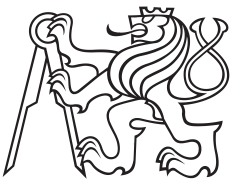
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Solving Scheduling Problems Using Evolutionary Algorithm

David Moidl
Open Informatics, Artificial Intelligence
moidldav@fel.cvut.cz

January 2015
Supervisor: Ing. Jiří Kubalík, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. David Moidl**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Solving Scheduling Problems Using Evolutionary Algorithm**

Guidelines:

1) Study an optimization problem known as the Traveling Tournament Problem (TTP). Focus on metaheuristic approaches, especially the evolutionary-based ones, used for solving the TTP.
2) Propose and implement your own evolutionary-based algorithm for solving the TTP focusing on the representation, constructive procedure, variation operators and evaluation function.
3) Design proof-of-concept experiments and experimentally evaluate a performance of the proposed algorithm on standard test instances.
4) Analyse achieved results and compare the proposed algorithm with other existing (meta)heuristic approaches.

Bibliography/Sources:

1. Luke, S.: Essentials of Metaheuristics. Lulu, 2. vydání, http://cs.gmu.edu/~sean/book/metaheuristics/, 2013.
2. Bong Min Kim. Iterated Local Search for the Traveling Tournament Problem. Master Thesis, Vienna University of Technology, 2012.
3. Uthus, D.C., Riddle, P.J., Guesgen, H.W.: An Ant Colony Optimization Approach to the Traveling Tournament Problem. Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation (GECCO ‘09), pp. 81-88, 2009.
4. A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. Journal of Scheduling 9, 2, pp. 177-193, April 2006.
5. Easton, K., Nemhauser, G., & Trick, M.: The traveling tournament problem description and benchmarks. In Principles and Practice of Constraint Programming—CP 2001, Springer Berlin Heidelberg, pp. 580-584. 2001.

Diploma Thesis Supervisor: Ing. Jiří Kubalík, Ph.D.

Valid until the end of the summer semester of academic year 2015/2016

L.S.

doc. Ing. Filip Železný, Ph.D.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, October 31, 2014

Acknowledgement / Declaration

I would like to thank my supervisor, Ing. Jiří Kubalík, Ph.D., for numerous consultations and useful insights and ideas on the topic.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 5.1.2015

Abstrakt / Abstract

Ačkoli to nemusí být na první pohled zřejmé, rozvrhování sportovních utkání často není vůbec jednoduchá práce. Ve skutečnosti je občas překvapivě těžké vytvořit optimální turnaj i pro hru se zdánlivě jednoduchými pravidly.

Jedním z takto obtížných problémů z domény rozvrhování sportovních utkání je *Traveling Tournament Problem* (TTP). Ten vychází z tvorby turnajů pro baseballovou ligu v USA a je znám pro svou kombinatorickou obtížnost.

V této práci jsme navrhli novou metodu pro řešení TTP založenou na *hybridním genetickém algoritmu* jejíž hlavní komponentou je tzv. *expanzní* operátor. Poté jsme navrhli a provedli řadu výpočetních experimentů, vyhodnotili jsme je a výsledky jsme porovnali s ostatními přístupy nalezenými v literatuře.

Analyzovali jsme získaná data a dospěli jsme k závěru, že klíčový prvek našeho algoritmu, expanzní operátor, funguje velmi dobře. Avšak jeho zakomponování do zbytku algoritmu způsobilo, že tento přestal pracovat optimálně, což mělo značný vliv na celkovou výkonnost.

Nakonec jsme navrhli několik kroků, které bychom mohli realizovat v budoucnu, a o nichž si myslíme, že by pomohly našemu algoritmu dosáhnout výsledků srovnatelných s těmi, které dávají současné state-of-the-art heuristiky.

Překlad titulu: Řešení rozvrhovacího problému pomocí evolučního algoritmu

Even though it might not be apparent, scheduling of various sport tournaments is not at all an easy job. In fact, some games with seemingly simple rules pose surprisingly difficult challenge when the aim is to create an optimal schedule.

One of the very challenging sports-scheduling problems is the *Traveling Tournament Problem* (TTP). It abstracts features of major league baseball in the United States and is known for its high combinatorial complexity.

In this work, we propose a new approach for solving TTP based on *hybrid genetic algorithm* main feature of which is a novel *expansion* operator. Then we conduct series of extensive computational experiments and evaluate their results which we compare to results of other approaches from the literature.

We analyze results of our approach and conclude that the key component—the expansion operator—works very well, but when incorporated into the rest of the algorithm, it causes it to function non-optimally which has noticeable effect on its performance.

Finally, we propose actions to be taken in the future which we believe would help our algorithm to achieve results comparable to those of current state-of-the-art approaches.

Keywords: Traveling Tournament, TTP, Genetic Algorithm, Constraint Satisfaction, CSP, Domains

Contents /

Tables / Figures

Chapter 1
Introduction

1.1 Motivation

For many, "scheduling" is not the first word that comes to mind when you say "sports". It is much more likely that people would think about their favorite football teams or players, yet sports and scheduling undeniably belong together.

There are several reasons why it is so. The most pragmatic one is that companies profiting on today's sport leagues (like organizers, internet broadcasters, television stations and other media) need the "best" schedule (the "hottest" teams playing each other at the most interesting locations) to attract fans and viewers, since that's how these companies make money.

Another reason is that it's actually beneficial to search for optimal schedules as better schedule may lead to noticeable resource savings for some (or possibly all) of the teams involved. A resource might be a fuel required to travel from one place to another, which—when saved—doesn't produce unnecessary pollution. Such a resource might also be a time spent on the road which can be used to do something more productive, like additional training or excersise before a match. But that's only possible if the journey is shorter and therefore closer to being optimal.

Last but not least is the fact that problems originating in sport scheduling often pose quite a difficult challenge when the aim is to solve them to optimality. Interestingly enough, optimization problems derived from sport leagues can be incredibly hard to solve even if the rules of the game are seemingly quite simple.

An excellent example of such a problem is *Traveling Tournament Problem* (TTP) which is an abstraction of major league baseball (MLB) of the United States. The goal is to find the shortest possible *double round-robin tournament* (DRRT) which also satisfies additional TTP constraints. After being defined for the first time in 2001 by Easton, Nemhauser and Trick [1], TTP has proven to be a tough problem and being such a challenge, it attracted number of researchers.

The complexity exhibited by the TTP might be sligtly unexpected. At first glance, it resembles two problems that are known quite well: *Traveling Salesman Problem* (TSP) and sports timetabling. The former is is pretty much a synonym for finding shortest path through predefined set of locations. The latter is implicitly present in TTP through the criteria imposed over resulting tournament. Even though both of these problems (TSP and timetabling) are quite well known and many efficient approaches exist to solve them, when combined together in a way TTP does it, they form a problem much harder to solve. That's probably why the problem has received noticeable amount of attention from researchers who tried to tackle it with different approaches.

At the beginning, Easton, Nemhauser and Trick [1] tried exact methods like *integer linear programing* or *constraint programing*, but soon after it was clear that TTP is too much of a challenge for these methods. Especially when it came to bigger instances. Others therefore used non-exact, typically (meta)heuristic approaches hoping that their

algorithms will be able to navigate efficiently in the vast space of posible solutions of TTP instances and will eventually find the optimum.

To name some of these approaches, Anagnostopoulos et al. [2] used *simmulated annealing* (SA) in 2006 and then Van Hentenryck and Vergados used it again in 2007 [3]. Also Lim et al. used SA (in combination with hill-climbing) to solve the problem. Others [4], [5] have chosen different metaheuristic based on local search - *tabu search*. And of course, there were yet other methods—population based—amongst which we can find a *hyper-heuristics* [6], *ant-colony optimization* [7] or *genetic algorithms* [8].

We think that population-based metaheuristics, in contrast with metaheuristics based on local search, were not exploited enough on the TTP and we want to change that by proposing new *hybrid genetic algorithm* for solving the problem.

1.2 Aims of this thesis

The goals of this thesis are following:

1. Review and compare current state-of-the-art approaches for solving TTP.
2. Propose and implement new GA-based algorithm to solve TTP.
3. Design and conduct a series of computational experiments and use them to assess performance of the aforementioned algorithm.
4. Compare experimental results with results of state-of-the-art approaches and conclude on the effectiveness and overall usability of the GA framework applied to TTP.

1.3 Organization

The rest of this work is organized as follows: chapter 2 presents formal definition of the problem and reviews related work already done on this topic—including some of the state-of-the-art approaches. In chapter 3 we review the GA framework on which our method is based. The proposed algorithm itself is described in detail in chapter 4. Chapter 5 then illustrates our experiments together with their results and compares these results with outcomes of other approaches from the literature. Finally, chapter 6 concludes this work with summary of the most important findings learned through the results of our experiments and provides an outlook on possible future work.

Chapter 2
TTP — Problem definition and related work

To be able to properly define the problem, let us first explain some of the often-used terms from TTP-related terminology and briefly introduce the background of the task. Let us remind that TTP is not an artificially constructed problem, it originates in a real world since it abstracts scheduling of MLB tournaments.

2.1 Used terminology and background of the problem

Consider n teams, n being even. A *round-robin tournament* (RRT) is such a tournament among the teams in which every team plays every other team exactly once. This tournament consists of $n-1$ *rounds* each being composed of $n/2$ *games*. At each game, one of the teams involved is said to play *at home* while the other plays *away*. Obviously, the game is played at the venue of the *home* team while the *away* playing team is the one who has to travel to opponents location.

A *double round-robin tournament* (DRRT) is an extension of simple round-robin tournament consisting of $2n-2$ rounds in which every pair of teams plays exactly twice—once at each teams venue.

Using the terminology defined in [1], when travelling from one place to another, successive games played at home are called a *home stand* and consecutive games played away are called a *round trip*. The number of consecutive games in a home stand/road trip defines its *length*. These metrics are important since TTP imposes some special constraints over them.

Venues of the teams are assumed to be different spatial locations. Distances between these locations are given in a form of symmetric $n \times n$ matrix D. A single element $D_{i,j}$ then corresponds to a distance between locations i and j.

At the very beginning of the tournament, all teams are positioned at their own venues. When playing *away*, the team travels from its last position to the new one. At the very end of the tournament, those teams, which played away in the last round, travel back to their home venues.

2.2 Problem definition

Being familiar with the background and basic terminology of the TTP, we may now properly define the problem.

Let us again have n teams, n even. Let us also have a distance matrix D and non-negative integers u and l. Having this input, we can define the *Traveling Tournament Problem* as a problem of finding a schedule amongst all the teams satisfying following constraints:

1. *Double round-robin* constraint: the schedule has to be a valid DDRT amongst all n teams
2. *noRepeat* constraint: it must hold that no pair of teams plays twice in two consecutive rounds (i.e. there are no teams i and j such that i would play j and then j would play i in the very next round)
3. *atMost* constraint: the length of any home stand and road trip is no less than l and no more than u

The reader may now be slightly confused with the name of the *atMost* constraint. Why name it *atMost* when *both* lower and upper bounds are given? The reason is simple—even though TTP in its general form accepts both lower bound and upper bound on the length of road trips and home stays, in practice it's rarely used. The most common scenario is that l is set to one and u is either a fixed constant or input parameter greater or equal to one. With lower bound being virtually neglected and only the upper bound affecting the problem, many researchers adopted the name *atMost* as it more accurately reflects the nature of the constraint.

