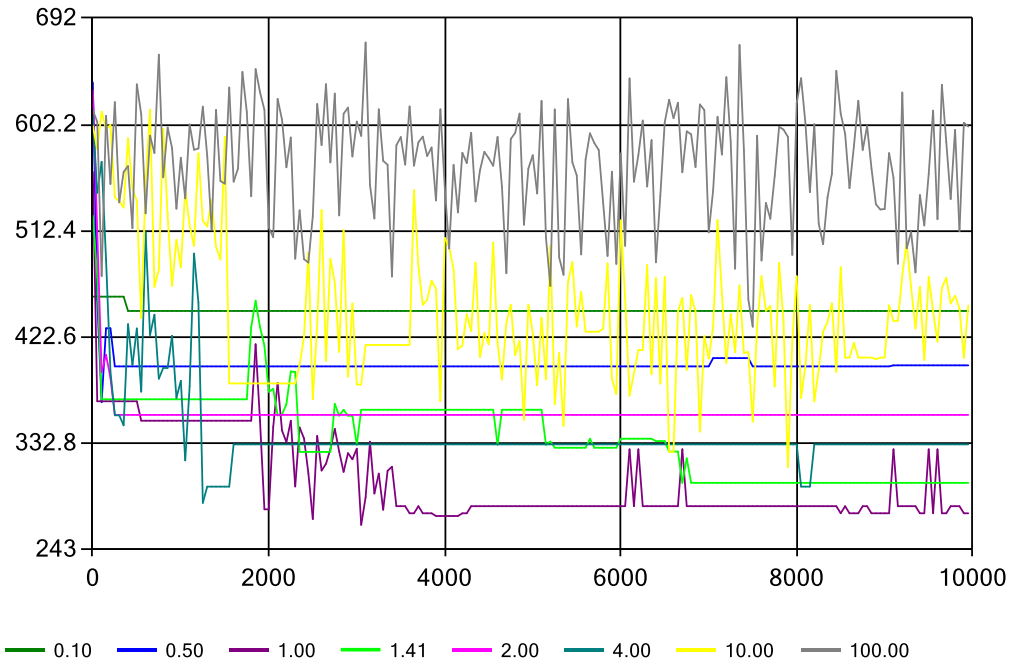


Figure 8 Measuring 2 Exploration constant in *direct* operators – The best in generation

The Figure 8 clearly shows how the *UCB selectors* react to the various exploration constant. The values that are lower than one generate very constant development of the best chromosome in the generation. It is observable that they stick to the good values that they saw at the beginning of the evaluation. The exploration mode is totally suppressed.

On the other hand, the greater values (10 and 100 in our experiment) obviously omit the exploitation part. Despite the fact that the exploration sometimes hits very promising chromosomes, the greater values show no signs of systematic convergence at all.

To the winning position aspire the values chosen from the interval $[1.00, 4.00]$. In these particular measuring results shown in Figure 8, there is one line, whose progression shows all the excellent attributes. It is the exploration constant value $\sqrt{2}$. This setting of the exploration constant does not have a constant invariant development of the best chromosome. What is more, the $\sqrt{2}$ line does not have a wavering progression, but it shows the slowly gradual convergence to the better values. These attributes show that not only does the $\sqrt{2}$ exploration constant provide a mechanism that can explore for better solutions, but it is also able to exploit the chromosomes giving good gain values.

One more thing that should be pointed out about the *Measuring 2 Exploration constant in direct operators* is that when we have repeated the same measuring several times, not always was the best value the $\sqrt{2}$. Nevertheless, the division into the three groups of the only exploiting, the only exploring and the balanced, was always the same.

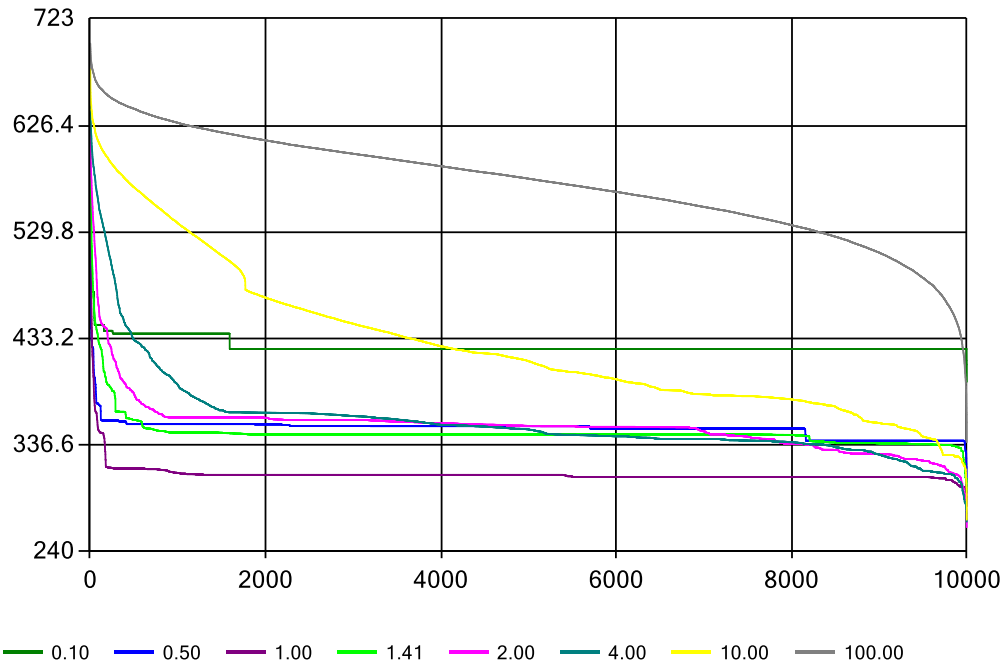


Figure 9 Measuring 2 Exploration constant in *direct* operators – Avg. histogram of the best in generation

The astute reader will notice that in the *direct* operators in the *Measuring 2 Exploration constant in direct operators* hit a better fitness values than the *repaired* operators in the *Measuring 1 Repairing strategies*. To find out what impact does the exploration constant have on the *repaired* operators and if it differs from the *direct* operator, we have chosen the most interesting exploration constant values a tested them against the both types of operators.

Measuring 3 Exploration constant

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constants: 1.00, 1.41, 2.00, 4.00
> Operator: Direct single allele selecting, Randomly repaired single allele selecting

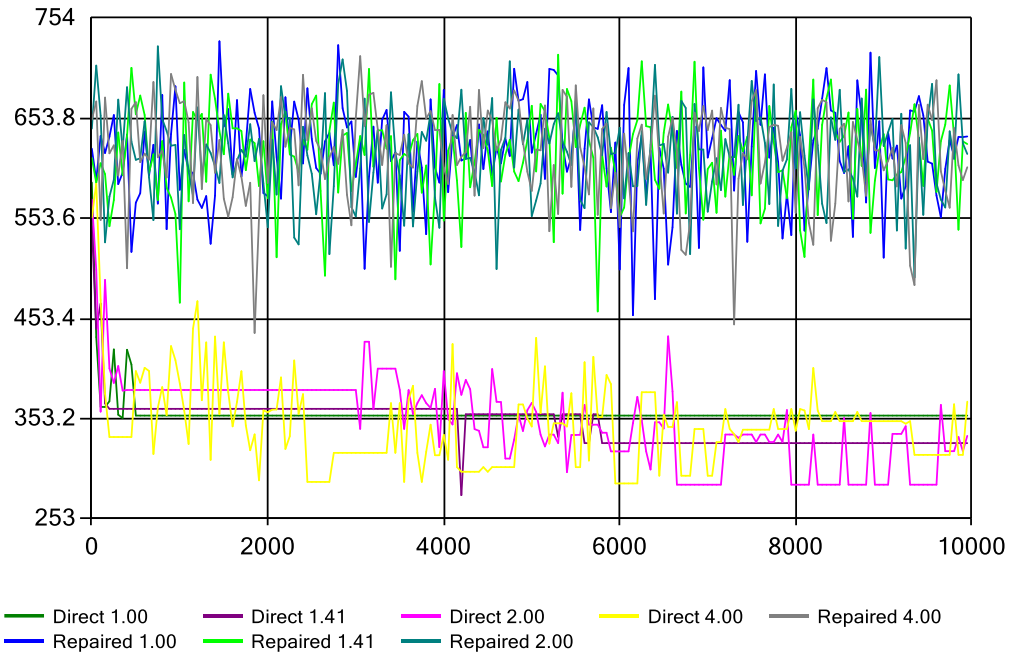


Figure 10 Measuring 3 Exploration constant – The best in generation

This comparison, as shows the figure above, does not change the hypothesis that the *direct* operators are more powerful. None of the tested exploration constants did bring the *repaired* operator into the competitive results. What is more interesting is that unlike the *direct* operators, there is no obvious impact of the different exploration constant on the fitness value progression. All the *repaired* lines waver in the similar variance and all the *repaired* histograms are completely the same.

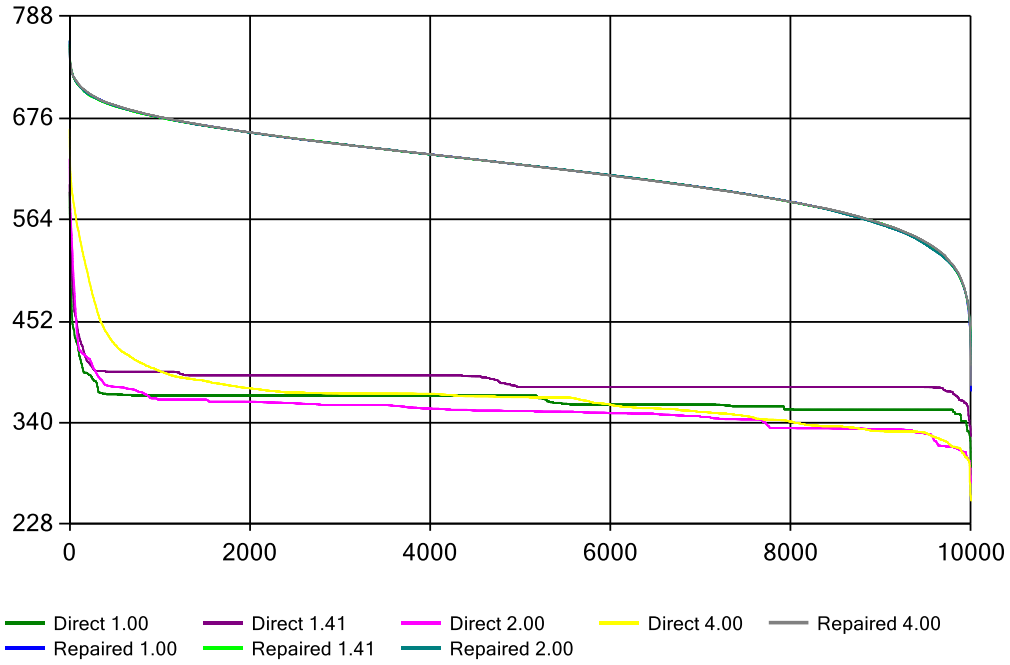


Figure 11 Measuring 3 Exploration constant – Avg. histogram of the best in generation

The unstable characteristic of the *repaired* operators is probably caused by the repairing concept itself. The *UCB selectors* maybe do not have enough opportunity to exploit the learned gains. The *direct* operators have a better perspective about the built context inside the evaluated chromosome. The context in the *direct* operators is represented by the available options set. Whereas in the *repaired* operators, this form of context is used only in the repairing phase and only at a few *UCB selectors*. Hence the context is not distributed as wisely as in the *direct* operators.

5.2.3. UCB selector initialization

The original idea how to implement the *UCB selector* dealing with the not tried options (options whose counters are zeros), was that this not tried options are selected preferentially in a random order. There we were a little bit afraid of what effect will bring this select-each loop across all the options (described in chapter 4.1.2). Therefore, we introduced an alternative solution for the not tried options. This alternative is the constant gain initialization for the *UCB selector*: the *UCB selector* starts with all the option counters at number one (instead of zero) and the average gains at the specified constant g.

The actual effect of the different initializing methods and the various constants g we will find out in the two experiments below. Due to the fact that the average gain is part of the $argmax$ decision formula which is influenced by the exploration

constant, we decided to do this experiments with two of the good-performing exploration constants.

As the chart containing 16 different configurations would be a little bit chaotic, we split this measuring into two separated experiments – we divide the *direct* and the *repaired* operators.