Since TTP is an optimization problem, it not only searches for a valid solution, but also for the *optimal* one, hence it tries to minimize the total distance travelled by all teams. To explain how exactly is this distance computed, we first need to describe the representation of the problem.

2.3 Problem representation

It is very common (and this work makes no exception) that TTP schedule gets represented as a table (or a two-dimensional array when it comes to implementation). There are two different ways how to organize things in such a table: either teams are represented by rows and rounds as columns or the other way around. Both possibilities are used in the literature. And even though it's just a matter of perspective (two-dimensional array can be interpreted either way), we like to think about the representation as if teams were represented by rows and rounds by columns. For simplicity, each team is assigned with a number from one to n which is then used as a unique identifier of that team.

Formally, we represent the tournament by M, an $m \times n$ matrix of integers where n is the (even) number of teams and $m = 2 \cdot (n - 1)$ is the minimal number of rounds needed to constitute a valid DRRT. A single element $x = M_{i,j}$ then represents one game played by team i in round j. This value specifies both the opponent of team i and the location at which the game is held. If x is positive, then team i plays team x at home. However, if x is negative, then team i plays with team x at the venue of its opponent.

An example of actual representation of valid TTP solution is given in figure 2.1 which depicts an optimal solution to an instance with six teams. The table therefore has six rows and ten columns.

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	3	1	5	-2	-3	-4

Figure 2.1. A table representing the optimal solution of NL6 instance

Since this way of representing the tournament seems natural and is very straightforward, we are not the first to use it. Hence, our representation is practically the same as in [2] or [9].

Knowing how the problem gets represented, lets now focus on the criterion function of the TTP.

2.4 Criterion function

In section 2.2 we have already mentioned that TTP is an optimization problem and as such it aims at finding an optimal solution amongst all the valid ones. We have also noted that the criterion function, by which the quality of a single solution is measured, is total distance travelled by all teams.

Having described the representation of a single solution (tournament), we can now formally define the criterion function as follows:

Lets have n teams, the $n \times n$ distance matrix D and $n \times (2n-2)$ matrix M representing the tournament. The total distance travelled by all teams according to this schedule is:

$$totalDistance = \sum_{t=1}^{n} \sum_{r=0}^{2n-2} dist(t, r)$$

where

$$dist(t,r) = \begin{cases} 0 & \text{if } r = 0 \text{ and } M_{t,1} > 0 \\ D_{t,|M_{t,1}|} & \text{if } r = 0 \text{ and } M_{t,1} < 0 \\ 0 & \text{if } 1 \leq r < 2n-2 \text{ and } M_{t,r} > 0 \text{ and } M_{t,r+1} > 0 \\ D_{t,|M_{t,r+1}|} & \text{if } 1 \leq r < 2n-2 \text{ and } M_{t,r} > 0 \text{ and } M_{t,r+1} < 0 \\ D_{|M_{t,r}|,|M_{t,r+1}|} & \text{if } 1 \leq r < 2n-2 \text{ and } M_{t,r} < 0 \text{ and } M_{t,r+1} < 0 \\ D_{|M_{t,r}|,t} & \text{if } 1 \leq r < 2n-2 \text{ and } M_{t,r} < 0 \text{ and } M_{t,r+1} > 0 \\ 0 & \text{if } r = 2n-2 \text{ and } M_{t,r} > 0 \\ D_{|M_{t,r}|,t} & \text{if } r = 2n-2 \text{ and } M_{t,r} < 0 \end{cases}$$

The function *dist* describes a distance that has to be travelled by a specific team when moving from one round to the next. A special case is the beginning of the tournament (represented by variable r set to zero) at which all teams start at their home locations. As a consequence, if a team is required to play away in the first round, the distance from its home location to opponents location does count to the total distance of that team. Furthermore, all teams must also return to their home locations after the last round of the tournament (if they are not already there) and distance of that journey also contributes to the resulting distance.

For example, lets examine the total distance traveled by *Team 1* in figure 2.1. The distance would be equal to $0+0+0+D_{1,3}+D_{3,4}+D_{4,6}+D_{6,1}+0+D_{1,2}+D_{2,5}+D_{5,1}$. We can observe that three zeros at the beginning are caused by a home stand spanning first three rounds. The rest of the formula corresponds to *Team 1* traveling back and forth amongst places of its opponents. The very last element $D_{5,1}$ then captures the fact that *Team 1* had to return to its home location after the end of the tournament.

2.5 Other variants of the TTP

The problem we have just defined is the original form of TTP. However, over the years several mutations of this original emerged and we will briefly introduce those of them which are most likely to be encountered in the literature.

2.5.1 Mirrored Traveling Tournament Problem (mTTP)

The mTTP makes just a single, but very significant change to the definition of the problem: the *noRepeat* constraint is replaced by newly introduced *mirror* constraint which is defined as

$$M_{t,r} = -M_{t,r+n-1} \quad \forall t \in \{1, \ldots, n\}, \forall r \in \{1, \ldots, n-1\}.$$

The mirrored double round-robin tournament is then such a tournament where every team has to play every other team in first $n-1$ rounds and in the other $n-1$ rounds, the same games are played but with reversed venues.

As a side note, the mTTP is not just an artificial modification of the original as it reflects the structure of tournaments which is common in some countries.

2.5.2 Relaxed Traveling Tournament Problem

This variant of TTP was defined by Bao and Trick[1]) and relaxes the compactness of the resulting tournament. That is achieved by introducing *byes* to the schedule allowing teams not to play in a round. It is apparent that by allowing a team to skip a round, the tournament may suddenly consist of more than $2n-2$ rounds.

The number of byes per team is controlled by input parameter K. When $K = 0$, the problem degenerates to classic TTP. Byes do not contribute to length of home stays/road trips and are ignored when checking for the *noRepeat* constraint.

2.5.3 Non-Round-Robin Tournament Problem

The *Non-Round-Robin Tournament Problem*[2]) was formulated by Douglas Moody and differs greatly from the original TTP.

The most important difference is the fact that teams do not play DRRT. Instead, so-called "matchup" matrix is given which defines how many times each team has to visit every other team. More specifically, the "matchup" matrix is $n \times n$ matrix of non-negative integer numbers where each element $M_{i,j}, i \neq j$ defines how many times team i has to visit team j.

2.6 Related work

We have said already that TTP has gained quite a lot of attention, which is why numerous papers have been written on the topic and multiple approaches have been proposed. In this section, we review some of them and point out the key features making these approaches efficient or interesting.

[1]) `http://mat.gsia.cmu.edu/TOURN/relaxed/`
[2]) `http://mat.gsia.cmu.edu/TOURN/nonrr/`

2.6.1 Single-solution approaches

Substantial portion of methods in the literature uses so-called *single-solution* approach. Typically, these are based on local search and as such they operate on a single solution (or *configuration*) which is modified via *perturbation* operator and either gets accepted for next iteration or rejected. On following lines we review some of the most important single-solution methods.

In 2006, four researchers—A. Anagnostopoulos, L. Michel, P. Van Hentenryck and Y. Vergados—published a paper [2] in which they proposed algorithm for solving TTP based on *simulated annealing*. Amongst several key features of their algorithm, there are two we want to go into detail with, since these features has proved to be very important and were adopted by many later on.

Hard and soft constraints: they separated the TTP constraints into two groups. *Hard* constraints are those which ensure that produced solution will constitute a valid DRRT. These constraints can't be violated by any candidate solution (configuration) generated by the algorithm. That results in much smaller search space—instead of searching through all possible tournaments, the algorithm is only concerned with those which represent a valid DRRT. On the other hand, produced candidate solutions are free to violate the *soft* constraints. These are the *noRepeat* and *atMost* constraints. Of course, a solution violating any of them would not represent a valid TTP solution, therefore violations of *soft* constraints are penalized. The actual criterion function they used then looks like this:

$$Cost(S) = \begin{cases} length(S) & \text{if } nbv(S) = 0 \\ \sqrt{length(S)^2 + [w \cdot f(nbv(S))]^2} & \text{otherwise} \end{cases},$$

where S is the candidate solution, $length(S)$ is the function computing total distance of the solution (note that we use the very same representation as they did, so this function is exactly the same as the one described in 2.4), $nbv(S)$ stands for the number of violations of soft constraints, w represents adaptive weight of these violations and f is defined as

$$f(x) = 1 + \frac{\sqrt{x} \cdot ln(x)}{2}.$$

The reasoning behind chosen f was to penalize consecutive violations of soft constraints by gradually smaller penalty. For example: one more violation on a schedule which already violates five constraints doesn't make much difference, but one more violation on a schedule not violating any constraints is crucial as it suddenly makes that schedule infeasible.

Local neighborhoods: since SA is a local search-based single-solution method, it needs to have one or more local neighborhoods defined. These are then implemented in the *perturbation* operator and allow the algorithm to move from one configuration to another. Neighborhoods defined by Anagnostopoulos et al. seem to be one of the most important parts of their work as they were used in nearly every other single-solution method that was yet to come. We discuss these neighborhoods in detail in section 4.5 of chapter 4 as we use them as well—in fact, most of the local search component of our algorithm is based on these principles.

One year later, in 2007, Van Hentenryck and Vergados proposed rather unusual approach to the TTP [3]. Observant reader might wonder why we list this paper in section dedicated to single-solution algorithms when it's name is "Population-based simulated annealing for traveling tournaments". The reason is that despite its name the paper still deals with single-solution method, yet it handles it in a special way.

Van Hentenryck and Vergados both cooperated on a paper released one year earlier which we consider one of the most important works in the field of TTP [2]. In 2007, they decided to reuse their work and solve TTP using the same algorithm (called TTSA), but very differently. They introduced a "population" based method which, however, didn't work over population of candidate solutions, but rather over population of single-solution search algorithms.

The idea was to maintain several TTSA instances and view each of them as a black-box. These instances represented the population and were run in completely independent fashion. The execution of all instances in a population was called a *wave*. Once all of them were evaluated (the wave was completed), k best ones "survived" to the next generation simply by not being terminated. The others were restarted from the best solution they have each found during their runtime. This way, the most promising instances were granted the longest running times while others were given another chance by being restarted from the best solution they found.

The algorithm was implemented such that each TTSA instance was run in parallel. And provided with cluster of servers to run at, it was able to find solutions of very high quality which were in several cases better than the best-so-far solutions of that time.

Other single-solution method was proposed by Di Gaspero and Schaerf in 2007 [4]. They designed a *tabu search* algorithm for solving the TTP which uses almost the same basic principles as the one developed by Anagnostopoulos et al.

These common features are:

- search space consisting of valid double round-robin tournaments
- concept of soft constraints—Di Gaspero and Schaerf called them *H1* and *H2* instead of *noRepeat* and *atMost* respectively
- in both papers, the local neighborhoods defined in [2] are used, yet Di Gaspero and Schaerf altered the *PartialSwapTeams* and *PartialSwapRounds* a little

The criterion function, however, is a bit different. In this case, it consists of weighted sum of number of violated constraints *H1* and *H2* and the travelled distance. Weights are then set in such a way that constraint violations are always more significant than the distance itself. Yet these weights are not fixed and get adapted during the search to allow the algorithm to also search the infeasible regions of the search space.

Where both approaches differ greatly is the way of generating initial solutions. While Anagnostopoulos et al. used quite simple recursive procedure, Di Gaspero and Schaerf use more sophisticated strategy. They use so-called *patterns* which are tournaments with unique placeholders (represented by integer numbers) for individual teams. Teams are then randomly paired with placeholders and put at corresponding positions in the tournament.

Patterns are created by solving a problem of finding *1-factorizations* of a complete graph K_n where n is (even) number of teams playing in the tournament. The aim is to partition this graph in $n-1$ sets of $n/2$ edges such that all edges in one set are pairwise not adjacent. Such a set is called *1-factor* and corresponds to one round in

the tournament. The arcs inside the set then each correspond to a single game of two different teams.

The algorithm itself uses a tabu search metaheuristic. That means it maintains so-called *tabu list* which is a collection of already visited states in the search space preventing the algorithm from both getting stuck in local optima and cycling through already visited places. Tabu search therefore needs a notion of *equivalence* between two states in the search space to check whether a state was already visited or not.

To determine number of possible duplicities in the search space, the authors subjected it to an in-depth analysis. They discovered that different local neighborhoods may in some cases lead to the same states and tracked down the moves being responsible for that. They discarded these moves from the algorithm and constructed their search space using altered versions of aforementioned neighborhoods. This way they didn't loose any reachable state but reduced the number of duplicate states in the search space.

Having reduced the search space, the authors ran a series of experiments according to which their algorithm worked quite well. The average results were close to the best known results of that time and shown that tabu search can be an efficient way to solve TTP.

An interesting approach to tackle the TTP was chosen by Lim et al. in 2006 [10] when these researchers designed an algorithm making use of both simulated annealing and hill-climbing. Their strategy was to divide the search space into a *timetable* space and a *team assignment* space. To be able to do so, they defined a concept of *pattern* and used it quite heavily. A pattern is a string of length n (where n is the number of teams in the instance) consisting of letters H and A which stand for "home" and "away" respectively. This pattern then defines where the team *assigned* to it will play at each round. A tournament is represented by a set of n patterns consistent with each other. Consistency in this case means that when put together, these patterns would fit the definition of a round-robin tournament. Initial solutions are then generated by non-trivial three-phase approach which is designed to find just these sets.

Having the notion of pattern, the authors could define a timetable space in which teams were fixedly assigned to individual patterns and the patterns themselves were altered during the search. On the other hand, in the team assignment part of the search space, the timetable was fixed and the assignment of teams to patterns was optimized.

Each part of the search space was then explored by different search algorithm—simulated annealing component searched the timetable space while the team assignment space was explored by a hill-climbing algorithm.

A *controller* unit then acts as a bridge between these two. According to Lim et al., the controller invokes the SA component to create better timetables. These are passed to the hill-climber which searches for better team assignments. Optimized team assignments are then sent back to the SA for further refinement of the timetable and the process continues until termination condition is met.

It is clear from the description of inner workings of both search algorithms that the SA component uses local neighborhoods defined in [2] and therefore operates only on valid DRRTs while the hill-climbing component uses local exchanges of teams and only accepts such that lead to better schedules using the first-improving strategy.

The criterion function they used takes into account both total distance of the tournament and estimate of future costs of current choices. That is implemented via so-called *look-ahead* procedure which is based on beam search.

The conclusion is that Lim, Rodrigues and Zhang developed quite sophisticated and rather efficient method for solving TTP. They combined several search algorithms together in innovative way and created an algorithm capable of finding solutions of high quality.

2.6.2 Population-based approaches

Besides numerous single-solution algorithms designed for TTP, there are few based on general framework of *evolutionary algorithms* (EA). These are population-based metaheuristic approaches working quite differently than their single-solution counterparts. Let us remind that this work approaches the problem with *hybrid genetic algorithm* and other EA-based methods are therefore of high relevance to us.

One such method was proposed in 2006 by Biajoli and Lorena [8] who used *memetic algorithm* (MA) composed of general genetic algorithm and embedded simulated annealing to solve the mirrored version of TTP.

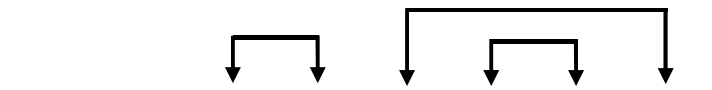
Proposed algorithm is a GA working over population of chromosomes represented by vectors of length n (the number of teams) of integer numbers. Each of these vectors contains a permutation of sequence $[1 \dots n]$. The algorithm then uses so-called *polygon method* to expand the genotype into RRT.

The polygon method works as follows:

1. team at the first position (index 1) of the sequence is called the base team
2. the base team plays the team at index 2
3. team at index $i \in \{3, \dots, (n/2)+1\}$ plays team at index $n - i + 3$
4. once the round is finished, each team at index $i \in \{2 \dots n\}$ is shifted left to index $i - 1$ and team at index 2 is moved to index n, the base team remains at index 1
5. steps 2 to 4 are repeated $n - 1$ times

Lets assume following sequence: `4 3 6 1 5 2`. The application of the polygon method to this sequence is depicted in figure 2.2. As we can see, the result of that process is not valid round-robin tournament as the information about where the games are held is missing. According to Biajoli and Lorena, they use a simple heuristic to complete the tournament, yet no details about the procedure are given.

Note that the algorithm always produces only one half of the solution. The other half is implicitly known due to the mirror constraint.



Round 1	4	3	6	1	5	2
Round 2	4	6	1	5	2	3
Round 3	4	1	5	2	3	6
Round 4	4	5	2	3	6	1
Round 5	4	2	3	6	1	5

Figure 2.2. Illustration of "polygon method" used to generate round-robin tournaments

The genetic algorithm itself works in a standard way. Individuals are selected using unspecified selection operator. Two selected parents are then subjected to so-called *block order* crossover which produces single offspring. This new individual may, with

certain probability, undergo a mutation process implemented as *GameSwap* operator which is slightly modified *SwapPartialTeams* procedure from [2].

After being created (and possibly mutated), each individual is refined using the SA local search algorithm. The authors, however, provided very little information about the local search component of their algorithm. We can only say that the local search component addresses the individual to nearest local optimum using local neighborhoods very similar to those defined in [2].

Overall, the paper doesn't provide much detail about some of the key features of the algorithm. Nevertheless, the results seem to be satisfactory. Unfortunately, the is no way we could directly compare their algorithm to ours since TTP and mTTP are in fact two different problems.

Another approach using EA framework was posted in 2009 by Mısır, Wauters, Verbeeck and Berghe. Their method was, however, very different from ours. Instead of using genetic algorithms, these four researchers decided to utilize *hyper-heuristics* (HH). The concept of HH still belongs to (quite broad) family of evolutionary algorithms, yet it works in a very specific way. Hyper-heuristics do not operate over population of solutions of the actual problem as they rather work with population of so-called *low-level heuristics* (LLHs).

A low-level heuristic is a simple heuristic specifically designed for the problem at hand. The HH algorithm then maintains a set (or a population) of these LLHs and *evolves* a way of combining them together to achieve the best possible solution of underlying problem. The low-level heuristics used in their paper were the local neighborhoods defined in [2] which we describe in detail in section 4.5.

Interestingly enough, the mechanism responsible for selecting appropriate LLHs was not evolutionary algorithm at all. Instead, a technique called *learning automaton* (LA) was employed. According to [11], learning automata are simple learning devices originally designed to study human or animal behavior. The objective of such a device is to learn to select optimal action based on past experience. Internally it is represented by a probability distribution over set of possible actions—LLHs in this case—which determines how likely it is for each action to be selected. This distribution is updated via predefined scheme according to whether or not the selected action succeeds. The rationale behind this system is to increase probability of applying successful actions and decrease probability of unsuccessful ones.

The algorithm itself then consists of single learning automaton and several low-level heuristics. These heuristics are repeatedly selected with respect to the probability incorporated in the automaton and their effect on underlying problem is evaluated. However, it is unclear from the description of the process to what exactly are those LLHs applied. It seems that they alter either one or several TTP tournaments. Once evaluated, the information about the "success" of the LLH is passed to the controlling LA which then updates the probability for that LLH accordingly.

A LLH is deemed successful if it generates a solution which gets accepted. The acceptance criterion embedded in the algorithm always accepts non-worsening solutions, but sometimes it's also able to accept worsening ones. That happens after a predefined number of worsening solutions have been generated and rejected. In such case the algorithm assumes that current solution is not likely to ever get improved and allows worse one to be accepted.

According to authors of the paper, their algorithm is able to produce solutions of high quality which are very close to currently best known solutions. Furthermore, they

claim their method works noticeably faster than some of other approaches from the literature. All this proves that the hyper-heuristics are robust enough to solve complex optimization problems such as TTP.

One more method based on evolutionary algorithms was developed by Uthus, Riddle, and Guesgen in 2009 [7]. They approached the problem using *ant colony optimization* (ACO) technique. ACO is a metaheuristic approach inspired by real ants. These little insects communicate using *pheromone* trails. The more ants go the same way, the stronger the pheromone trail and the more attractive that path becomes to other ants. This behavior is mimicked by the ACO framework where ants are replaced by very simple worker units which communicate with each other using "pheromone matrix".

The paper posted by Uthus et al. requires rather deep knowledge of ACO and other ant-colony-based systems, though. That's why we will describe it very briefly.

In their work, the authors used ants assigned to individual teams. These ants then built the tournament round by round whilst interacting with each other using pheromone matrix. The actual procedure being carried out by the ants used some features from constraint programing (like the teams being viewed as variables with domains filled with potential opponents) together with backjumping mechanism for reverting changes leading to infeasible states. The constraint programing features are interesting for us as we use some of them in our implementation as well, yet after deeper examination of their work, we found out that this is the one and only similarity to our approach.

Above that, Uthus et al. incorporated a concept of home/away patterns similar to those used in [10]. These patterns are then used during constraint propagation. Additionally, the algorithm also contains a local search component which is a simulated annealing equipped with local neighborhoods defined in [2] and which is used to locally optimize generated solutions.

The algorithm is quite complex since it employs several different strategies, which may be why the authors were able to achieve considerably good results. According to their comparison with past ACO approaches, this one has proved to be the best so far.

Chapter 3
Genetic Algorithms

In this chapter, we will provide an insight into the GA framework on which our work is based. We will present and explain basic terminology on which we will rely later on when describing our approach.