Measuring 4 UCB selector initializing in direct operators

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constants: 1.00, 4.00
> UCB initialization: all opts at first, 0.1, 0.5, 0.9
> Operator: Direct single allele selecting

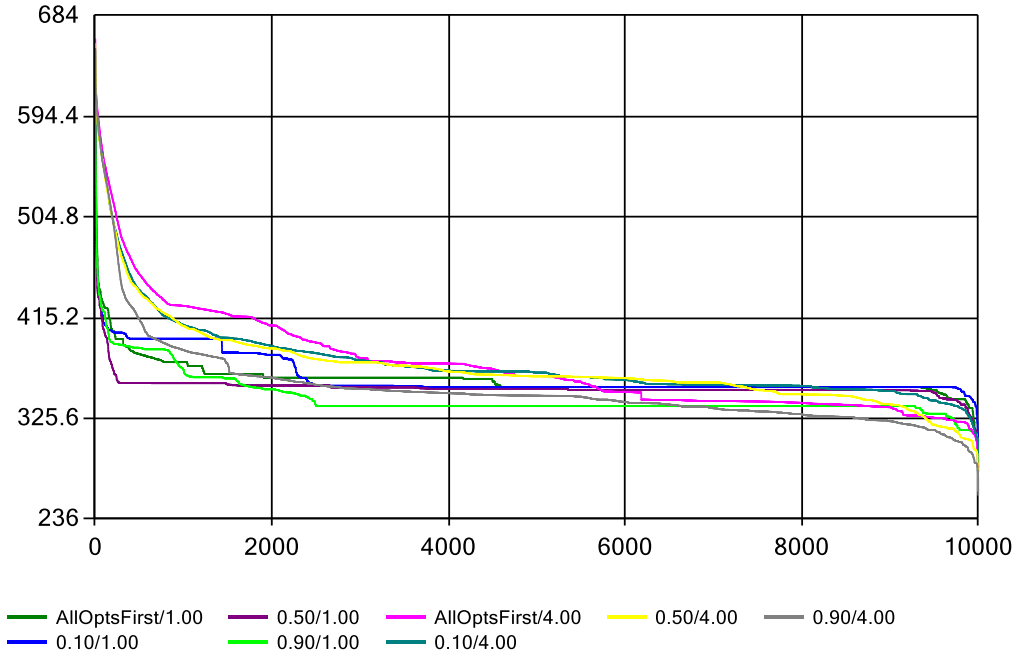


Figure 12 Measuring 4 *UCB selector* initializing in *direct* operators – Avg. histogram of the best in generation

The *direct* operators have shown that the chosen *UCB selector* initialization strategy does not matter a lot. There is no obvious winning or losing strategy in the Figure 12. We also cannot declare that the strategy "All options at first" is significantly better or worse than the constant initialization.

Altogether, the *UCB selector* initialization is not as important as it could seem. At least in the *direct* operators. Let us see, if there is any difference in the *repaired* operators.

Measuring 5 UCB selector initializing in randomly repaired operators

TSP instance: Random 15
Number of generations: 10000
Population size: 40
UCB exploration constants: 1.00, 4.00
UCB initialization: all opts at first, 0.1, 0.5, 0.9
Operator: Randomly repaired single allele selecting

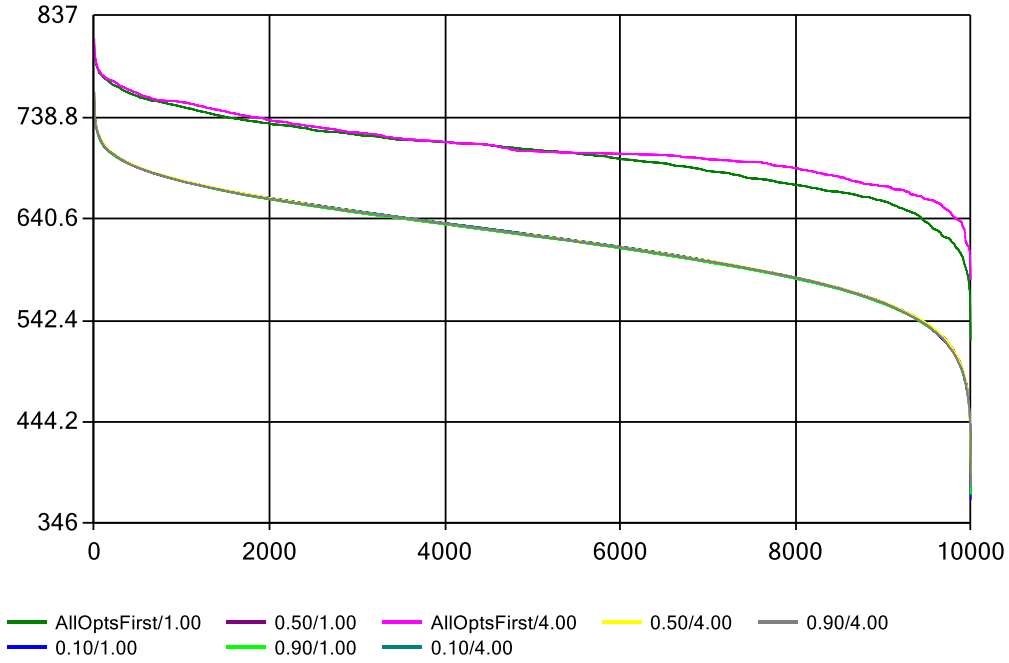


Figure 13 Measuring 5 *UCB selector* initializing in randomly repaired operators – Avg. histogram of the best in generation

If we have declared that in the *direct* operators the UCB initialization constant g makes no important effect, here in the *repaired* operators the constant g makes no effect at all. There is absolutely no difference between the curves showing the development of the various constants g. On the contrary, there is an obvious difference between the constant initialization and the *all option at first*, denoted AOAF, strategy. The constant approach clearly dominates the AOAF.

The reason why the AOAF initialization strategy worsens only the *repaired* operators could be again in the *repaired* principle. In the random repair procedure, there is no mechanism for satisfying the "all options must be tried" requirement. The random ending of the solution building process can cause that even the *UCB selectors* already have plenty of information about most of the options, they still have to select from the rest of the options that was not tried yet. However, this enforced selection is broken in the repairing step. As a consequence, the cycle of wrong selections can repeat infinitely.

To confirm the hypothesis from the previous paragraph, we run the same experiment on the UCB repaired operators.

Measuring 6 UCB selector initializing in UCB-repaired operators

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constants: 1.00, 4.00
> UCB initialization: all opts at first, 0.1, 0.5, 0.9
> Operator: UCB-repaired single allele selecting

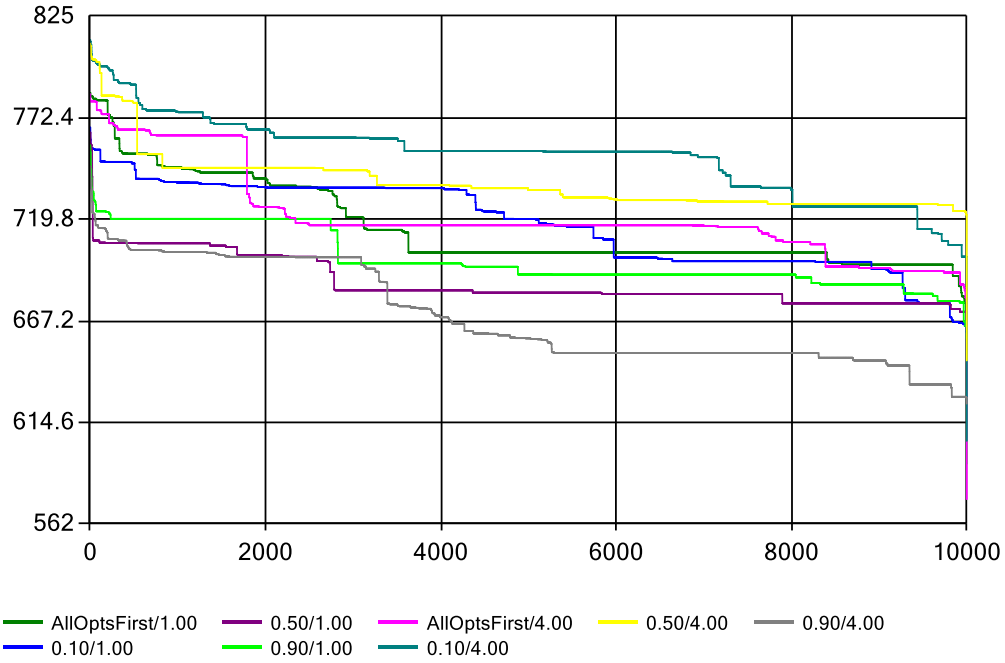


Figure 14 Measuring 6 *UCB selector* initializing in UCB-repaired operators – Avg. histogram of the best in generation

Observation made in Figure 14 goes with our hypothesis. Even though the *UCB-repaired* operators are generally worse, the AOAF initialization does not have the same impact on them as it has on the *randomly repaired operators*. The reason why the *UCB-repaired* operators can deal with the AOAF initialization is because the a*ll options at first* principle is applied even at the repairing phase (the repairing is done by the *UCB selectors*).

The conclusion of the experiments with the *UCB selector* initialization is to use an arbitrary constant g. In the very first experiments in which the initialization strategy was not even mentioned, there was used the constant initialization with $g = 0.50$. Hence, the experiments do not need to be repeated with a better *UCB selector* initialization.

5.3. Basic settings in high level operators

In the previous chapter 5.2, there were done several experiments about various parameters, configurations and settings of the methods that we are introducing in this paper. Some of the results were predictable and some were not. In this subchapter, we are going to see whether the results that were observed at the level one operators, will differ or be the same in the case of the level two and level three operators from chapters 4.4 and 4.5.

The higher level operators also bring new parameters which should impact their behavior. These parameters we are going to observe in this subchapter as well.

5.3.1. Repairing and selector initialization strategies

The tests made on the *repaired single allele selecting operators* brought to light some interesting observations:

- The random repairing is more powerful than the systematic UCB repairing strategy.
- The chosen selector initialization strategy is more or less irrelevant in the UCB repairing, but it has changed the behavior of the *randomly repaired operators*.

These observations and the generally worse results of the *repaired* operators we have explained by the absence of the context information for the particular *UCB selector*. Nevertheless, the higher level operators do use more of the context information in evaluating the chromosome. Let us see how the described behavior will change in the higher level *repaired* operators.

Measuring 7 Repairing and UCB initializing strategies in higher level operators

TSP instance: Random 15
Number of generations: 10000
Population size: 40
UCB exploration constant: 2.00
UCB initialization: all opts at first, 0.5
Operators: Randomly repaired conditional and local trees, UCB-repaired conditional and local trees
Local trees maturity threshold: 5
Local trees size limit: N^5

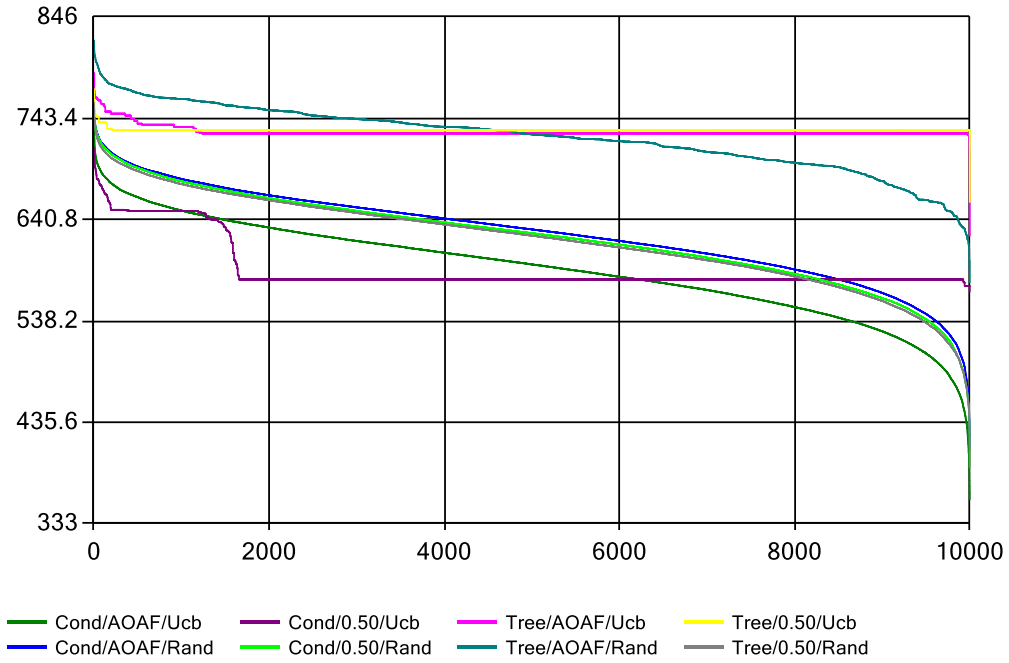


Figure 15 Measuring 7 Repairing and UCB initializing strategies in higher level operators – Avg. histogram of the best in generation

The **Measuring 7 Repairing and UCB initializing strategies in higher level operators** brings a very contradictory result. If the reader looks only at the *conditional operators*, it is obvious that the context information, which is provided by the *conditional* mechanism, helps and the AOAF initialization does not corrupt the results of the *random repairing*. On the contrary, in the *local trees operators*, there is a downgrade of the *randomly repaired* version using the AOAF initialization. In other words, while the *randomly repaired* level three operators react the same way as the level ones, the level two operators do not.

The *local trees operators* follow the results of the *single allele operators* in the other views as well:

- The *UCB-repaired* versions show much worse convergence than the *randomly repaired*.

- The AOAF initialization strategy downgrades only the *randomly repaired* version.

In contrast, the *conditional operators* have a very unusual behavior:

- The AOAF initialization does not impact the *randomly repaired* version.
- The *UCB-repaired* version with AOAF initialization is the best in this entire experiment.