Genetic algorithms are members of even wider family of biologically-inspired algorithms called *evolutionary algorithms* amongst which we can find methods like genetic programming, evolution strategies, differential evolution and others. All these methods utilize evolutionary principles we can observe with living things in nature such as reproduction, mutation, selection and recombination.

Above all, there are two main concepts that are common to virtually all evolutionary algorithms: *fitness* and *population*.

3.1 Population and fitness

Unlike conventional optimization algorithms working with only one solution, EAs maintain a *population*—a collection—of possible solutions. Typically, all members of the population are valid solutions to the problem being solved. This makes every EA an *anytime algorithm* able to return valid solution to the problem after being stopped after arbitrary amount of time spent in computation.

As with biological organisms—whose reproduction cycle EAs mimic—all individuals in the population are bearers of *genetic information*.

3.1.1 Genotype

Since all individuals are typically valid solutions of the problem solved by EA, the genetic information contained in each of them encodes a solution to that problem. The actual representation of this genetic information is called *genotype*.

For instance, in so-called *standard GA*, the genotype is an array of bits of fixed length. Such a binary sequence may represent pretty much anything and is therefore general enough to be applicable to a whole lot of various problems. But sometimes it is more convenient to represent the solution in other way. In such case, the genotype will be different, for instance an array of real numbers of variable length. There are no limitations on what the genotype should look like so the most appropriate representation for the problem at hand may be used.

3.1.2 Phenotype

There are usually numerous ways how to encode a solution to a problem. One can use the *standard* way and go for a bit array. Or an arbitrary (and possibly more complex) representation may be chosen. But whichever way the solution is encoded, the genotype always represent the same thing. And that thing is a solution of the problem, or—in EA terminology—a *phenotype*.

The term "phenotype" is used when speaking about the individuals in context of the actual problem. It allows for distinguishment between the actual representation of the solution and the solution itself.

For example, having an individual whose *genotype* is an array of bits of length eight, the *phenotype* may be a set of eight Boolean variables whose values would correspond to zeros and ones in the genotype.

3.1.3 Fitness

Based on the genetic information stored in a form of genotype, different individuals usually represents different solutions to the problem. To be able to determine the *quality* of each of them, a *fitness* is used.

The concept of fitness incorporates the "survival of the fittest" paradigm into evolutionary algorithms. The strengths and weaknesses of an individual originating in its genetic information are combined together and represented as single real number assigned to it by the evaluation function. The fitness specifies how "fit" is that individual for the "environment". In other words, it tells us how good the solution is with respect to the underlying problem.

By convention, higher fitness implies higher quality of the solution and vice versa. However, for minimization problems this notion is sometimes reversed as it is more convenient to work with the fitness "as is".

3.2 General scheme of GA

We have stated before that GAs work with population of individuals. We didn't say, however, what exactly they do with it. In this section, we present and explain inner workings of the most general genetic algorithm.

Genetic algorithms in general try to replicate the process of biological evolution. In nature, the strongest (or most fit) individuals have highest chance to reproduce and produce offsprings. During the process, genetic information is not passed perfectly from parents to their descendants which leads to mutations. The newly created generation of offsprings then replaces (or integrates with) the current population of parents. This very scheme is simulated in genetic algorithms using *genetic operators* which are explained in more detail below.

Figure 3.1 then shows a pseudocode of simple GA using following genetic operators: selection, crossover, mutation and replacement.

3.3 Genetic operators

There are four different types of genetic operators used in GAs. Each of them simulates a part of the evolutionary process as it happens in nature. In most applications, all four types are used, but in some special cases some of them are omitted which often leads to severe changes in behavior of the GA.

3.3.1 Selection

The selection operator replicates the "natural selection" mechanism from biological evolution. It operates over the whole population and *selects* individuals for mating. The selection is customarily fitness-aware and prefers individuals with high fitness over those of low quality. The selection operator is simulating the real-world scenario in which highly adapted individuals are more likely to breed and produce children.

```
1  ALGORITHM general GA
2   Input:  parameters
3   Output: best solution found
4
5  // initialization
6  currentPopulation = generateInitialPopulation()
7  bestIndividual = NULL
8
9  while not terminationCondition.isSatisfied() do
10     // selection
11     parents = select(currentPopulation)
12
13     // crossover
14     offsprings = recombine(parents)
15
16     // mutation
17     mutate(offsprings)
18
19     // replacement
20     currentPopulation = replace(currentPopulation, offsprings)
21
22     // update of best-so-far individual
23     if bestIndividual better than currentPopulation.bestIndividual
24         bestIndividual = currentPopulation.bestIndividual
25     end if
26 end while
27 // return the best-so-far individual
28 return bestIndividual
```

Figure 3.1. Pseudocode of the general GA

In GAs, the two most commonly used selection mechanisms are *tournament* selection and *roulette wheel* selection.

Tournament selection is quite simple and straightforward mechanism. It needs one parameter K which specifies the size of the tournament. The operator then randomly selects K "contestants" from the population and compares their fitness. The one with highest fitness is claimed winner of the tournament and gets selected for breeding.

Figure 3.2 depicts the working of tournament selection over small population of seven individuals with $K = 3$.

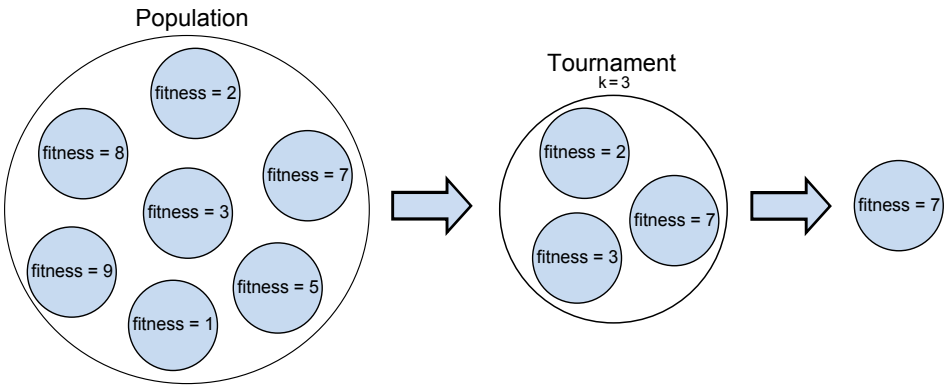


Figure 3.2. Example of tournament selection

Roulette wheel selection is another common selection strategy. With this operator, an individual i will be selected with probability

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j},$$

where f_i is fitness of individual i and n is size of the population. This can be viewed as putting individuals on virtual roulette wheel where each of them would occupy an area proportionate to its fitness.

In figure 3.3 a roulette wheel selection over population of six individuals is depicted. In this example, an individual with fitness 3 and probability of selection $\frac{1}{3}$ is selected.

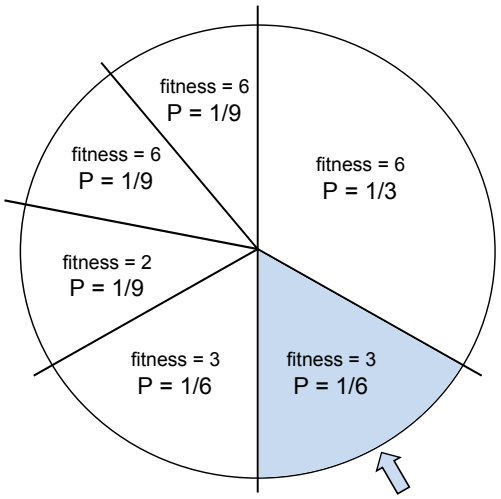


Figure 3.3. Example of roulette wheel selection

3.3.2 Crossover

Once the GA selects individuals from the population, it proceeds to the stage of recombination. This process simulates mating of living organisms and is represented by a *crossover* operator.

The crossover is responsible for generating new individuals out of selected ones. The process works by passing pieces of genetic information (genotype) randomly from parents to newly created children.

Unlike in real world, there are no restrictions on how many "parents" can get involved in the breeding process, but some of the most common crossovers do require two parents from which they generate one or two offsprings. An example of such a procedure is so-called "one-point crossover" (depicted in figure 3.4) which works with fixed-length array genotypes.

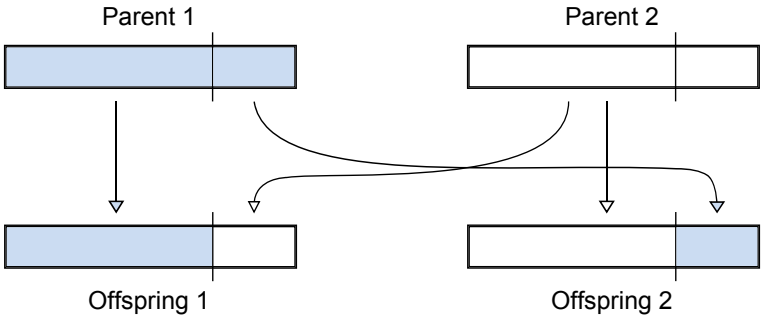


Figure 3.4. Illustration of one-point crossover

3.3.3 Mutation

There is one very significant difference between "crossover" in biological evolution and its EA counterpart. While in nature, the process of passing genetic information from parents to their children is imperfect and *mutations* can happen along the way, in EAs this process is typically flawless and information gets transferred from one individual to another unchanged. To simulate these imperfections, another genetic operator is used: *mutation*.

The mutation operator introduces diversity in the population and sometimes allows the algorithm to reach parts of a search space which otherwise couldn't be visited.

A most common scenario of incorporating a mutation into GA is by defining a *mutation rate*. That is typically a parameter of the search which specifies how likely it is for an individual to get mutated. Small value implies sparse application of the mutation operator which may lead to insufficient diversity in the population, whereas too high value may introduce unnecessary amount of diversity which can prevent the algorithm from converging towards optimum.

3.3.4 Replacement

The last of genetic operators is called *replacement* and its function is to integrate newly created offsprings into the population. The result of this process is also a population which represents new generation of individuals. There are two most common replacement schemes: *generational* and *steady state*.

Generational replacement is very straightforward. As the name suggests, it replaces the whole generation. This scenario mimics the life cycle of short-lived creatures such as some types of insects—the parents mate and die and are immediately replaced by their descendants. This is often implemented by throwing away all individuals in current population and replacing them with newly created ones.

Steady-state replacement, on the other hand, replicates the life cycle of long-lived creatures which produce their offsprings and continue living with them. However, newly created individuals cannot be added to the population just like that since sooner or later the population wouldn't fit into memory. And on top of that, the population is usually fixed in size.

To be able to incorporate newly generated offsprings to the population, the replacement operator first has to make space for them by getting rid of some individuals in the population. Among several ways of how to select which individuals to evict, the following two are the most common:

- completely at random—the usual way as it doesn't decrease diversity of the population, but there is no guarantee that individuals of high quality won't be discarded
- based on fitness—worst individuals are removed from the population which may cause premature convergence

Most replacement implementations can discard individuals of high quality during the process. To prevent this, a concept of *elitism* was introduced.