These extraordinary results can seem to be just a coincidence; however, they are not. We did repeat this measuring multiple times and the results as they were described were stable. There could be a plenty of explanations for the *conditional operators'* behavior. Maybe the constellation of context of length one plus the other parameters is the ideal setting of the *UCB-repaired* operator. On the other hand, maybe the experimented parameters are not advantageous for the *repaired local trees operator*. We do not have any logical explanation at this moment; nevertheless, we will try to figure out some reason by other experiments.

5.3.2. Exploration constant once more

Our first experiment on level two and level three operators has brought confusing results. The most worrying fact is that the *local trees operators* show less performance than the *conditional operators*, which are using a shorter context information.

We did some experiments aside and these have pointed out the problem. At the beginning of this experiments chapter, we have declared that the basic configuration of the base principles will be measured on the level one operators. Than we have decided, that the best-detected setting will be used in the higher level operators. The presumption that the common parameters for all the levels can be set equally is wrong. The most core parameter of all – the exploration constant – breaks it already.

Measuring 8 Exploration constant in Direct local trees operators

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constants: 0.00, 0.20, 0.40, 0.70, 1.00, 4.00, 10.00, 100.00
> UCB initialization: 0.5
> Operators: Direct local trees
> Local trees maturity threshold: 5
> Local trees size limit: N^5

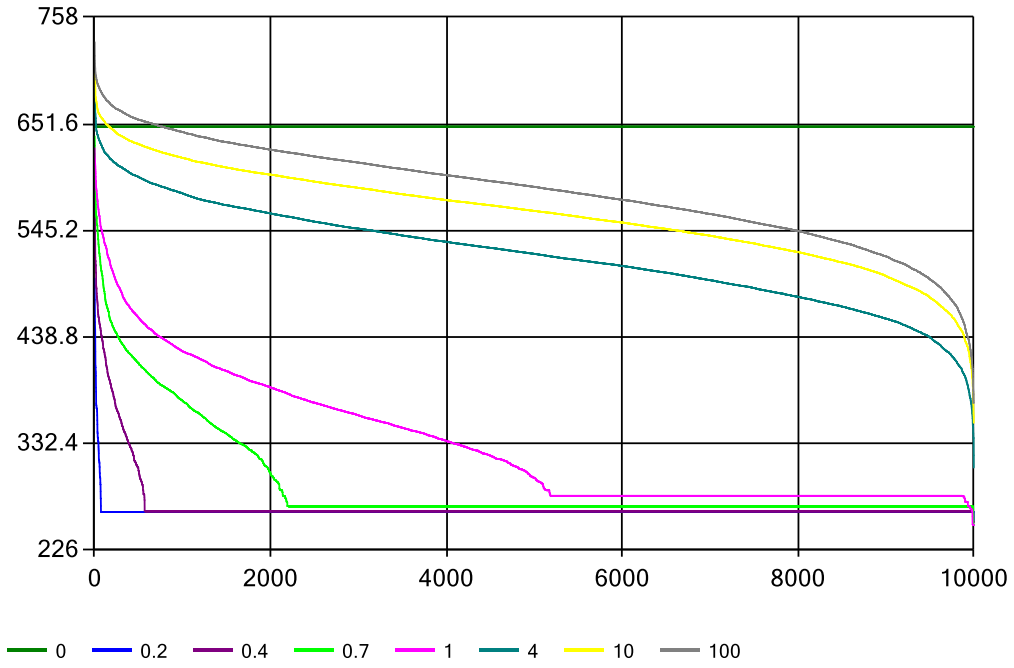


Figure 16 Measuring 8 Exploration constant in *Direct local trees operators* – Avg. histogram of the best in generation

It is obvious that in the case of the *local trees operators* the effect of the exploration constant is slightly shifted. While in the *single allele selecting operators* the optimal interval was declared as [1,4], the *local trees operators* work good with a little lower values. The reasonable exploration constant should be around the interval $[0.4, 1.0]$. To see how the exploration constant affects the particular exploration/exploitation development, let us see one concrete experiment run.

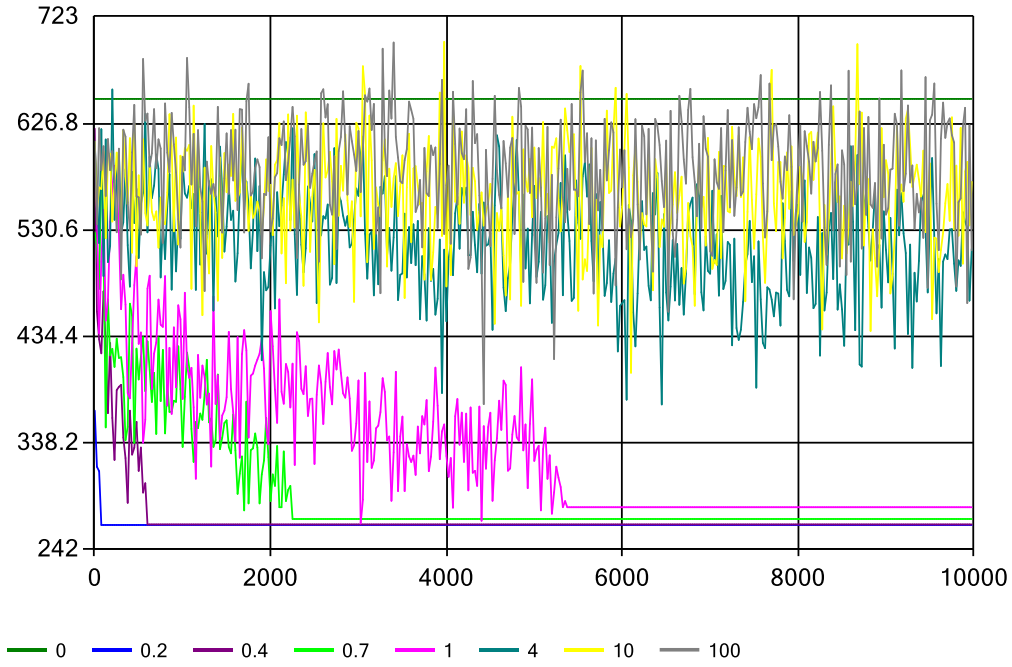


Figure 17 Measuring 8 Exploration constant in *Direct local trees operators* – The best found solution

As you can see, the exploration constant value one is the last that does provide a systematic convergence. The higher values seem to explore a lot, but do no exploitation at all. The user of our *direct local trees operator* should also take into account the threat of the premature convergence. Therefore, we would prefer to not use the exp. constant such low as 0.2. despite the best development in our experiment.

To be complete with the exploration constant, which turned out to be the key parameter, we should do the experiment with the *repaired* operator and with the level two operators.

Measuring 9 Exploration constant in Repaired local trees operators

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constants: 0.00, 0.20, 0.40, 0.70, 1.00, 4.00, 10.00, 100.00
> UCB initialization: 0.5
> Operators: Randomly repaired local trees, UCB-repaired local trees
> Local trees maturity threshold: 5
> Local trees size limit: N^5

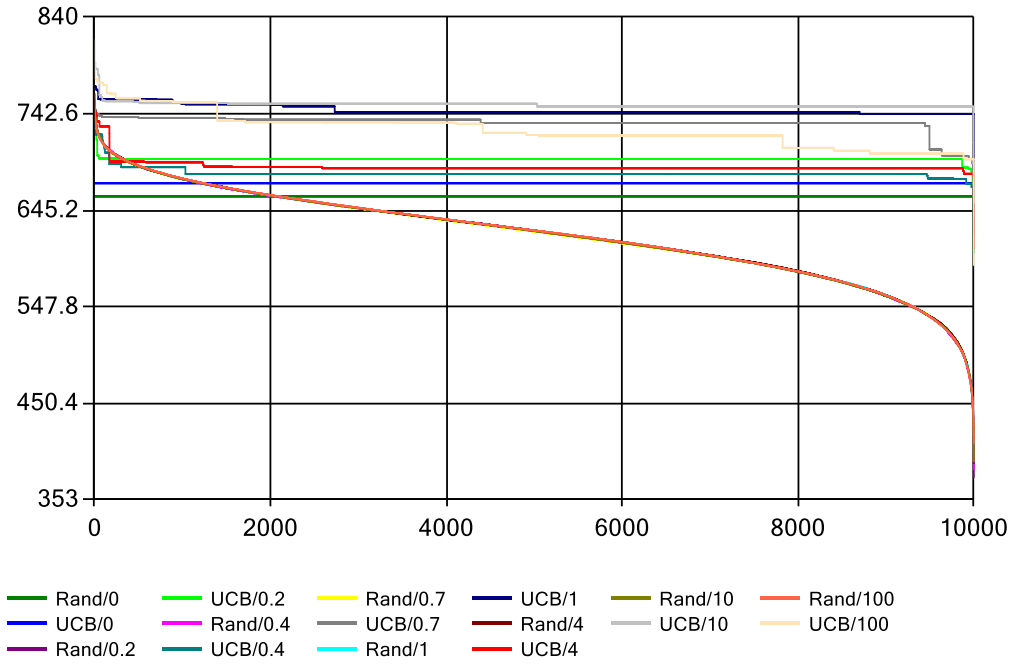


Figure 18 Measuring 9 Exploration constant in *Repaired local trees operators* – Avg. histogram of the best in generation

While in the *direct* version the exploration constant does matter, in the *repaired* does not. The *UCB-repaired* version again does not produce good solutions at all and the *randomly repaired operators* do not depend on the actual value of the exploration constant. The development of the both *repaired* versions is more like a random searching than a systematical approach. That should be the reason why the exploration constant has no impact in here.

Measuring 10 Exploration constant in Conditional operators

TSP instance: Random 15
Number of generations: 10000
Population size: 40
UCB exploration constants: 0.00, 0.20, 0.40, 0.70, 1.00, 4.00, 10.00, 100.00
UCB initialization: 0.5
Operators: All conditional – Direct, Randomly repaired, UCB-repaired

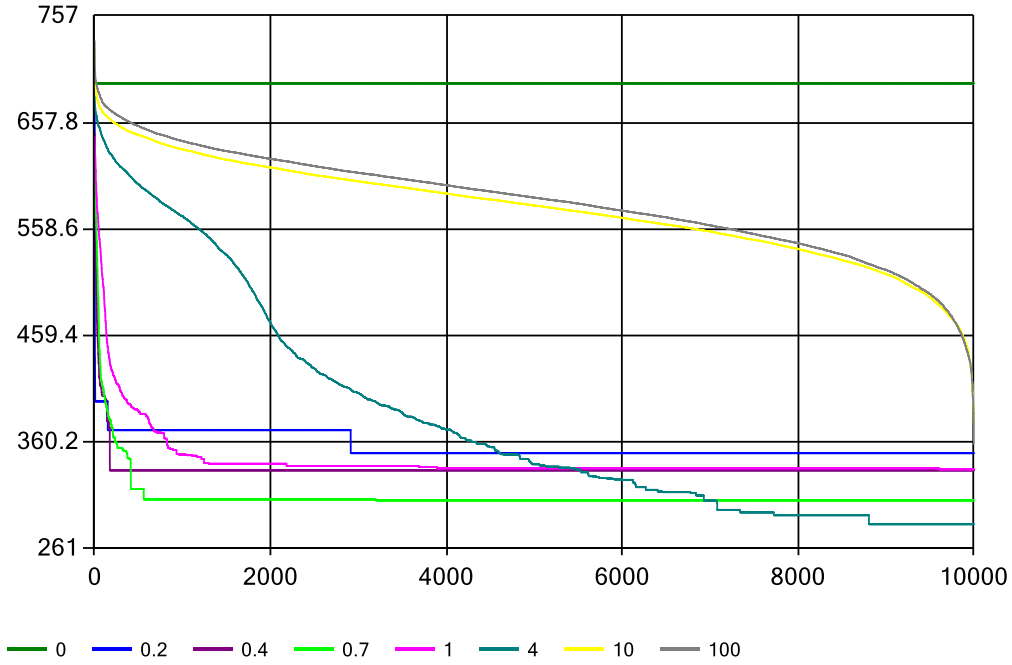


Figure 19 Measuring 10 Exploration constant in *Conditional operators*, *direct* version – Avg. histogram of the best in generation

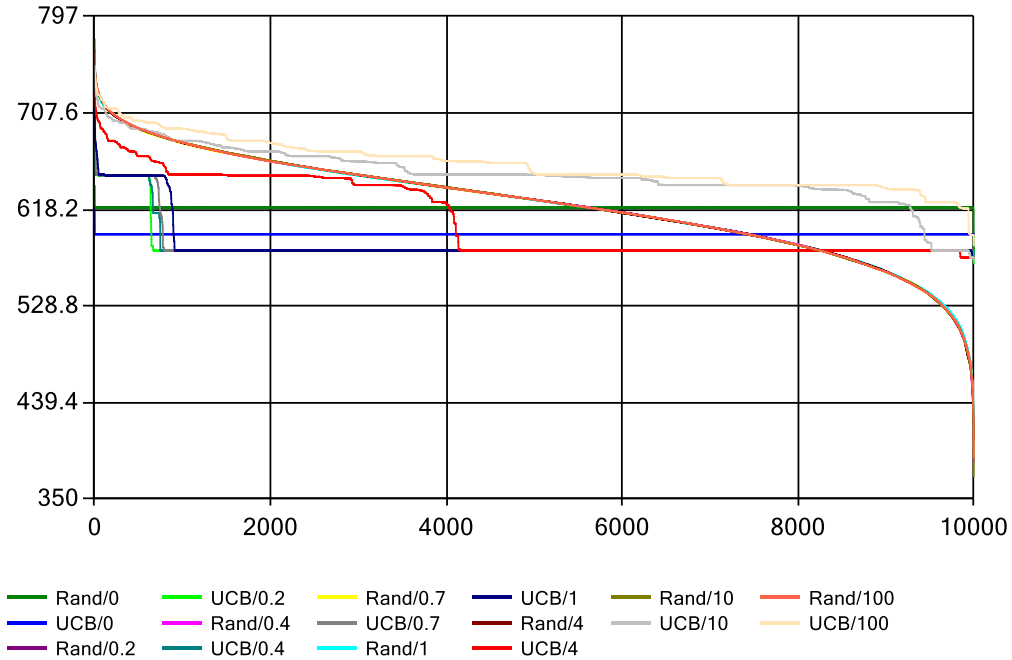


Figure 20 Measuring 10 Exploration constant in *Conditional operators*, *repaired* version – Avg. histogram of the best in generation

There is no doubt that the *direct* version of the level two operators does have shifted the reaction to the exploration constant as well. However, it is not shifted the same amount as in the level three operators. Even though the lower exploration constant values very quickly find a good solution, they do not seek for a better one due to the enormous exploitation phase. As a consequence, the exp. const. value 4 development defeats the lower values using slower but more stable convergence. As a result, we recommend using the same exploration constant values as in the level one operators.

The *repaired* operators again show that this experiment is not worth for them and we should focus on the *direct* versions.

5.3.3. Tree maturity threshold

In the following paragraphs, we are going to look closer at the level three operators and the parameters that are not present in the lower level implementations. We are intentionally going to prefer the *direct* version of the level three operators. The *repaired* version will occur in the experiments only marginally because we do not expect their positive reaction to the parameters tuning.

The one of the *local trees* parameter that was not tested yet is the "Tree *maturity threshold*". This parameter simply determines when the *local tree* can expand into a new node and when it cannot. What impact on the TSP solution quality does this parameter have, we are going to discover in the following experiments.

Measuring 11 Tree maturity threshold

TSP instance: Random 15
Number of generations: 10000
Population size: 40
UCB exploration constant: 0.40
UCB initialization: 0.5
Operators: Direct local trees, Randomly repaired local trees
Local trees maturity thresholds: 1, 2, 5, 12, 64, 512
Local trees size limit: N^5

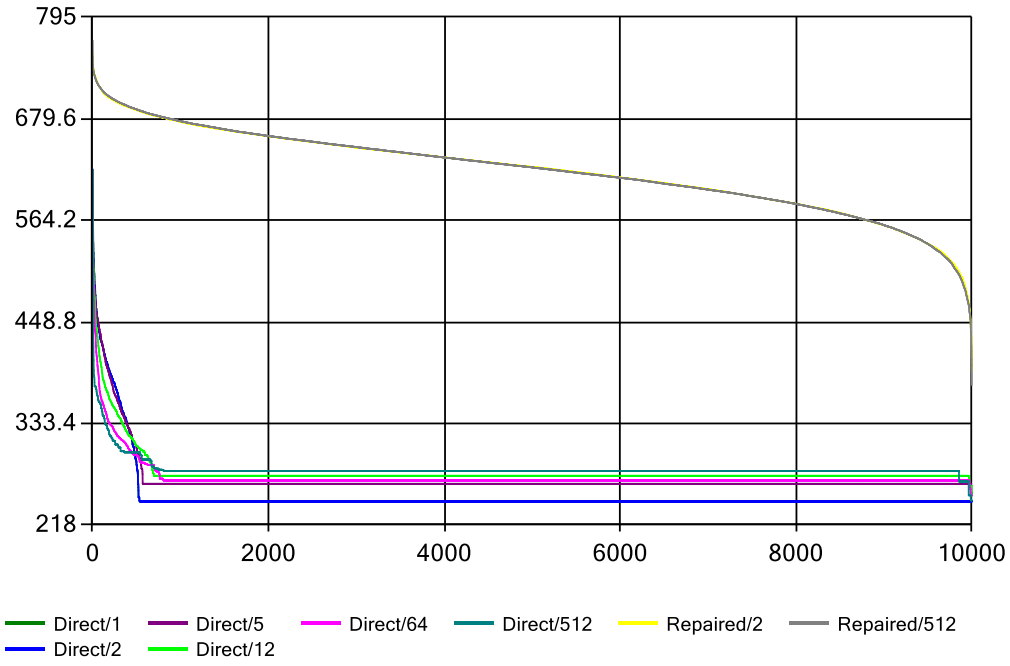


Figure 21 Measuring 11 Tree maturity threshold – Avg. histogram of the best in generation[1]

The Figure 21 shows us that the impacts of the various *maturity thresholds* are very close. Interesting is that the highest value – the Direct/512 – provides the fastest convergence at the beginning. This can be explained by the fact that at the beginning the tree that is growing slower learns more quickly than the trees that are growing faster. There are less tree nodes, among which the acquired knowledge is distributed, in the slowly growing trees. These small trees can stick the first good solutions they have seen. Of course, when enough of attempts are made, the large trees provide more quality results thanks to their specialized branches.

The *repaired* operators do not benefit from the advantage of the low *maturity threshold*. The reason probably is that the repairing mechanism completely trumps the UCB-selections and the counters of not always good options are increased. These increased worse options are not expanded by the strict expansion policy already.

[1] The green line "Direct/1" is completely hidden by the blue line "Direct/2".

5.3.4. Tree size limit

The next parameter that is present only in the level three operators is the tree size limit. This parameter directly determines the considered context length. Moreover, it also determines the actual memory and time spending of the whole algorithm. The need of the fast and practical algorithms tells us to keep this parameter very low. Regardless of it, we still need this parameter to allow the trees to learn and exploit some information. Thus, these trees should not be as small as possible.

Measuring 12 Tree size limit

> TSP instance: Random 15
> Number of generations: 10000
> Population size: 40
> UCB exploration constant: 0.40
> UCB initialization: 0.5
> Operators: Randomly repaired local trees, UCB-repaired local trees
> Local trees maturity threshold: 3
> Local trees size limits: $\frac{N}{4}, \frac{N}{2}, N, N^2, N^3, N^4, N^5$

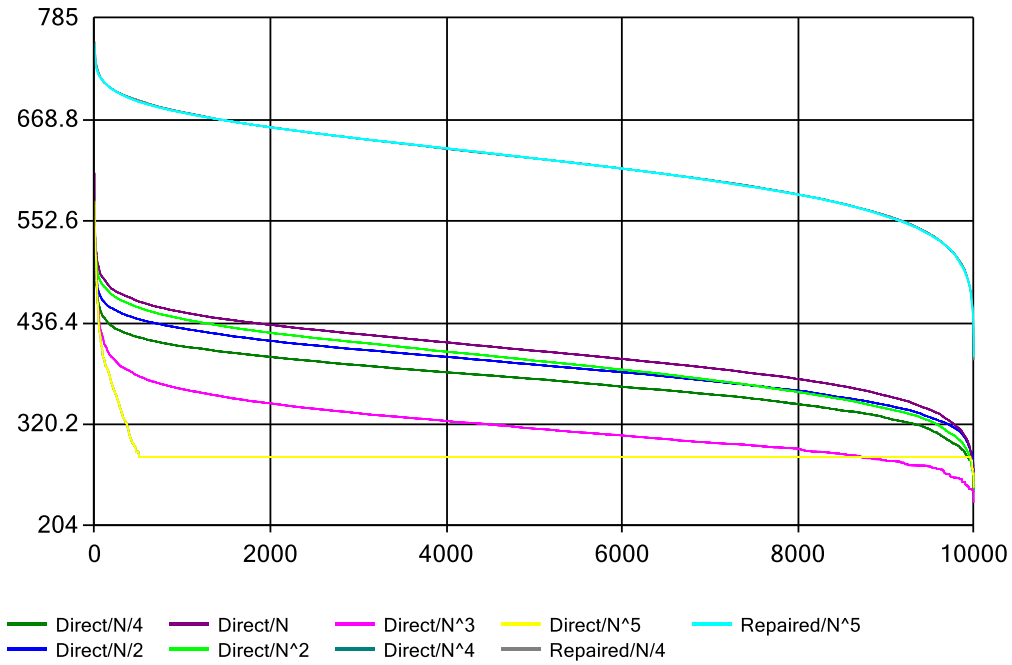


Figure 22 Measuring 12 Tree size limit – Avg. histogram of the best in generation[1]

The *repaired* operators do not show any sign of reaction to this parameter setting. The explanation could be identical to the presented in the previous subchapter (why they do not react to the *maturity threshold* parameter).

On the other hand, the *direct* operators brought into light interesting results. The operators with the generous limit repeatedly (in every experiment iteration) fall

[1] The dark cyan line "Direct/N^4" is completely hidden by the yellow line "Direct/N^5".

into a premature convergence problem and stuck in a local optimum. What is more, the slower (convergence meaning) operators with the less generous size limit do not stuck; they slowly converge and overtake the faster ones afterward. The most interesting result is the order of the rest (limits $\frac{N}{4}, \frac{N}{2}, N, N^2$). It is good to see that the lowest size limit did not end up as the worst one. However, we do not have any general explanation why the N^2 development showed less performance than for instance the $\frac{N}{4}$ development. Probably the reason would be somewhere in the particular random TSP instance which was used in this experiments.

The conclusion for the tree size limit is that we do not have to be afraid of the resources-reasonable (computer memory/time) approaches. They can be as powerful as the less limiting settings. We also should avoid the uselessly generous limits due to the resources wasting and the premature convergence threat.

5.4. Conclusion

All the experiments done in the chapter 5 have explored the various parameter settings of all our MCTS-inspired operators. We have also seen the comparison of different strategies for the operators' subroutines (UCB initialization, repairing strategies…). The settings that have turned to produce the best results, we are going to use in the rest of our experimental work. All the same, the reader should keep on mind that the optimal setting of the numeric parameters can differ in various applications. For instance, the exploration constant is directly dependent on the gain computation technique. Hence, it should be experimentally chosen for every implementation.

The main discovery is that the *repaired operators* are generally worse than the *direct* version on each corresponding level. In other words, it is better to let the *UCB selector* choose only from the really available options. Otherwise, the selector can select an option that will turn out as forbidden and the remaining options will be poor quality. However, there could be a potential application of our method where the *direct* version would be hard to implement. This is the reason why we introduced and tested the *repaired operators* as well. For the potential user, it could be a good news that the *repaired* operators do converge, but very slowly. Nevertheless, for the following improvements and experiments we aim only at the *direct* versions.

6. Extensions and improvements

In chapter 4 we have introduced the whole concept of our solution methods – operators for genetic algorithm. During the methods explanation and measuring, the astute reader has certainly gone thru parts where it occurred to him that he would construct the particular part much better than we did. We agree with the reader because we ourselves have written down a couple of ideas actually. However, we intentionally did not use them in the operators explanation and base implementation. The reason was that we did not want to make the core principle[1] any more complex for the reader. Moreover, it absolutely was not certain that these ideas would work well.

In the base implementation, we used several basic variants; in this chapter, we are going to introduce the ideas for the methods improvement. The impact of these changes we are also going to measure.

6.1. Gain computation

One of the quests of applying the UCB principle for all three levels operators was to register the individual gain $g \in [0,1]$. Our base solution is simple linear transformation of the individual's fitness (the Hamiltonian cycle weight). The gain then linearly depends on the ratio between the fitness and its estimated upper bound u.

$$g = 1 - \frac{fitness}{u}$$

6.1.1. Nonlinear gain

The problem of the linear transform is that it divides the gain value between the cycle weights equally. With this distribution, a random permutation can get relatively good gain value. Simple greedy solutions, which are very far from the upper bound estimate can reach the gain very close to 1 but are not optimal at all.

Our operators need to target more at the best solutions because the average are not interesting at all because they are beaten by very simple heuristic algorithms. There is very simple way how to focus more at the best individuals. Since the gain value comes from the interval between zero and one, we can simply apply the nth power on the value g and produce the result gain value g'.

$$g' = g^n$$

[1] Principle of MCTS inspired operators

This change brings exactly what we need. The parabolic curve indeed releases the [0,1] interval for the excellent individuals and does not much distinguish between the average and the bad solutions.

This improvement can be of course applied on any gain value computation, which we will experiment with in the case of the following gain computations.

6.1.2. Prefer the seen interval

The next problem of the base gain computation is that both ends of the interval are unreachable. They maybe are not even close to the real interval of the real Hamiltonian cycle weights. We would like to cut off these interval overlaps. Of course, if we figured out how to effectively find the real interval endings, we would not need to create any heuristic algorithm at all.

To estimate the real fitness interval, we can use only the values that we have already seen. As a consequence, the gain computation dynamically changes while more individuals are evaluated. The current upper bound estimate u_t for the fitness is the worst fitness seen yet and the current lower bound estimate l_t is the best fitness in the moment. The gain g is then placed into the current interval.

$$g = 1 - \frac{fitness - l_t}{u_t - l_t}$$

This dynamic computation causes that the gain of the same individual changes thru the time. This should not significantly hurt the self-balancing mechanism of *UCB selection*. When the gain of the particular individual gets lower, the operator will find out quickly because it will try to exploit it. On the other side, when the gain will increase, it implies that there is a better individual, which will be exploited (otherwise, the gain would be just one). However, the self-balancing mechanism will after some period try to generate this individual again and the gain will be slowly updated.