Elitism is a mechanism that protects high-quality individuals from being disposed by the replacement operator. With elitism enabled, GA keeps track of best individuals in the population and directly injects some of them in new population when it comes to replacement. These individuals are called *elite* individuals and their number is determined by input parameter.

3.4 Memetic algorithms

Besides the standard genetic algorithm we described in previous sections, there are other, more specialized variants, of GAs like *memetic algorithms* (MAs) which are important to us since our implementation utilizes this technique.

3.4.1 General scheme

A memetic algorithm is a genetic algorithm with embedded local-improvement component. Individuals are locally refined to imitate the process of learning that living things exhibit during their lifetime. The idea is that genetic information is just a predisposition that gives an individual its *potential*, but only through learning this potential can be fully unlocked.

In MAs, this is usually done by applying local search on every newly generated individual right after application of all genetic operators. The general scheme of memetic algorithm is presented as a pseudocode in figure 3.5.

```
1  ALGORITHM general GA
2   Input:  parameters
3   Output: best solution found
4
5  // initialization
6  currentPopulation = generateInitialPopulation()
7
8  while not terminationCondition.isSatisfied() do
9      // selection
10     parents = select(currentPopulation)
11
12     // crossover
13     offsprings = recombine(parents)
14
15     // local optimization
16     for offspring in offsprings do
17         locallyOptimize(offspring)
18     end for
19
20     // mutation
21     mutate(offsprings)
22
23     // replacement
24     currentPopulation = replace(currentPopulation, offsprings)
25
26     // update of best-so-far individual
27     if bestIndividual better than currentPopulation.bestIndividual
28         bestIndividual = currentPopulation.bestIndividual
29     end if
30 end while
31 // return the best-so-far individual
32 return bestIndividual
```

Figure 3.5. Pseudocode of general memetic algorithm

When compared to pseudocode of GA (figure 3.1) we notice that MA performs the local optimization as an additional operation on each generated offspring (lines 16 to 18 in figure 3.5).

3.4.2 Local optimization techniques

In theory of memetic algorithms, there are two different approaches when it comes to local optimization.

Lamarckian evolution is based on thoughts of Jean Baptiste Lamarck who defined evolutionary process by which an organism can adapt to environment through learning. Specifically, if an individual learned to use some of its features, that feature would become "stronger" over the lifetime of the individual. In terms of genetic algorithms, this means that learning actually changes the genotype of the individual.

With lamarckian evolution paradigm, the MA applies a local optimization step to an individual and keeps only the refined version in the population.

Baldwin effect, on the other hand, states something little different. An individual is assumed to be able to learn, but learning can only increase increase its chance to reproduce. According to Baldwin, if an individual is capable of learning something that makes it more fit to the environment, it only increases its fitness and the individual has thus higher chance to reproduce, but no physical changes will occur because of learning.

Translated to terms of memetic algorithms, once an individual is locally optimized, its *fitness* is improved to match the result of local optimization procedure, but the genotype remains the same.

Chapter 4
Proposed approach

In this chapter, we will provide detailed explanation of our implementation of GA for solving the TTP. First, we will give detailed description of our representation of the problem including the so-called *expansion operator*. Then we will focus on genetic part of the algorithm and we will finish this chapter with thorough explanation of the local-optimization component.

4.1 Introduction

As we have said several times before, our implementation is based on genetic algorithms, more specifically it utilizes a concept of a memetic algorithm.

As such, it incorporates all the features of GAs (like population and genetic operators) together with local-optimization procedure which is implemented by means of a local search. Figure 4.1 shows the algorithm in the form of a pseudocode.

4.2 Representation

Partially, we have described the representation before in section 2.3 dedicated to general representation of TTP. But since our algorithm is a genetic algorithm, we had to find a way how to encode the solution into genotype of individuals. The most straightforward approach would be to work with population of two-dimensional arrays, each encoding directly one particular tournament. But such a representation is difficult to work with when it comes to genetic operators.

Instead, we used a concept we call *expandable chromosome*. Such a chromosome has two representations—the genotype which encodes the solution to the problem and secondary (*extended* or *expanded*) representation with which we can programmatically operate. This secondary representation is in fact a *phenotype* (as we defined it in section 3.1.2) of the individual. The former can be transformed (or *expanded*) into the latter by so-called *expansion operator* (described in section 4.4) and the genotype therefore defines how the phenotype will be constructed. This double representation is beneficial since the genotype can be designed in such a way that it is easy to work with for genetic operators while the secondary representation (phenotype) can be used to evaluate the individual (and, as in this work, it may be also used for local optimization).

In our case, assuming instance of n teams, the genotype is an array of positive integers of length $n \cdot 2(n-1)$ divided into $2(n-1)$ blocks. Each of these blocks then contains a permutation of set $[1 \dots n]$. An example of a chromosome with this genotype for instance of TTP with four teams is given in figure 4.2.

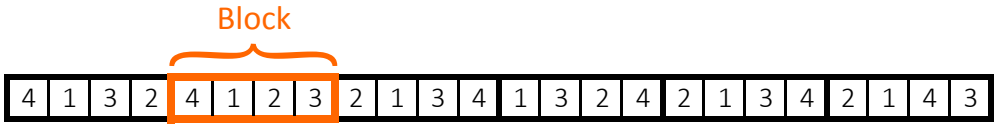


Figure 4.2. Example of genotype of our chromosome

```
ALGORITHM TTPMA
 Input:   parameters
 Output:  best solution found

// initialization
currentPopulation = generateInitialPopulation()

while not terminationCondition.isSatisfied() do
    // selection
    parents = select(currentPopulation)

    // offsprings creation - either via mutation or recombination
    offsprings = []
    if random() < mutationProb
        // creates offspring by mutating parent
        for offspring in offsprings do
           offspring = mutate(parent)
           offsprings.add(offspring)
        end for
    else
        // creates offsprings using crossover
        offsprings = recombine(parents)
    end if

    // local optimization
    for offspring in offsprings do
        locallyOptimize(offspring)
    end for

    // replacement
    currentPopulation = replace(currentPopulation, offsprings)

    // update of best-so-far individual
    if bestIndividual better than currentPopulation.bestIndividual
        bestIndividual = currentPopulation.bestIndividual
    end if
end while
// return the best-so-far individual
return bestIndividual
```

Figure 4.1. Pseudocode of our genetic algorithm

As we said earlier, the genotype serves as an input to constructive procedure which takes the genotype and produces corresponding phenotype. The procedure is called the *expansion* operator and is described in detail in section 4.4.

The secondary representation (phenotype) is then the commonly used table representation of TTP, just like we described it earlier (section 2.3).

4.3 GA component

The memetic algorithm we use doesn't deviate much from the standard scheme of memetic algorithms. The only noticeable difference between pseudocode of our algorithm (fig. 4.1) and pseudocode of the general MA (fig. 3.5) is that we use *either* selection and recombination *or* mutation to generate new individuals.

4.3.1 Initialization

The initial population is generated in a completely random fashion. Each individual is generated by creating $2n - 2$ blocks making up its genotype. Each of these blocks is constructed by randomly shuffling a sequence of numbers $[1 \dots n]$.

4.3.2 Fitness

As we said before, GAs typically work with population of valid solutions of underlying problem. Our algorithm makes no exception. The fitness function is therefore simply the total distance defined in section 2.4 which is computed from the *secondary* representation of a chromosome.

4.3.3 Selection

When it comes to selection, we have implemented both the tournament selection and roulette wheel selection in their basic forms as described in section 3.3.1.

4.3.4 Crossover

There are three different crossover strategies incorporated in our algorithm, all of them being quite standard and well-known. As most of the mutation operators, these crossovers operate over whole blocks within the genotype.

OnePointCrossover is the standard crossover which has been described earlier in section 3.3.2. It takes two parents and produces two offsprings by randomly splitting each parent in two pieces and passing one part of it on either offspring.
TwoPointCrossover works very much alike the *OnePointCrossover* with the only distinction that each of the two parents is split into three parts and each of two offsprings receives two pieces from one parent and one piece from the other.
RandomCrossover takes arbitrary number of parents and produces arbitrary number of offsprings. Every offspring is generated block by block by copying random block from random parent. This operator has the highest exploratory potential since the offsprings are likely to be very different from their parents.

4.3.5 Mutation

The algorithm uses several mutation operators, from very disruptive ones to ones having rather small effect on the genotype.

BlockSwap mutation simply swaps two randomly selected blocks in the genotype. The effect of this operator is depicted in figure 4.3. The magnitude of changes caused by this operator is relatively high as the chromosome is processed from left to right by the expansion operator and even small change in a block positioned early in the chromosome may lead to significant change in produced tournament and its total length.

BlockShuffle mutation shuffles contents of one randomly selected block. The illustration is given in figure 4.4. The strength of this mutation varies. It is capable of doing rather small changes to resulting tournament (for example shuffling the right-most block

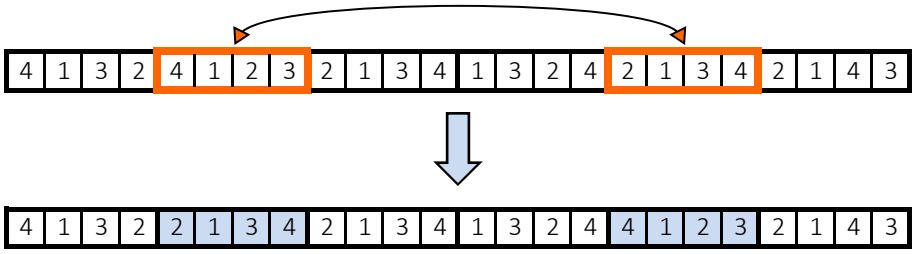


Figure 4.3. Illustration of block-swap mutation

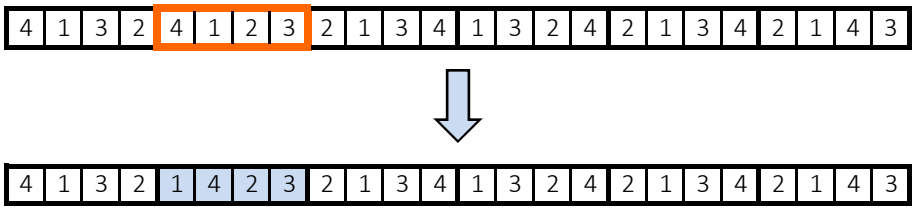


Figure 4.4. Illustration of block-shuffle mutation

can affect only the last round of the tournament) as well as significant ones (shuffling the left-most block can result in completely different schedule).

BlockSequenceSwap mutation randomly selects a block after which it "cuts" the genotype in two pieces. It then swaps these two pieces. This operator is quite disruptive as it affects the whole genotype. The process is depicted in figure 4.5 where the genotype was cut after the fourth block.