6.1.3. Prefer above average

Since we are seeking for the best solution of all, we want to make the curve steeper (the same motivation as for the nonlinear gain). In the previous chapter, we have reduced the actual gain interval to the real fitness occurrences. However, we do not stop there. We can afford to reduce the interval even more because we are not

interested into the bad solutions. Therefore, we store the current fitness average $\overline{f}_t$ instead of the upper bound estimate. This is the result formula:

$$g = 1 - \frac{fitness - l_t}{\overline{f}_t - l_t}$$

6.1.4. Impact measurement

Let us see, how the different gain computations will impact the operators' behavior. We are going to compare the various gain computations on all versions of *direct* operators. Before that, we bring a small overview of the gain computations. On the illustration picture below we can see the differences in gain computations' dynamic.

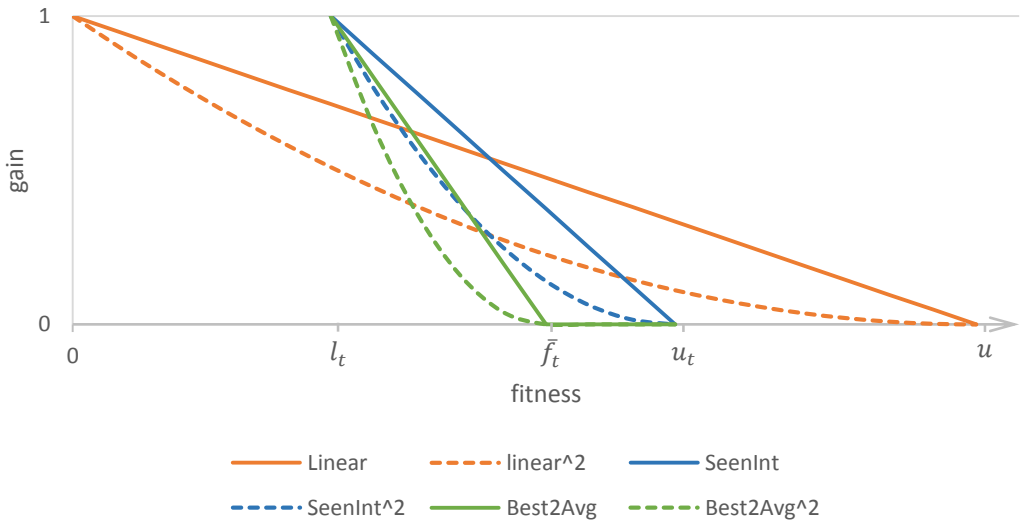


Figure 23 Gain computations' dynamic - illustration figure

Measuring 13 Gain computation

TSP instance: Random 15
Number of generations: 10000
Population size: 40
UCB exploration constants: 2.00 (levels one, two), 0.40 (level three)
UCB initialization: 0.5
Operator: Direct single allele selecting, Direct conditional, Direct local trees
Gain computations: Linear (base), Seen interval preferred, Above average preferred
+ quadratic (only on single allele) and quartic version of each
Local trees maturity threshold: 3
Local trees size limit: N^3

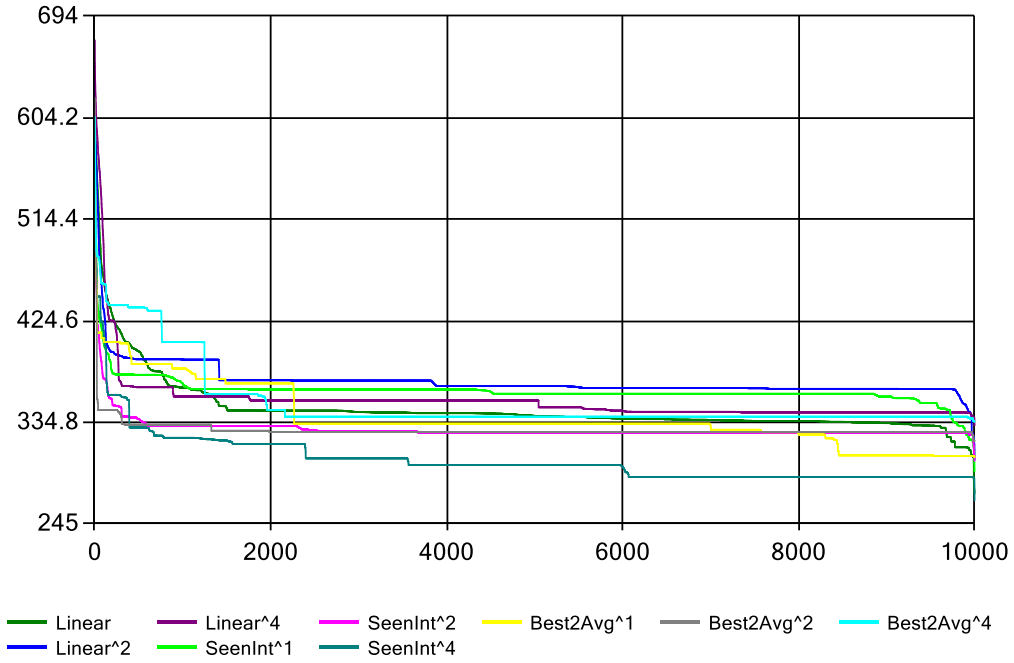


Figure 24 Measuring 13 Gain computation, Single allele selecting operators – Avg. histogram of the best in generation

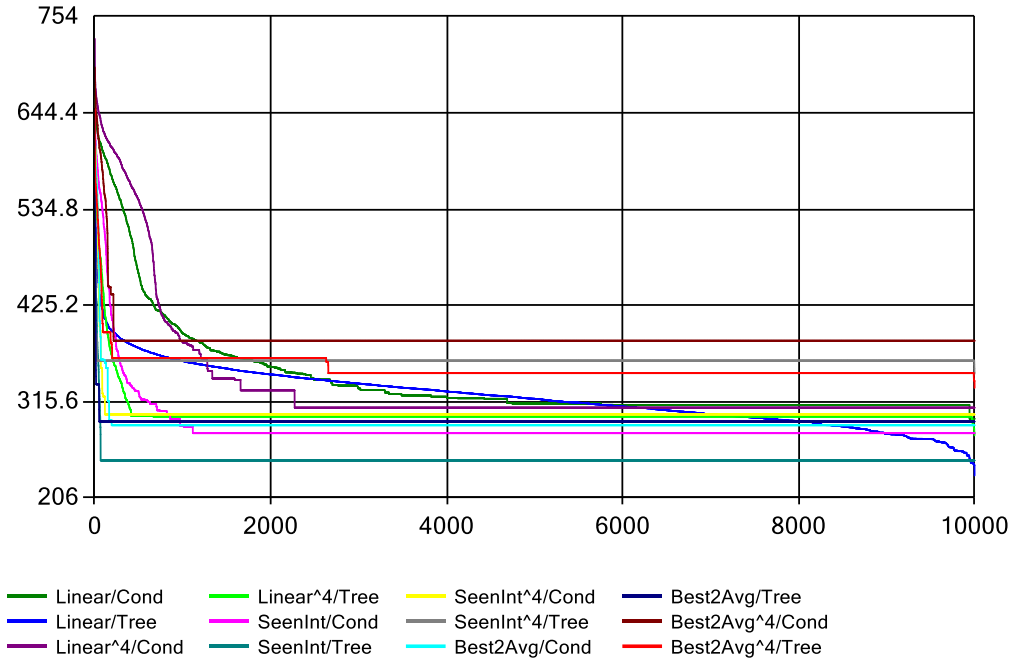


Figure 25 Measuring 13 Gain computation, *Conditional* and *Local trees operators* – Avg. histogram of the best in generation

After analyzing the measuring results, we can hardly say, that any of the gain computation technique is clearly better than the other ones. The improved techniques (*prefer seen internal*, *prefer above average*) in average give slightly better results; nevertheless, the base implementation (called "Linear") also shows itself as a competitive. The quartic extension (fourth power) works well mainly for the base technique and in the case of the *single allele selecting operators*. On the other hand, the improved techniques cause doing more exploitation for the higher level operators. This is also obvious on the other charts from this measuring, which are located on the CD attached to this paper.

We presume that the actual impact of the gain computation techniques would differ if we were using various exploration constants. However, we are not going to measure every possible combination of the system settings. Therefore, for potential future usage of our operators we recommend doing tests of the particular setting that is going to be used.

For the rest of the tests in this thesis we should pick one of the gain computation techniques. The most generally successful seems to be the "Prefer seen interval" (SeenInt) technique.

6.2. Less strict expansion policy

In chapter 4.5 we adopt the idea of MCTS and put it into the *local trees operators*. This first implementation draft did not meet the EA operator basic assumptions (to be simple and quick): it was very complex and slow. This was caused by the very large trees quickly expanded in the memory. That was the reason for us to use the *maturity threshold* parameter, which is a common technique used for limiting the expansion in MCTS. The second and very effective technique for reducing the tree expansion was the "Third quartile policy". Nevertheless, this policy is not part of the original MCTS algorithm.

Our "Third quartile policy" keeps the trees targeting on the best-known branches. However, the optimal solution could be hidden in a branch that does not begin with a highly rated option. In other words, the "Third quartile policy" can discard complex combinations that are not so obvious to choose.

Let us spend one measuring to find out whether the speculation from the previous paragraph is correct. We will try to disable this policy and look for the different behavior.

Measuring 14 Without quartile policy

> TSP instances: Random 15, Triangle unequal 15, Grid 5x3
> Number of generations: 200
> Population size: 40
> UCB exploration constant: 0.40
> UCB initialization: 0.5
> Operator: Direct local trees operator
> Gain computation: Prefer seen interval
> Local trees maturity threshold: 3
> Local trees size limit: N^3

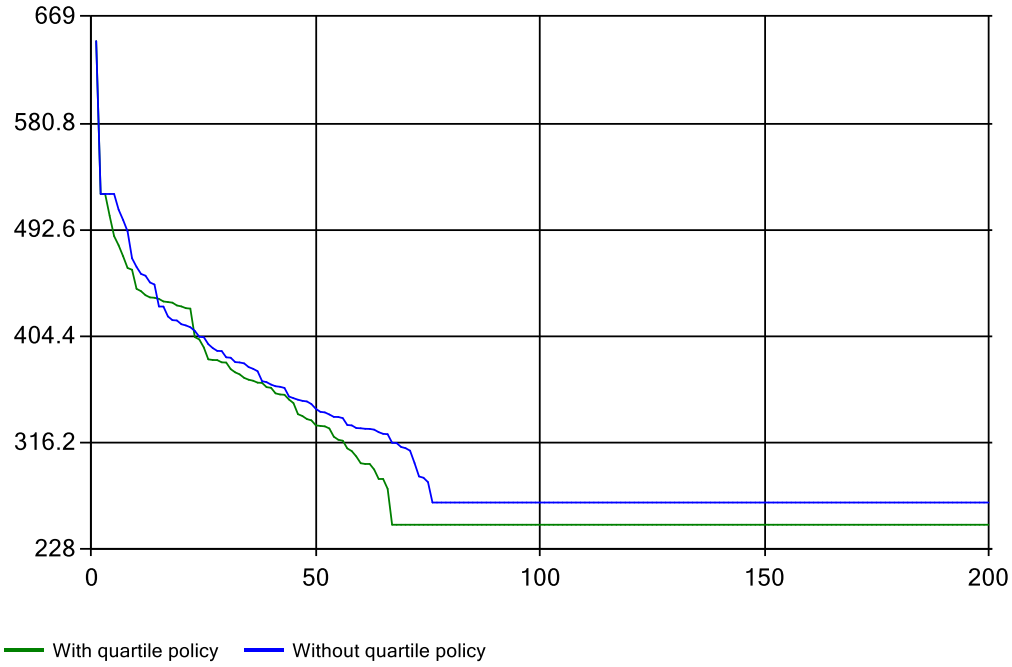


Figure 26 Measuring 14 Without quartile policy – Avg. histogram of the best in generation, Random 15

The measuring results have brought good news. It seems that the "Third quartile policy" causes no good solutions elimination. We were also afraid that this measuring highly depends on the TSP input (the graph). Hence, we did this measuring on the more special TSP instances too. Not always the version with the policy returns slightly better results, but in every test run the results were very close.

6.3. Other approaches cooperation – evolutionary computation

This part is the highlight of this text. We are going to fulfill the subject of this thesis. So far, we have presented our operators for an evolutionary algorithm. However, the reader could certainly have a comment at the moment: *The operators could be called algorithms and run in a loop (in a programming meaning). Only the case of data representation of the TSP solution, the whole evolutionary framework and*

the genetic background could be omitted without loss of generality. That might be true, but we are heading to the extension that needs the evolutionary algorithm design.

We are going to make our operators a part of the evolutionary computation.

6.3.1. Evolutionary operators for TSP

In order to demonstrate our ideas about integration of our operators into the evolutionary computation, we need to use some of the known techniques for evolutionary TSP solving. There is a lot of EC[1] papers focused especially on TSP. Many sophisticated operators are developed and tuned to get the best results from the EA[2]. Despite this, we get by with very basic and simple operators. It will be enough to show the principle and we believe that with the more sophisticated operators would be the result only better.

In our experiments, we use one type of genetic crossover and one type of mutation. The used crossover is going to be the OX (Ordered crossover) [17]. The mutation is a simple change of the Hamiltonian cycle. It removes one city from the sequence and inserts it between two cities that were originally immediately behind. The moved city and the target position are chosen randomly.

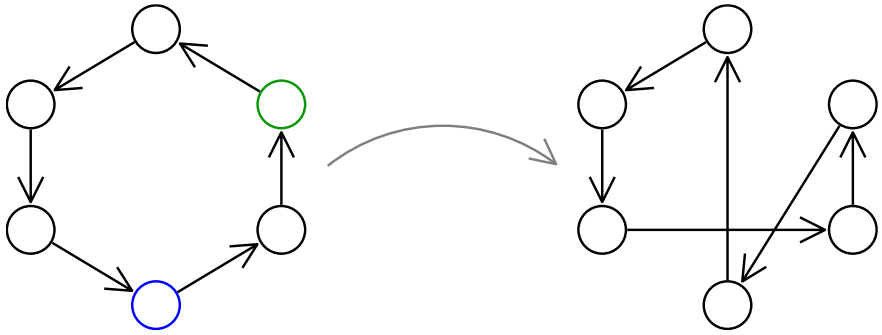


Figure 27 Simple random mutation example. The chosen city is blue and its new predecessor is green.

6.3.2. Cooperation

Technically, we have available several operators for evolutionary TSP solving. Some of them are classical and some are the MCTS inspired[3]. The operators could be now put into the general evolutionary algorithm and the whole process could be started. However, we have one more improvement.

[1] Evolutionary computation

[2] Evolutionary algorithm

[3] The MCTS inspired are all the operators introduced in this thesis (in chapter 4) – all three levels.

The cooperation among the operators is not complete yet. A classical operator can get as an input an individual that was produced by a MCTS operator. The classical operator changes it and can improve it. On the other hand, the only input registered by the MCTS operators is the gain information, which is based on the individual that was just produced by the actual operator. Hence, the operators developed by us cannot gather any information from other operators' products.

There is one simple solution for the cooperation problem. All the MCTS inspired operators have something in common. When they produce an individual, they compute the gain of the particular individual and register it (the gain) in their inner data structures (single *UCB selectors*, *conditional* selectors, *local trees*). This gain registration we now apply on all the individuals that will be produced by any of the operators in the EA set. This will bring more information into our MCTS operators without any additional expenses (except the gain computation and registration). At this point, the cooperation among the operators is complete.

6.3.3. Measuring

We find the complete cooperation EA very promising because this is the place where two different approaches meet. To verify our beliefs we are going to do some experiments. We are going to compare all levels of our *direct* operators with the classical evolutionary TSP solution[1] and moreover, we are going to measure the improvement of the full cooperation.

[1] The implemented basics

Measuring 15 Evolutionary TSP compare and cooperation

TSP instances: Random 15 (all levels), Random 100 (only levels two, three)
Number of generations: 800 (level one), 200 (levels two, three)
Population size: 40
Selection: Tournament, winner proceeds prob. 0.9
UCB exploration constants: 2.00 (levels one, two), 0.40 (level three)
UCB initialization: 0.5
Operator: Direct single allele selecting / Direct conditional / Direct local trees
+ OX (prob. 0.8), Random mutation (prob. 0.6)
Gain computation: Prefer seen interval
Local trees maturity threshold: 3
Local trees size limit: $\frac{N}{4}$

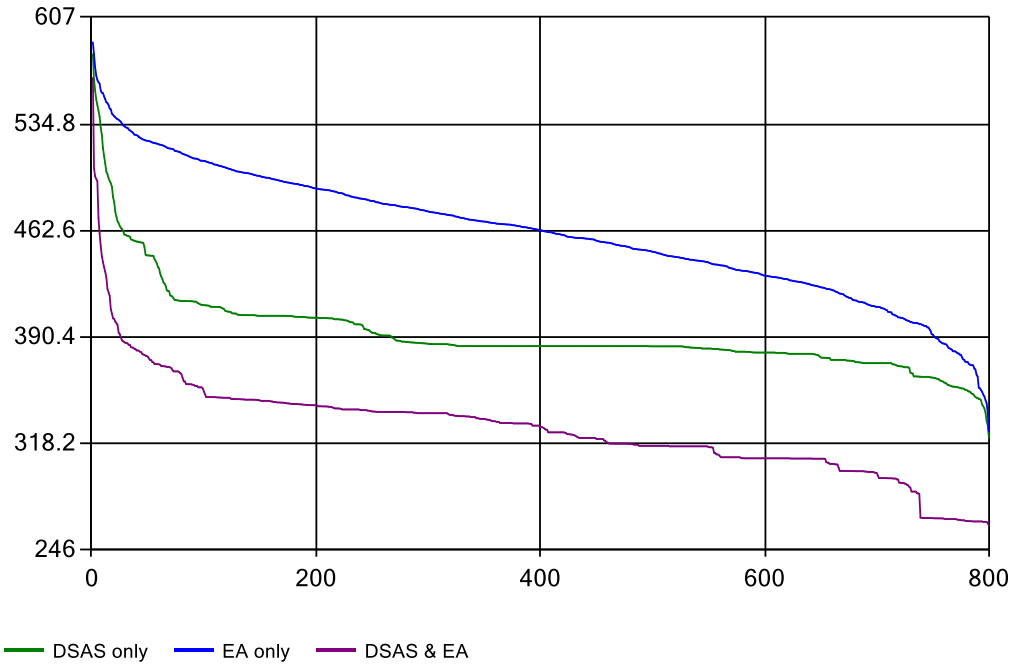


Figure 28 Measuring 15 Evolutionary TSP compare and cooperation – Avg. histogram of the best in generation, *Direct single allele selecting*

The first part of the measuring using only the level one operator brings very good news. Already the level one operator shows better results development than the basic evolutionary approach. What is more, the cooperation indeed improves both techniques, which is a great outcome. Let us see how what will be the effect in the case of the level two and three operators.

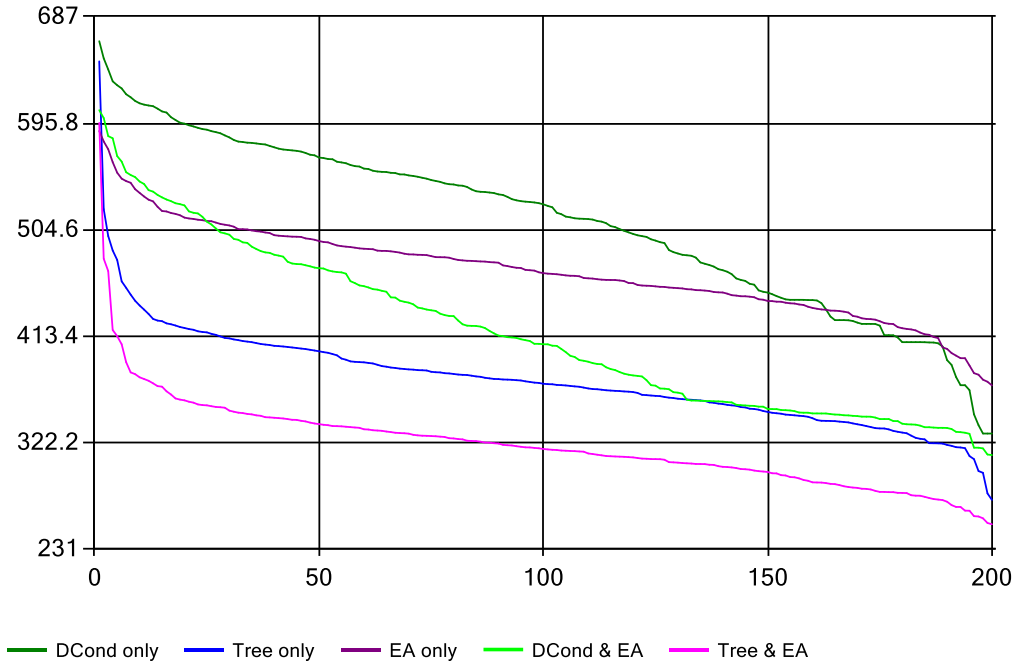


Figure 29 Measuring 15 Evolutionary TSP compare and cooperation – Avg. histogram of the best in generation, *Direct conditional*, *Direct local trees*; Random 15

The results are perfect in the case of the higher levels operators too. We can observe how the performance of any of the methods increases when we are using the cooperation.

On the box plot charts, which are attached on the CD with other measuring outputs, we can see the possible explanation of this success. Although we have tuned the exploration constant for the best results, the dispersion of the population generated by our operators is smaller than in the case of classical evolutionary TSP approach[1] or the cooperation mode. Hence, the classical operators help us mostly with the exploration part. They search the possibilities that the given system of *UCB selectors* did not yield yet. Our operators then get a broader view earlier than they would create it by themselves. As a consequence, this symbiosis works very well.

[1] With little higher mutation and crossover probabilities than commonly used.

7. Performance tests

In the previous text, we have showed that our operators can converge toward the fitness function (look for the lowest values). We have also tuned their parameters and applied several improvements. In chapter 6.3 we have compared our operators with the basic classic genetic operators for TSP. There we have also found out that the cooperation of our operators and the classical ones brings better results earlier[1] than the any of the separated runs.

In this chapter, we are going to test whether our approach can handle larger TSP instances than the base ones that were used previously. We are also going to compare the results with other non-evolutionary heuristic approach. Then we are going to check the time requirements of each type of our smart (unusually complex) operators. At the end, we are going to test other TSP types than the general random graphs.

The result quality will be compared with non-evolutionary heuristic method: greedy algorithm. We have prepared two versions of the greedy algorithms (**Grd1**, **Grd2**). The **Grd1** algorithm works as follows:

1. Start with one vertex.
2. Choose pair (v, i) that minimizes the result cycle weight increase. Where v is one of the unused vertices and i is the v's target position in the result permutation (N^2 minimization step).
3. Repeat step 2 until the full permutation is built.

On the other hand, the **Grd2** algorithm is simpler:

1. Start with one vertex.
2. Choose the nearest unused neighbor as the successor.
3. Repeat until the Hamiltonian cycle is built.

7.1. Various problem sizes

In the following experiments, we check whether our approach can even compete with the greedy algorithms. We test it on various problem sizes.

[1] In the generation number meaning.

Measuring 16 Various sizes

TSP instances: Random 15, 30, 50, 80
Number of generations: various
Population size: 40
Selection: Tournament, winner proceeds prob. 0.9
UCB exploration constants: 2.00 (levels one, two), 0.40 (level three)
UCB initialization: 0.5
Operator: Direct single allele selecting / Direct conditional / Direct local trees
+ OX (prob. 0.8), Random mutation (prob. 0.6)
Gain computation: Prefer seen interval ^ 4
Local trees maturity threshold: 3
Local trees size limit: $\frac{N}{4}$

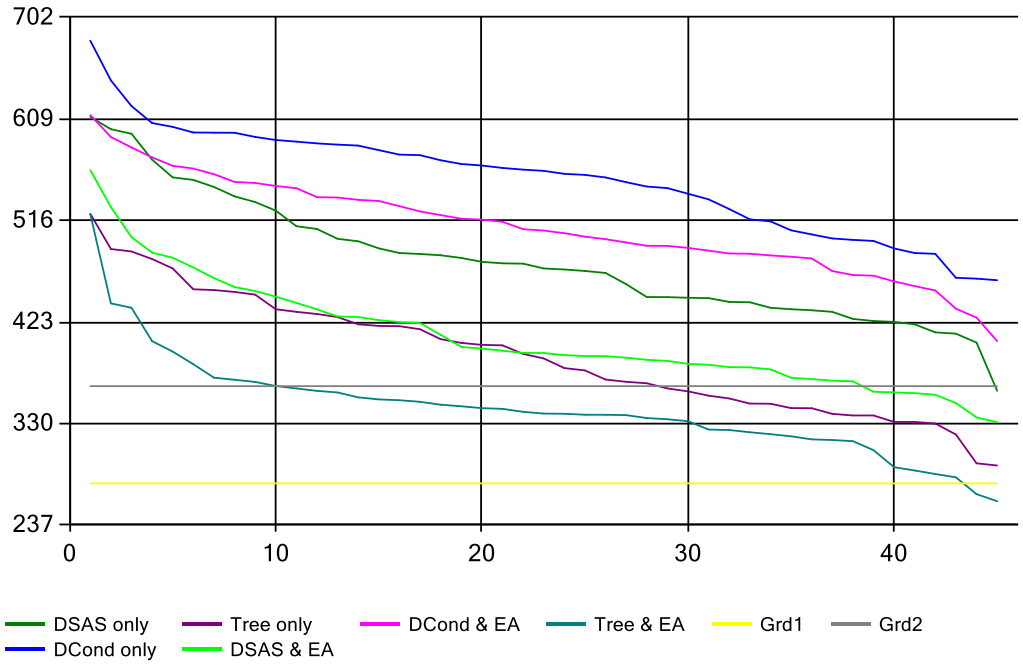


Figure 30 Measuring 16 Various sizes – Avg. histogram of the best in generation, Random 15

We can see that only 45 generations are enough for the best cooperation to beat the better of the greedy algorithms. What is more, the chart above shows the average results. The Figure 31 recorded during the fifth run shows that the *local trees operator* in cooperation with classic evolution overtook **Grd1** algorithm in the 22. generation.