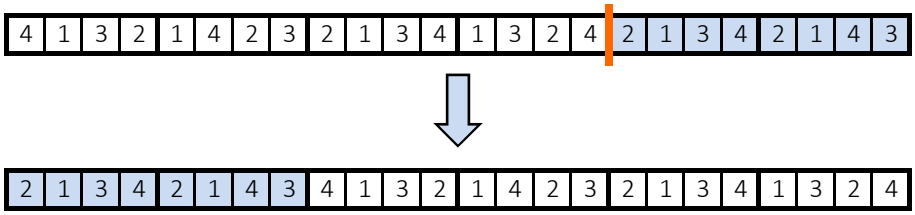


Figure 4.5. Illustration of block-sequence-swap mutation

GuidedBlockSwap mutation works the same as *BlockSwap* mutation with just one difference: with probability increasing over time, generated indices of blocks to be swapped are more and more likely to be close to the end of the genotype.

We rationalized this behavior by assumption originating in how the expansion operator works. We assumed that changes to blocks closer to the end won't have such a disruptive effect as changes to blocks at the beginning. And by increasing the probability of selecting blocks closer to end for swapping, we wanted to allow the algorithm to make significant changes at the beginning of computation and smaller ones later on when solutions of high quality may have already been found and would have been broken by too powerful mutation.

GuidedInBlockSwap mutation works similarly as the *GuidedBlockSwap* operator as it also gradually shifts its attention towards the end of the chromosome over time. But rather than swapping two blocks from the right-hand part of the genotype, it selects just one block (likely from the end of the chromosome) and swaps two values *inside* that block. The process is illustrated by figure 4.6.

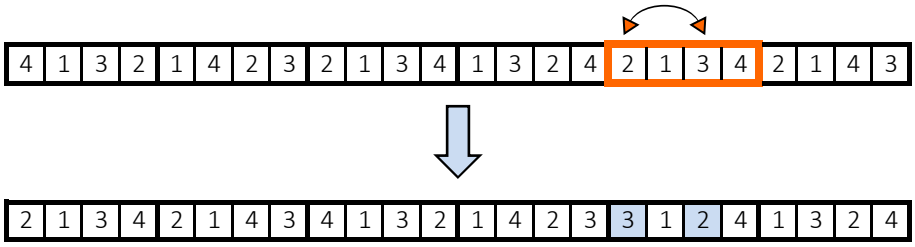


Figure 4.6. Illustration of guided-in-block-swap mutation

Amongst all the mutation operators, this is probably the weakest one. Sometimes, its application may not have any effect at all since swapping two teams which would play each other results in the same two teams playing each other—that is, no change.

4.3.6 Replacement

Our algorithm is equipped with two most common replacement strategies: *generational* and *steady-state* (see section 3.3.4). Both of them incorporate a mechanism for preserving arbitrary number of *elite* individuals.

4.4 Chromosome expansion

Before we advance to the description of local optimization component, we shall shed some light on the process which transforms the genotype of our individuals into their secondary representation. It's necessary since the LS component works exclusively with the secondary representation of the chromosome.

We call the process of transformation from genotype to the secondary representation *expansion* of a chromosome. This *expansion operator*, as we call the actual component doing the job, is a vital part of the algorithm as without it we wouldn't be able to evaluate our chromosomes.

4.4.1 General overview

Our idea was to incorporate part of the optimization logic into the expansion operator which should be able to generate solutions of reasonable quality. These would then serve as well-fit starting points for the local search component giving it higher chance to improve them even further. To achieve that, we proposed a method based partially on nearest-neighbor heuristic and partially on constraint programming. Pseudocode of the process is shown in figure 4.7.

4.4.2 Domains

At the very beginning, each team is assigned with a set of possible opponents—a set of both positive and negative indices of all other teams—which we call a *domain* (line 9 in figure 4.7). Formally, for instance of n teams, a domain of team i is defined as

$$D(i) = \{-n, -n+1, \ldots, n-1, n\} \setminus \{-i, i, 0\}.$$

Note that we exclude zero from the set. That's because team indices start with one and no team therefore has index zero. Reason of that is utterly pragmatic: we need to be able to store any index of a team as either positive or negative value and that would be little difficult if we allowed zero.

The domain is maintained for each team during whole expansion process and plays crucial role as it allows us to construct the tournament without ever having to backtrack.

```
FUNCTION expand
 Input:   chromosome
 Output:  schedule

// prepare result
result = NULL

// create domains of all teams
domains = buildDomains()

// do the transformation
schedule, flag = transform(chromosome)

// check the result
if flag = ASSIGN_LAST
    result = assignLastRound(schedule, domains)
else if flag == USE_SYSTEMATIC_SEARCH
    // try to find valid rest of the solution using systematic search
    restOfSolution = systematicSearch(schedule)
    if restOfSolution != NULL
        result = append(restOfSolution, schedule)
    end if
end if

// if the process was successful
if result != NULL
    result = makeValidTournament(schedule)
end if
// return the result
return result
```

Figure 4.7. Pseudocode of the expansion operator

4.4.3 Building the tournament

Having constructed domains for all teams, the algorithm starts the transformation by calling the `transform` method. Inner workings of this function are given in figure 4.8.

This function builds the tournament round by round. It iterates through all *blocks* in the chromosome and for each of them, it tries to place all the values in currently created round of the tournament.

More specifically, it iterates through values in the block one by one and for each of them, it finds all possible opponents of team whose index matches currently placed value. Let i be the value currently being placed from a block to the schedule and r be index of the round currently under construction. An opponent (a team) is claimed valid if it does not yet play in round r and if i is in its domain. These basic requirements ensure that each team plays exactly once in each round and that no team will play any other team more than twice. Resulting tournament won't thus violate any of the hard constraints originating in the DRRT definition.

Valid opponents are then sorted by increasing distance. The distance represents an increment in total distance which would occur if team i and current opponent would play each other in round r and it is computed using following formula:

$$distance = dist(i, r) + dist(o, r),$$

```
FUNCTION transform
 Assumes: n <- team count; distance matrix
 Input:   chromosome, domains
 Output:  schedule, resultFlag

// prepare the result
schedule = empty (2n - 2) x n table

// build schedule round by round
for roundIndex in [1 ... (2n - 3)]
    // get one block from the chromosome
    block = chromosome.getBlock(roundIndex)

    // try to place each value from the block into the schedule
    for value in block
        // skip teams already playing in this round
        if round.contains(value)
            continue
        end if
        // find nearest opponents ordered by ascending distance
        nearestOpponents = findNearestOpponents(value)
        for opponent in nearestOpponents
            placedValue = value
            // negate currently placed value if needed
            if opponent > 0
                placedValue = -value
            end if
            // check constraints
            feasible = assumeMatchup(placedValue, opponent)
            // if these two teams can play each other
            if feasible
                // make them play
                matchup(round, placedValue, opponent)
                break
            end if
        end for
    end for
    // if we were unable to find opponent for all teams
    if round.length < n
        return schedule, USE_SYSTEMATIC_SEARCH
    end if
    // set this round to the schedule
    schedule[roundIndex] = round
end for
// last round is implicitly known
return schedule, ASSIGN_LAST_ROUND
```

Figure 4.8. Pseudocode of the transformation function inside the expansion operator

where i is index of a team being placed into the schedule, o is index of opponent of team i and $dist$ is the length function defined in section 2.4.

An important observation is that opponents are sought for both i and $-i$. An information about whether the distance applies to positive or negative version of placed value is determined from the opponent itself.

For example, if we are trying to place value 3, we find opponents for value 3 *and* value -3. Opponents of value 3 will be negative values and opponents of value -3 will be positive values to indicate where the game would be played. This is checked on line 25 in figure 4.8.

But before we can place any value in the tournament, we have to perform additional checks. This time, we need to make sure that this move will not have any unpleasant consequences later which may force us to backtrack some changes.

Let us assume that currently placed value is i and nearest valid opponent is o. Then we assume that teams i and o would play each other in round r (lets call it a *matchup*).

We do this by calling the `assumeMatchup` function which simulates the matchup and then observes if it would break anything or not. Note that all changes done to the resulting schedule and/or domains of the teams are temporary and are reverted right before the method returns.

At the very beginning, it places i and o on their respective places in round r. Then it removes o from team's i domain and vice versa. After that, it iterates through all domains and for each of them it performs following check:

Two and one: Let us assume that $D(x)$ is the currently inspected domain and $r = 2n - 4$. If it happens that $D(x)$ looks like this: $D(x) = \{y, -y, z\}$, i.e. it contains two elements that are the same in absolute value and one completely different value, then we must check that domain $D(y)$ does not look like this: $D(y) = \{x, -x, z\}$. If it did, this move is not applicable. The reason is that this situation can only occur exactly four rounds from the end of the tournament and it implies that there exist two teams which need to play each other twice while each of them has to play the same third team once. This setup, however, is invalid as it is impossible to play all these games in just three remaining rounds. Therefore a failure is returned in such case.

If all domains pass this check, the algorithm continues by removing all teams placed in round r from domains of all teams. Then it iterates all domains again to perform tests described below.

Nonempty domains: possibly the most obvious constraint is that the matchup cannot cause any domain of a team, which is not yet placed in round r, to become empty. That would mean that this team has no suitable opponent and the schedule cannot be completed without violating any of the hard constraints. If that happens, the function returns a failure.

Applicable obvious moves: when stripped of teams playing in current round, domains can (and usually do) shrink significantly. In case that a team—let's call it x—is not yet placed in round r but its domain now contains only one element, let's say y, it is inevitable for team x to play team y in this round as there is simply no other option. We call that an *obvious* move and we check that matchup of x and y is valid by recursive call to the `assumeMatchup` function.

If these tests are successful as well, there is one last check we perform. To make sure that it is even possible to complete the schedule with changes we have done so far, we employ a CP solver[1]).

[1]) `http://jacop.osolpro.com/`

CSP feasibility: An integer variable is created for each team in every remaining round and our domains are converted to actual domains of these variables. Then we create constraints defining valid solution which is in our case an `all-different` constraint (to ensure that every team will play exactly once in every round) and `if-then` constraints securing that teams will be matched up in pairs. The solver is then asked to find any solution satisfying these constraints. As far as we can tell, it internally uses depth-first search with pruning which is fast enough for us to call it many times.

Once the `assumeMatchup` function returns, we either perform the matchup or not based on its result. If the move is applicable, it is applied using `matchup` procedure which places team i and its opponent o on proper places of current round and removes i and o from each other's domains.

The last important feature of the `transform` method is shown on lines 39 to 41 in figure 4.8 where it checks whether it was able to place all values from the block. If not, then a flag is returned to the `expand` function indicating that a systematic search needs to be used to complete the tournament.

The systematic search then uses the very same process described in the *CSP feasibility* checking, but instead of searching for *any* solution, it instructs the solver to search for *all* possible solutions and amongst them it selects the one minimizing total distance.

It can happen that the solver is unable to find any valid solution at all. In that case, we simply return nothing as the expanded representation and the individual is discarded by the genetic algorithm. We don't mind this happening since we have experimentally verified that we discard no more than three percent of individuals. That's small enough percentage for us to neglect it as the GA is able to compensate for it by quantity of generated individuals. We do also think that we don't really waste too much time expanding solutions which will get discarded, because if we were to repair them, we would have to invest additional effort into backtracking and constructing portion of the schedule again.