When we continue the evolution, other lines cross the yellow base. After 450 generations, only the lines **DCond only** and **DSAS only** did not overtake the greedy result. The chart is of course available on the attached CD.

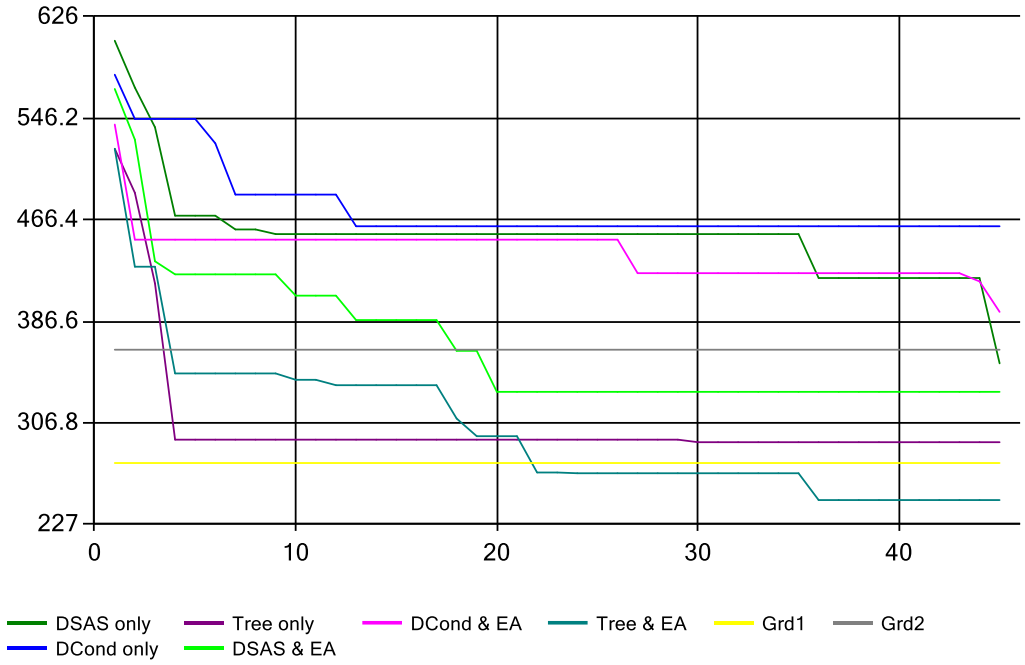


Figure 31 Measuring 16 Various sizes – The best found solution, Random 15

The larger inputs do not significantly change the general behavior. However, a few new observations can be made. As we can see on Figure 32, the distance between the **Tree & EA** and the other methods is even more significant than it was at the smaller input. In addition, the rest cooperative methods are more competitive than they seemed to be at the smaller inputs. On the other hand, the level one and level two operators separately do not bring good results. They would need a lot more generations or maybe different setting.

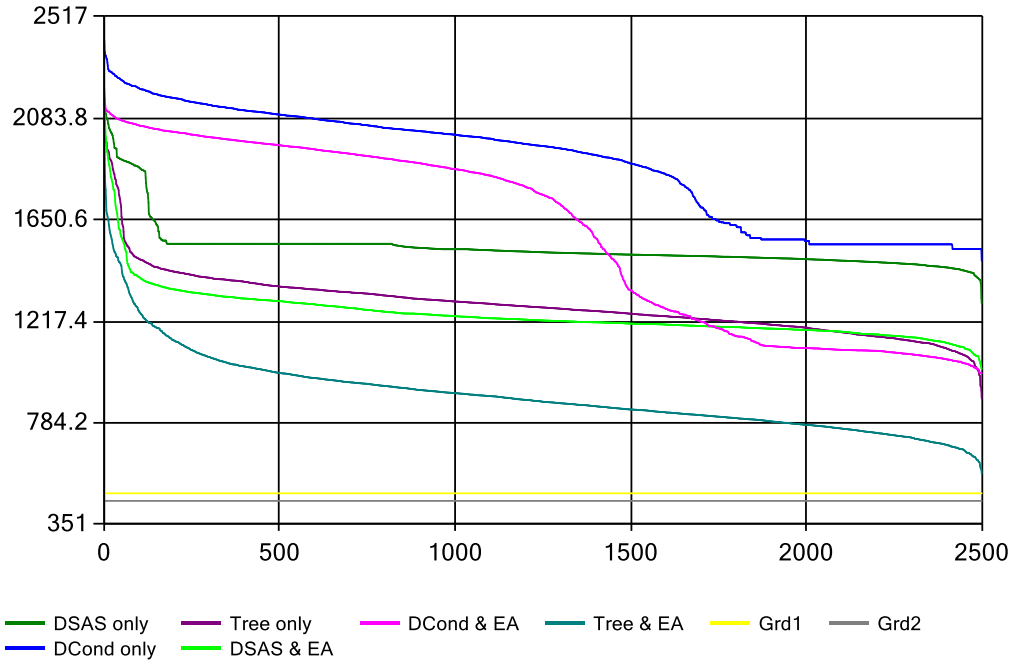


Figure 32 Measuring 16 Various sizes – Avg. histogram of the best in generation, Random 50

In the experiments with the higher input sizes, we have found out, that one of the parameters shows different characteristics than we have expected. It was the gain computation. When we introduced the gain computation in chapter 6.1, the tests showed that the all of the improved[1] techniques have a very similar characteristics. The larger inputs showed that the difference among the techniques does not appear until the increase of the fitness dispersion. The quartic extension of the "Prefer seen interval" has turned to be a very good approach and so we have used it for the final measuring and the charts above.

The tests, made on the larger inputs, also showed, that the needed number of generations grows very fast. We used quadratic determination ($\#gen = N^2$), which did not turn out as sufficient. The input Random 50 is the last one where we reached the greedy algorithms' results. Nevertheless, we would not recommend setting higher number of generations. We would rather suggest using any problem specific heuristics or some advanced classical TSP operators in cooperation with the MCTS-inspired operators. The reason is the high time spending, which we are going to look onto in the following chapter.

[1] All the computations suggested as an improvement in chapter 6.1.

7.2. Time complexity

We saw that our approach can bring nice results, but as the input size increases, the needed generations count is growing very fast. Nevertheless, the number of generations is not very practical information. We have not yet outlined how increases the actual time spending, which is very important for the potential user.

7.2.1. Time spending per generation

In order to get an overview about the operators' time complexity, we have prepared a special experiment. We run our operators separately for the same amount of generations. We repeat this on various input sizes. Each run is measured by stopwatch and the result time is divided by the number of generations. As a result, we get the average generation duration per operator and input size. The time results we compare also with the classical operators for TSP.

Measuring 17 Time spending

> TSP instances: Random 15, 30, 50, 80, 100, 500
> Number of generations: 200, 400
> Population size: 40
> Selection: None (MCTS operators), Tournament, winner proceeds prob. 0.9 (classical operators)
> UCB exploration constants: 2.00 (levels one, two), 0.40 (level three)
> UCB initialization: 0.5
> Operator: Direct single allele selecting / Direct conditional / Direct local trees / OX (prob. 0.8), Random mutation (prob. 0.6) - EA
> Gain computation: Prefer seen interval
> Local trees maturity threshold: 3
> Local trees size limit: $\frac{N}{4}$

		Time per generation [ms]					
Generations	**Operators\Size**	**15**	**30**	**50**	**80**	**100**	**500**
200	DSAS	0.63	1.08	2.46	5.85	8.81	197.43
	DCond	0.46	1.26	2.73	6.76	10.18	218.33
	DTrees	5.96	5.82	10.31	19.66	29.05	691.00
	EA	0.88	0.90	1.14	1.55	1.76	9.97
400	DSAS	0.39	1.05	2.43	5.68	8.56	197.90
	DCond	0.47	1.19	2.93	7.02	10.41	196.21
	DTrees	4.28	5.56	9.43	17.92	28.23	643.04
	EA	0.73	0.89	1.16	1.53	1.83	9.98

Table 2 Measuring 17 Time spending – Time per one generation

At the first sight, it is obvious that our "smart" operators spend much more time than the classical evolutionary approach. Naturally, the fastest-growing time spending is caused by the *local trees operator*. This shows that our operators are not very time efficient. We think that this could be changed by some implementation optimizations. Our implementation is not primarily aimed at efficiency; our implementation is more

illustrative and tries to help understand the principles of our algorithms. However, since our operators are “smart”, memetic and Lamarckian, their time efficiency will be generally always worse than the classical simple evolutionary operators will.

7.2.2. Actual performance

As we have found out in the previous experiment, our “smart” operators spend much more time for one generation than the classical evolutionary TSP approach. The potential end-user does not care about the number of generations or how long one generation takes. The end-user compares the methods by the ratio between the result quality and total time spent. This ratio means the actual performance of the approach. To provide this type of comparison, we perform the next measuring.

We fix an absolute time limit for the run and we compare the results achieved by the various methods right in the time limit. The limit is of course the same for all the methods.

Measuring 18 Performance in limited time

TSP instances: Random 15, 30
Time limit: $5N^2$ milliseconds
Population size: 40
Selection: Tournament, winner proceeds prob. 0.9
UCB exploration constants: 2.00 (levels one, two), 0.40 (level three)
UCB initialization: 0.5
Operator: Direct single allele selecting / Direct conditional / Direct local trees / OX (prob. 0.8), Random mutation (prob. 0.6) – EA / combinations
Gain computation: Prefer seen interval ^ 4
Local trees maturity threshold: 3
Local trees size limit: $\frac{N}{4}$

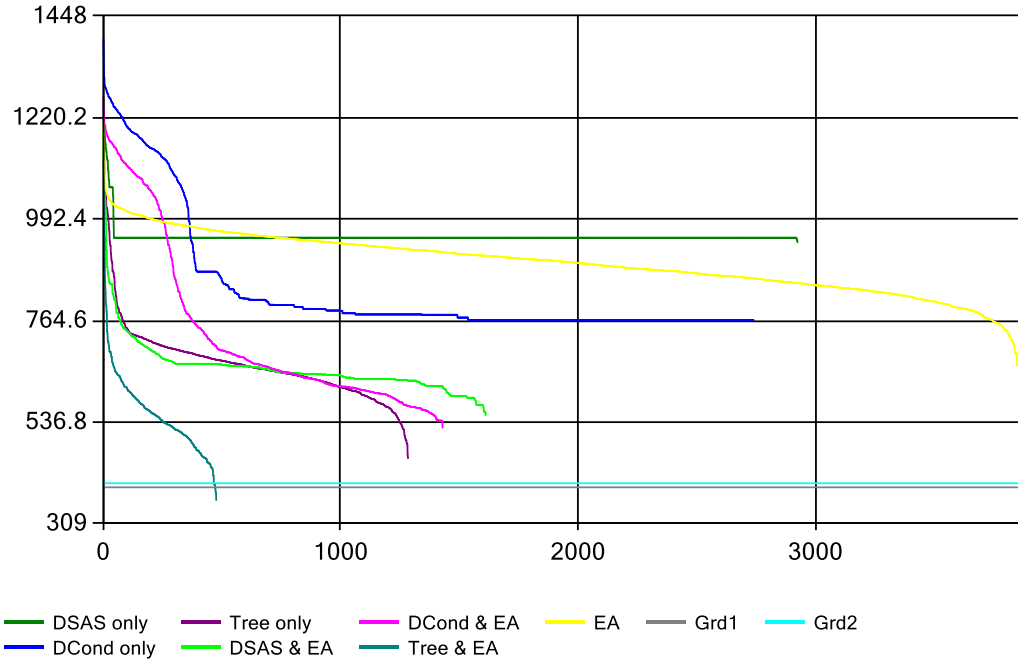


Figure 33 Measuring 18 Performance in limited time – Avg. histogram of the best in generation, Random 30[1]

This experiment is the most interesting in this paper. It brings the full and clear overview of all the mentioned operators and their combinations.

The number of samples (horizontal axis) shows the number of generations that each of the methods did reach. It totally corresponds with the results of **Measuring 17 Time spending**. Only the usages of our operators singly could reach more generations than they did. The reason of their slowdown is the used Tournament selection, which was pointless because the single operator produces the same number of children as it

[1] This experiment cannot be measured repeatedly and then averaged. The reason is that other runs could reach different number of samples (generations). The shown graph type is "Avg. histogram of the best in generation", but the actual average is always from one record in this case. The whole measuring was, of course, done more than one times. The results are on the attached CD.

was in the previous generation. On the other hand, the Tournament selection was used in all the measured methods; hence, the results are fair.

The only run that beats the greedy algorithms in the chosen time limit is the *direct local trees operator* in cooperation with the classical evolutionary TSP. This method also reached the lowest number of generations. This shows us that the "smart"[1] and slow operators could be very efficient. It also again proves how advantageous is the cooperation approach.

7.3. Special TSP types

All the experiments, which we have made, use generic random TSP inputs. If we look at real problems in practice, we could find out, that the graphs are not completely random. They satisfy several conditions and various practical applications of TSP contain only graphs from special classes. For instance, the triangle inequality is the one of the usual graph's attributes.

If there is some special class of graphs expected, the solving algorithm can use more specialized heuristics. There are also classes where a specialized efficient algorithm can find the optimal solution or at least the solution with constant ratiometric error.

Our operators do not use any class-specific heuristics. So, we are going to find out whether they work on the special TSP inputs too and whether their characteristics will change. The inputs in these experiments were already described in the chapter 5.1.2.

[1] The smartness is redeemed the grater memory allocation. The learned information makes the operator smart.

Measuring 19 Special TSP types

TSP instances: Triangle unequal 15, 30, 50, Grids 5x3, 6x5, 10x5
Number of generations: various
Population size: 40
Selection: Tournament, winner proceeds prob. 0.9
UCB exploration constants: 2.00 (levels one, two), 0.40 (level three)
UCB initialization: 0.5
Operator: Direct single allele selecting / Direct conditional / Direct local trees
+ OX (prob. 0.8), Random mutation (prob. 0.6)
Gain computation: Prefer seen interval ^ 4
Local trees maturity threshold: 3
Local trees size limit: $\frac{N}{4}$

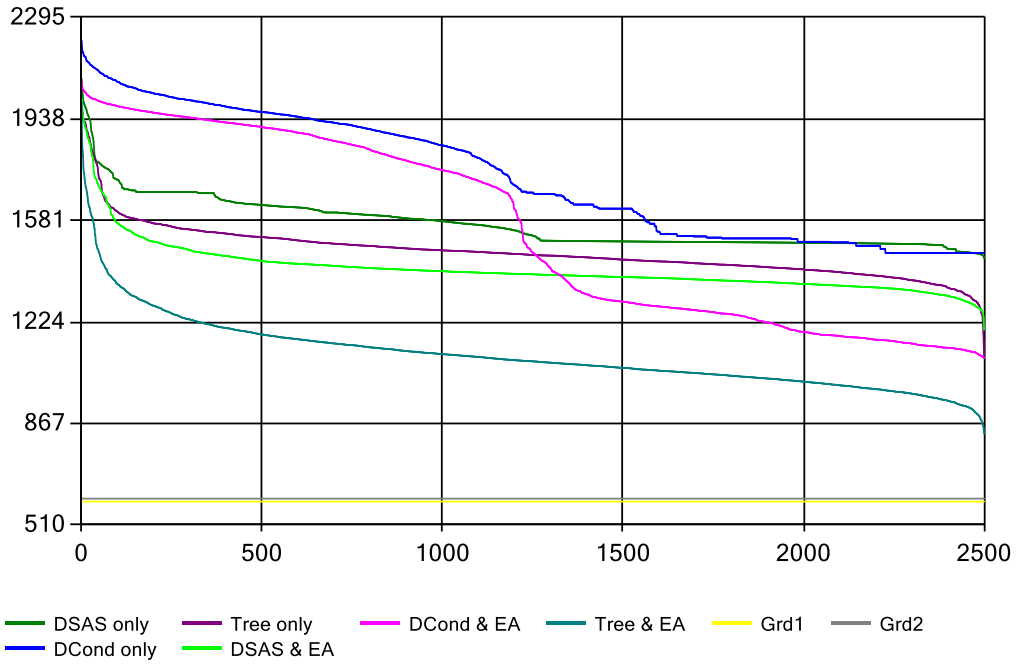


Figure 34 Measuring 19 Special TSP types – Avg. histogram of the best in generation, Triangle unequal 50

If we compare the result charts of the Triangle unequal or Grid inputs with the Random input of the same size, we see very similar developments. The only slight difference can be observed on the lines' slope. It seems to be little bit less steep at the special TSP types. This could mean, that our approach has a little bit greater tendency to get stuck in a local optimum at this TSP types. That can be caused by the special graph conditions that can produce more different solutions with a very similar weight. For example, the Grid inputs have many symmetrical solutions.

The slow convergence and local optimum problem could be prevented by setting higher exploration constant. However, we think that the higher exp. constant would not help a lot. Much better would be some symmetry breaking modification of the whole method, or some heuristic helper in our operators. The heuristics could be applied inside the gain computation for instance.

Especially in the case of the Grid instances, the results are very similar to Random inputs as well. When the specified number of generations is enough, our method is able to find even the optimal solution.

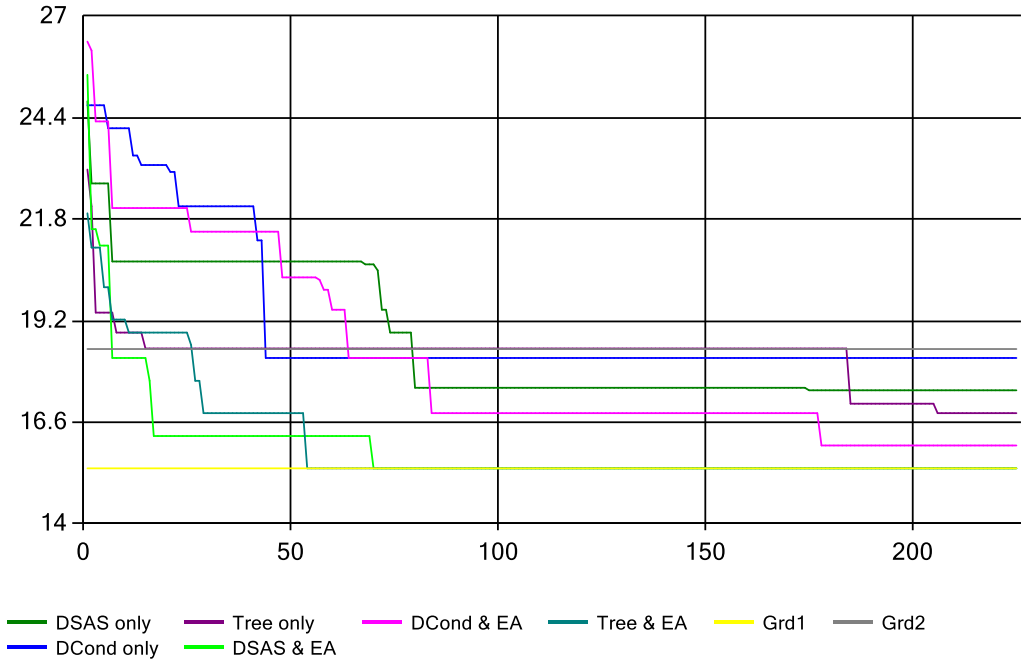


Figure 35 Measuring 19 Special TSP types – The best found solution, Grid 5x3

On Figure 35 we can see that the *direct single allele selecting operator* and *direct local trees operator*, both in cooperation with the classical evolutionary approach, found the optimal solution $(15 + \sqrt{2})$ very quickly. The greedy algorithm **Grd1** found the optimal solution as well.

Conclusion

This thesis brought a problem solving method, which is based on the combination of the two different techniques: Monte Carlo Tree Search and Evolutionary algorithms. This method is divided into six versions (three levels, each level with two approaches). For every version, the best-found parameters configuration and setting was mentioned. As the better approach turned out to be the *direct* variant. Unsurprisingly, the best results are yielded by the most complex variant – *direct local trees operator*.

The greatest success of this thesis is the cooperation mode, where the *direct* versions of our operators are collaborating with the classical evolutionary operators for TSP. This cooperation produces better results than any of its parts separately. Even in the same absolute computation time.

Advices for potential use

Our method is designed to be able to be used wherever the classical genetic evolutionary algorithm can be used. The only thing that is required is the data representation in a finite vector (chromosome) where each component (allele) is filled by an element from a finite domain. The possible constraints among the alleles can be solved by one of the proposed approaches: *repaired* or *direct*. The experiments showed that the *direct* variant is more powerful. However, there can be possible applications where the *direct* variant would be very hard to implement.

Many performed experiments showed that the best-found parametrizations and settings are very influenced by the input and by the fitness function computation and dispersion. For the potential application, we suggest to take some sample group of the possible inputs and do the parameters tuning again. The finally chosen setting can differ from the one that was proposed by this thesis. We also admit that it could be possible to find better settings than we did on the same inputs. The number of combinations is very large and we could not try all of them.

Future work

To obtain better results than we did, there are a lot of possible improvements of our methods. As we already did mention in the text of this thesis, there can be used

some problem-specific heuristics in our method. One of the possible places for a heuristics placement is the gain computation.

Another possibility for enhancing the performance is the usage of other classical operators for evolutionary TSP solving. In general, the more or the better operators will be used in the cooperation, the smarter the MCTS operators will get. Very interesting would be to use evolutionary operators that use different data representation of the individual. These operators could help with the exploration in the directions, which the original representation is not easy to transform towards. Of course, the both-way individual conversion is needed. Actually, this is exactly what we implemented for the purpose of using the OX crossover. Therefore, we suggest trying to combine and to cooperate with other unusual representations.

Another space for possible development is in the MCTS operators' core. There are a lot of papers enhancing Monte Carlo Tree Search. Using some of the enhancements could bring again better results. Very interesting would be for example an integration of RAVE technique for MCTS [18] into our operators.

Bibliography

[1] B. Hall and B. Hallgrímsson, Strickberger's Evolution, Sudbury, Massachusetts: Jones and Bartlett Publishers, 2008.

[2] A. Brooks, R. Brown, C. Chen, M. Daly, H. Dinh, E. Hama, R. Hinman, J. Ng, M. Sneddon, H. Troung, J. Wang and C. F. Yung, "Bioalgorithms.info," 1 8 2004. [Online]. Available: http://bix.ucsd.edu/bioalgorithms/presentations.old/Ch03_molbio.pdf. [Accessed 10 10 2015].

[3] S. Arora and B. Barak, Computational complexity: A modern approach, New York, New York: Cambridge University Press, 2009.

[4] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach (3rd Edition), 3. ed., Upper Saddle River, New Jersey: Prentice Hall, 2010.

[5] M. Mitchell, An Introduction to Genetic Algorithms, 5. ed., Cambridge, Massachusetts: MIT Press, 1999.

[6] P. Moscato, "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms," Pasadena, 1989.

[7] N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association,* pp. 335-341, 9 1949.

[8] B. Brügmann, "Monte Carlo Go," 1993.

[9] A. G. Association, "What is Go?," [Online]. Available: http://www.usgo.org/what-go. [Accessed 23 10 2015].

[10] M. H. M. W. H. J. v. D. H. J. W. H. M. U. B. B. Guillaume M. J-B. Chaslot, "Progressive strategies for Monte-Carlo Tree Search," *New Mathematics and Natural Computation,* pp. 343-357, 03 2008.

[11] M. Research, "Multi-Armed Bandits," Microsoft Corporation, [Online]. Available: http://research.microsoft.com/en-us/projects/bandits/. [Accessed 28 10 2015].

[12] P. Auer, N. Cesa-Bianchi and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning,* pp. 235-256, 05-06 2002.

[13] L. Kocsis and C. Szepesv¶ari, "Bandit based Monte-Carlo Planning," in *Machine Learning: ECML*, Berlin / Heidelberg, 2006.

[14] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, "A Survey of Monte Carlo Tree Search Methods," in *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, 2012.

[15] J. Grefenstette, R. Gopal, B. Rosmaita and D. Van Gucht, "Genetic Algorithms for the Traveling Salesman," in *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, Pittsburgh, 1985.

[16] S. Russel, "Efficient memory-bounded search methods," in *ECAI '92 Proceedings of the 10th European conference on Artificial intelligence*, 1992.

[17] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3. ed., Berlin Heidelberg: Springer-Verlag, 1996.

[18] S. Gelly and D. Silver, "Combining Online and Offline Knowledge in UCT," in *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007)*, Corvallis, 2007.

List of Measurements

List of Figures

List of Tables

Attachments

There is a CD attached to this thesis. It contains:

- The electronic version of this text (PDF file).
- All experiments' output charts (EMF files) and tables (CSV files).
- Source codes of our example implementation and of all experiments.

Implementation

The example implementation, which is created according to this text, is programmed in C# language for .Net Framework 4.5.1. On the CD is situated the whole Visual Studio solution package that contains libraries for EA, MCTS and our method. The solution also contains one executable application "Sandbox". Sandbox can be used for running all the experiments that were performed for this thesis or it can be simply reprogrammed to do more experiments. The Sandbox application also contains the `App.config` file where the output settings can be changed without programming.

Most of the libraries also contains a variety of unit tests.

VS solution structure description

- `EvolutionLibrary`
  - Library for the Evolutionary algorithm framework contains generic interfaces for population, individuals, operators, selection, etc.
  - Also contains base implementation of evolutionary TSP approach.
- `MonteCarloTreeSearchLibrary`
  - Library for the MCTS integration. Contains UCB selector, UCB trees classes and the expansion or initialization logic.
- `SharedFramework`
  - Common library used across the solution.
- `StoredTSPInstances`
  - Repository of the saved TSP instances used in our experiments.
- `TSPGeneticSolution`
  - The implementation of our method – the MCTS-inspired operators. It also contains the gain computation classes, etc.

- `TSPLibrary`
  - Library for generating, saving, loading or working with the TSP instances.
- `Sandbox`
  - Application for experiments and testing. Contains output saving, multiple test scenarios combinating.
  - `StoredMeasuringBudgets`
    - Stored configurations of all the experiments done in this thesis.

Implementation main classes inheritance diagram