There is also a possibility that the systematic search is not needed at all. That happens if the heuristic `transform` method is able to correctly fill rounds up to index $2n-3$. Then the last round is simply read from the domains in procedure `assignLastRound` and correctly set to the schedule.

4.5 Local optimization component

Integral part of every memetic algorithm is a local search component used to address newly generated individual to (or near to) nearest local optimum.

4.5.1 Overview

Our algorithm, being a memetic one, comes with its own local optimizer which is implemented by local search. This sub-routine takes expanded chromosome as input and returns another chromosome with its expanded representation altered in such a way that it represents more optimal tournament. The pseudocode is given in figure 4.9 and we can easily recognize a random search using first-improving strategy in it.

Let us remind that the LS component works exclusively with the secondary representation of the chromosome (that being a table of teams and rounds) and *only* the secondary representation gets changed during the process, *not* the primary genotype of the chromosome.

```
1  FUNCTION localsearch
2   Input:   chromosome
3   Output:  optimized chromozome
4
5  // initialization
6  best = chromosome
7
8  while not terminationCondition.isSatisfied() do
9      // apply random action
10     altered = applyRandomAction(chromosome)
11
12     // did it improve the chromosome?
13     if altered better than best
14         best = altered
15     end if
16 end while
17 // return the best improved chromosome found
18 return best
```

Figure 4.9. Pseudocode of local search component of our genetic algorithm

4.5.2 Local neighborhoods

The most interesting part of the algorithm is the `applyRandomAction` method which does exactly what one would expect: it randomly generates an *action* and applies it to given candidate solution. There are five different types of actions the method can apply.

These actions are not an invention of our own, they were defined as local neighborhoods by Anagnostopoulos et al. [2] for their simulated annealing and became sort of a standard for almost all single solution approaches. We will briefly describe them to provide an insight on how they actually work.

SwapHomes neighborhood:

Given a valid DRRT and indices of two teams i and j, this move swaps the home/away states of these two teams at rounds where they play each other. In other words, if team i plays at home with team j in round r_l and team j plays at home with team i in round r_k, then application of this move will make team i play *away* with team j in round r_l and team j will also play *away* with i in round r_k. Figure 4.10 illustrates the effect of *SwapHomes* move.

As we can see, the *SwapHomes* move always affects exactly four games in the tournament. Since the number of changes made by this move is very small, it is the "smallest" (or least destructive) local move. It is also clear that *SwapHomes* move can never cause a valid tournament to become invalid.

SwapRounds neighborhood:

Given a valid DRRT and indices of two rounds i and j, the move just swaps these rounds (columns). The effect of *SwapRounds* move is depicted in figure 4.11.

When compared with *SwapHomes*, this move is more disruptive. The number of affected games is $2n$. But despite the fact that this move affects significantly more games than *SwapHomes* move, the resulting schedule will always remain valid, thus no repair action is needed.

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	3	1	5	-2	-3	-4

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	6	2	-1	-5	-6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	-3	1	5	-2	3	-4

Figure 4.10. Swapping home/away states of teams 3 and 6

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	3	1	5	-2	-3	-4

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	4	-4	-6	3	-3	-2	-5
Team 2	-6	-1	-5	6	5	-3	-4	4	1	3
Team 3	-4	5	4	-5	-6	2	-1	1	6	-2
Team 4	3	-6	-3	-1	1	5	2	-2	-5	6
Team 5	-1	-3	2	3	-2	-4	-6	6	4	1
Team 6	2	4	-1	-2	3	1	5	-5	-3	-4

Figure 4.11. Swapping rounds 4 and 8

SwapTeams neighborhood:

Given a valid DRRT and indices of two teams i and j, this move swaps the games of i and j in every round except when these two teams play each other. More specifically, this move makes following changes in each round:

1. opponents of selected teams are swapped
2. selected teams themselves are swapped

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	3	1	5	-2	-3	-4

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	4	6	-3	-2	-6	3	2	-4	-5
Team 2	3	-6	-3	4	1	5	-4	-1	-5	6
Team 3	-2	5	2	1	-6	4	-1	-5	6	-4
Team 4	-6	-1	-5	-2	5	-3	2	6	1	3
Team 5	-1	-3	4	6	-4	-2	-6	3	2	1
Team 6	4	2	-1	-5	3	1	5	-4	-3	-2

Figure 4.12. Swapping teams 2 and 4

Note that the order of steps 1 and 2 doesn't matter, the result will be the same. Figure 4.12 illustrates the process of applying the *SwapTeams* move to one specific configuration.

Apparently, the *SwapTeams* move is quite disruptive. It affects $2n-4$ rounds (which is all except two) and at each of them it alters exactly four games. That makes a total of $4 \cdot (2n-4)$ changed games.

Similarly to both moves mentioned before, the *SwapTeams* move also cannot void the validity of given tournament and every produced configuration is guaranteed to be a valid DRRT.

The three moves described above may seem sufficient as their "strength" varies from very small (*SwapHomes*) to rather high (*SwapTeams*). The authors of [2], however, see this differently. According to them, these moves are not enough to explore the entire search space and two additional moves are needed, namely the *PartialSwapRounds* and *PartialSwapRounds*. As their names suggest, these moves generalize the *SwapRounds* and *SwapTeams* moves providing even wider search space.

PartialSwapRounds neighborhood:

Like with the *SwapRounds*, also the *PartialSwapRounds* move takes valid DRRT and two round indices i and j. But it also needs one additional parameter t which is index of a team whose games are to be swapped. The move then swaps games of team t at rounds i and j.

Note that swapping two games just like that would result in an invalid tournament. This move therefore cannot be applied without employing a correction mechanism afterwards. Fortunately, there is a straightforward deterministic process to determine which games also need to be swapped for the tournament to remain valid. This procedure (which is quite well explained in [9]) checks for teams which are affected by the original swap and swaps their games at incriminated rounds as well.

The result of applying *PartialSwapRounds* move with parameters $i = 2$, $j = 6$ and $t = 4$ and the correction mechanism (in [4] called *repair-chain*) is shown in figure 4.13.

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	3	1	5	-2	-3	-4

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	4	6	-3	-4	-6	3	2	-2	-5
Team 2	-6	6	-5	4	5	-3	-4	-1	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-1	-3	-2	1	5	2	-6	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	-2	-1	-5	3	1	5	4	-3	-4

Figure 4.13. Application of *PartialSwapRounds* move and necessary repair chain

As we can see, the move didn't swap the whole rounds, but just their portions. In some cases, of course, the *PartialSwapRounds* may degenerate to *SwapRounds* move and swap the whole rounds, but the fact that it is able to swap only *some* of the games in both rounds provides this move with variable strength and allows it to reach parts of the search space unreachable by other moves.

PartialSwapTeams neighborhood:

As a generalization of the *SwapTeams* neighborhood, the *PartialSwapTeams* move takes the same parameters the former does (that being a valid DRRT and indices of two teams i and j) and additionally one more: round index r. Then it swaps games of i and j at round r.

Again, this move itself would produce and invalid tournament so the schedule needs to be updated using a repair-chain.

The repair-chain used with *PartialSwapTeams* differs from the one used with previously discussed move—rather than looking for affected teams, this correction mechanism looks for rounds which were affected by the move. At these rounds, standard *SwapTeams* move is applied. Again, there is detailed description of this procedure in [9]. Figure 4.14 captures the result of application of *PartialSwapTeams* move with parameters $i = 3$, $j = 6$ and $r = 1$ and the correction mechanism.

Like with the *PartialSwapRounds* move, also *PartialSwapTeams* may in certain cases affect $2n - 4$ rounds and degenerate to the *SwapTeams* move. However, in cases when that doesn't happen, the *PartialSwapTeams* provides yet another local neighborhood.

4.5.3 Incorporation of local search into the algorithm

The LS component is obviously directly connected to GA unit. Following the standard scheme of memetic algorithms, GA unit calls the LS every time a new individual is created. Once passed to the LS component, individual's *secondary* representation gets refined and its fitness is re-evaluated.

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-3	-4	-6	3	4	-2	-5
Team 2	-6	-1	-5	4	5	-3	-4	6	1	3
Team 3	-4	5	4	1	-6	2	-1	-5	6	-2
Team 4	3	-6	-3	-2	1	5	2	-1	-5	6
Team 5	-1	-3	2	6	-2	-4	-6	3	4	1
Team 6	2	4	-1	-5	3	1	5	-2	-3	-4

Team/Round	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Team 1	5	2	6	-6	-4	-3	3	4	-2	-5
Team 2	-3	-1	-5	4	5	-6	-4	3	1	6
Team 3	2	5	4	-5	-6	1	-1	-2	6	-4
Team 4	6	-6	-3	-2	1	5	2	-1	-5	3
Team 5	-1	-3	2	3	-2	-4	-6	6	4	1
Team 6	-4	4	-1	1	3	2	5	-5	-3	-2

Figure 4.14. Application of *PartialSwapTeams* move and necessary repair chain

Observant reader may have already noticed that interconnection of this kind inevitably predetermines our algorithm (specifically its GA part) to utilize *Baldwin effect* strategy. Indeed, it does.

Since only the secondary representation gets refined during the local optimization, fitness of an individual may improve, but its primary representation (the genotype) stays the same. That means that the "learning" process (simulated by LS component) doesn't make any "physical" changes to the individual, which is exactly how we defined *Baldwin effect* in section 3.4.2.

Even though it might look like that previous paragraphs completely described the involvement of LS into the algorithm, it's not the case. The LS component is used once more in our implementation, even though it's not that obvious. But if we review pseudocode of the expansion operator (fig. 4.7), we will notice a method named `makeValidTournament` (line 27) which has not yet been explained. It is this very method that hides the secondary usage for our LS routine.

As we said earlier, the GA component works with population of valid solutions of TTP (that is DRRTs not violating any hard nor soft constraint). But the expansion operator internally creates tournaments which are free to violate any soft constraint. To bridge this gap, the expansion operator is also equipped with LS instance which it uses to make generated solutions valid with respect to all TTP constraints.

The process is rather simple—whenever an individual is expanded, its brand new (and most likely invalid from the perspective of TTP) secondary representation is passed to this local search procedure which is configured in such a way that it only accepts a move if it makes the tournament violate *less* constraints and stops immediately once it doesn't violate any of them. This way we ensure that generated individuals will always represent a valid solution of the problem.

4.6 Implementation notes

The program was implemented using Java programming language, specifically Java 8. To be able to utilize the multi-core architecture of modern CPUs, we used Java's native `ExecutorService` API to make the implementation internally parallel. This parallelism is implemented in the main loop of GA component and works as follows:

1. several tasks are created in the main thread, each of them being responsible for creation, expansion and evaluation of new individuals
2. these tasks are run in parallel using `ExecutorService`, each producing few individuals (a fraction of newly created generation)
3. produced individuals are collected and replacement is done in the main thread again

The CP solver we used is also implemented in Java, which was one of our requirements, since this way we were able to interact with it "directly" without any overhead caused by translating the calls through APIs to solver written in whatever language.

By describing implementation details of the algorithm, we have completed the explanation of our approach. And knowing how the thing works in theory, we will proceed to the next chapter to see how it works in practice.

Chapter 5
Experiments

To evaluate performance of our algorithm, we have designed and conducted a series of computational experiments.

5.1 Test data

Fortunately, there are numerous test instances for TTP available at Michael Trick's webpage[1]) an they have been there for quite some time now. That is very helpful since Trick's data became sort of a standard when it comes to evaluating TTP-solving algorithms. Thanks to that we didn't have to make up our own test data and it also allows us to compare our results with others using the very same instances.

However, we didn't run our program on all the data available on aforementioned site. Some of the instances are too large and running experiments over them would take too much time. Therefore, we selected a subset of these instances for our experiments. Namely, we picked all of the *NL* instances as these are the most commonly used. Additionally, we selected the *Super* instances since those are also real-world data and are not too large either.

5.2 Choosing parameters

When using genetic algorithms, there are always several parameters one has to set, but it's not always obvious how to do that. Typical questions related to these choices are: "How large should the population be?", "What selection operators should I use?" and "When do I terminate the computation?".

When using memetic algorithms, there are few more one has to answer for himself, like "What local optimization routine to employ?" and "For how long should the local optimizer run on each individual?".

On following lines, we discuss the possibilities and rationalize our choices of values of these parameters.

5.2.1 Population

There are basically two decisions to make about the population: its *size* and whether or not to use *elitism*.

We decided that the population should consist of 100 individuals. The choice was mostly empirical: it cannot be considered too small, yet it's not too large. Note that size of the population has severe impact on runtime of the algorithm as it takes some time to construct, expand and evaluate each individual. By going with 100 individuals, we limited ourselves to not-so-huge population, but we knew the algorithm won't take too much time to run.

As for the elitism, we configured it to preserve ten best individuals using a rule of thumb.

[1]) `http://mat.gsia.cmu.edu/TOURN/`

5.2.2 Selection

We experimentally verified that the algorithm works pretty much the same whether using *tournament* or *roulette-wheel* selection (assuming the tournament size is reasonably small). The choice was made in favor of tournament selection which is configurable and provides variable selection pressure. And since we wanted our algorithm to be more explorative than exploitative, we set the tournament size to three.

5.2.3 Crossover

Once again we performed several experiments and discovered that *RandomCrossover* exhibits the highest explorative potential and causes the algorithm to try many different combinations of genes. Based on that observation, we selected it as our recombination operator and configured it in such a way that it creates two offsprings from two parents.

5.2.4 Mutation

The choice was not easy with mutations. We have implemented five different mutation strategies, but we didn't want to use all of them as we weren't convinced that the ones we considered "weak" would be of any use to the algorithm.

To decide which mutation(s) to use, we have designed a preliminary experiment. For each of the *NL* instances, we generated a test set of 100 individuals. Then, for each mutation, we iterated all individuals in the set for hundred times, copied each of them and applied the mutation to the clone. That makes a 10 000 applications of each mutation.

An important part of the experiment was that we simulated passing generations of the algorithm. When iterating individuals from the test set, we set index of current generation to be ten times the index of current iteration over the set. That made mutations behave like if a thousand generations passed every time.

This was particularly relevant for *GuidedRoundSwap* and *GuidedInRoundSwap* mutations which change their internal probability distributions used for random index selection based on number of generations passed. Other mutations just did not care about this and continued operating normally.

During the test, we measured fitness of the original individual from the set ($f_{original}$) and fitness of its clone after application of the mutation ($f_{mutated}$). Then we computed an *improvement* as

$$improvement = \frac{f_{original} - f_{mutated}}{f_{original}}.$$

Note that the improvement can be negative which indicates that the original chromosome was worsened by the mutation.

From collected data, we were able to compute following metrics for every mutation:

- ratio of improving applications to total number of applications
- ratio of worsening applications to total number of applications
- average improvement
- average improvement over improving mutations

We then plotted these metrics and used these plots to decide what mutation(s) to use for the actual experiments. Results of this preliminary experiment are given in appendix A.

We soon realized, that we cannot jump to any conclusions based only on average improvement, since the average improvement was usually so small that we had to multiply it by 100 to make it reasonably visible in the plots. That's why we decided to ignore the average improvement and look on other metrics.

We figured that we are not interested in mutations that do nothing and ruled out the *GuidedInBlockSwap* mutation as it had virtually no effect in almost 80% of applications on some instances. But the remaining four mutation operators were difficult to compare. After putting some though into it, we decided to also discard the *BlockSequnceSwap* mutation as it seems too disruptive. It didn't yield bad results in this test, but the algorithm was already set to high exploration by combination of other parameters and we didn't want it to work in a completely random fashion.

This way, we were left with *BlockSwap*, *BlockShuffle* and *GuidedBlockSwap* mutations which we incorporated into the algorithm in such a way that whenever a mutation was needed, one of these was randomly picked (with uniform probability) and used.

5.2.5 Replacement

With replacement, the choice was somewhat obvious. We knew that for larger instances the search space of TTP becomes really huge and we needed the algorithm to explore as much of it as it could. Apparently, *generational* replacement is much more fit for this task than *steady-state* strategy. All the more so as chromosomes of high quality are preserved via elitism in our configuration. There was simply no need to keep any more individuals besides the elite ones when transiting from one generation to the next.

5.2.6 Local search

The only thing we needed to figure out for the local search component was when to stop it. We decided to let it apply 5 000 random actions and then return the best secondary representation it found.

Rationale behind this decision was that the local search component is not powerful enough to improve the solution rapidly in short amount of time. We deduced this from results of single-solution approaches we reviewed according to which it was quite common to let the algorithm run for relatively long time (e.g. several days) before it was able to find high-quality solution. We therefore wanted the GA component to be responsible for searching the space of solutions and limited the LS unit to rather short runtime to provide nothing but an option to slightly increase the quality of individuals generated by GA part of the algorithm.

5.3 Experimental setup

Once we decided how to set up the algorithm for our tests, we could proceed to the actual experiments.

5.3.1 Structure of our experiments

The tests were designed as follows: for each test instance we ran the algorithm with the same parameters (shown in table 5.1). The algorithm was terminated after either running through one thousand generations or reaching a solution of predefined fitness.

The threshold was set as the highest lower bound currently known for corresponding instance (this information is also available at Trick's webpage). This proved to be useful

Component	Parameter group	Parameter	Value
GA	population	population size	100
		elitism	10
	selection	selection type	tournament
		tournament size	3
	crossover	crossover type	random
		required parents	2
		produced offsprings	2
	mutation	mutation types	BlockSwap, BlockShuffle, GuidedInBlockSwap
		probability of selecting	1/3 (uniform)
	termination	number of generations	1000
		reached fitness	highest lower bound for current instance
LS	termination	number of random actions	5000

Table 5.1. Parameters chosen for running the experiments

especially on smaller instances when the algorithm is able to find optimal solution in just few generations and there is then no point in running a thousand of them.

Each instance was evaluated ten times so that we could collect enough data for averaging the results. For larger instances where the algorithm actually reached the limit on number of generations, it may have spent up to 10 000 generations in total on a single instance.

5.3.2 Experimental environment

The experiments were run on a machine equipped with Intel Core i7 CPU and 8 GB of main memory. Our algorithm utilized all eight cores of the CPU by running in eight threads.

The machine was running Windows 7 operating system and result of `java -version` command issued before the tests was following:

```
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

5.4 Results and discussion

We have conducted numerous experiments in compliance with setup and parameters described in previous sections. On following lines, we will present the results for each "family" of instances separately and compare them with results of other approaches from the literature.

5.4.1 Observed metrics

There are not many metrics one needs to measure to be able to evaluate results of TTP-solving algorithm. In fact, there are only two: total distance and time.

Total distance is obviously the only measure needed for assessing the quality of found solutions. In TTP, the distance is an abstract value without any specific unit and we therefore present it as a simple number in the results.

The other observed metric—time—is then used to evaluate efficiency of the algorithm. Apparently, the faster the algorithm is able to find a solution of high quality, the better. But let us remind that genetic algorithms are usually (noticeably) slower than their single-solution counterparts because of their robustness originating in maintaining a

population of solutions which causes additional overhead and results in longer runtimes. All time values are presented in seconds in tables below.

For both the total distance and time required to find the best-of-run solution, we computed the minimal (`Min`), average (`Avg`) and maximal (`Max`) value together with standard deviation (`Std. Dev`) over all ten runs of the algorithm over each instance.

When comparing our results with results of some other approach, we computed a difference between *average* total distance of our method and the other approach (`Diff`).

Additionally, to be able to monitor the behavior of our algorithm, we also recorded the worst, average and median fitness together with number of individuals with unique fitness over all individuals in each population. These values should help us understand why the algorithm works the way it does.

5.4.2 Results for NL instances

As we mentioned before, the *NL* instance family is possibly the most commonly used test set for the TTP and most papers in the literature report their results on this set of instances.

We have selected four approaches to compare our results with, two of which deal with single-solution methods while the other two represent evolutionary approaches. The single-solution techniques are *simulated annealing (TTSA)* [2] and *tabu search (CNTS)* [4], whereas evolutionary methods are *ant colony optimization (AFC-TTP)* [7] and *hyper-heuristics (LHH)* [6].

Let us first present results of our algorithm. Overview of results we achieved on the *NL* instance family is given in table 5.2.

	Total distance				Time			
Instance	Min	Avg	Max	Std. Dev	Min	Avg	Max	Std. Dev
NL4	8 276	8 276.0	8 276	0	0	0.3	1	0.180
NL6	23 916	23 916.0	23 916	0	1	16.8	45	13.224
NL8	39 721	39 721.0	39 721	0	331	778.7	1 374	385.638
NL10	65 560	66 069.0	66 503	288.769	240	1 466.3	2 604	855.940
NL12	127 266	128 842.3	129 860	766.074	1 023	3 060.4	5 071	1 542.298
NL14	227 921	230 984.1	232 416	1 528.669	380	7 361.0	11 480	3 985.590
NL16	322 876	326 704.8	329 814	2 458.088	5 149	11 740.9	21 132	6 262.396

Table 5.2. Results for the *NL* family of instances

Comparison of our algorithm (*TTPMA*) with *TTSA*, *CNTS*, *AFC-TTP* and *LHH* is then given in tables 5.3, 5.4, 5.5 and 5.6 respectively.

Note that for these comparisons, we omitted instances of less than ten teams. The reason is that on these small instances, virtually every approach was able to find optimal solution every time (including ours), hence the comparison would be meaningless.

	TTPMA				TTSA				
Instance	Min	Avg	Std. Dev	Avg. Time	Min	Avg	Std. Dev	Avg Time	Diff
NL10	65 560	66 069.0	288.769	1 466.3	59 583	59 606.0	53.36	40 268.6	9.782%
NL12	127 266	128 842.3	766.074	3 060.4	112 800	113 853.0	467.91	68 505.3	11.634%
NL14	227 921	230 984.1	1 528.669	7 361.0	190 368	192 931.9	1 188.08	233 578.4	16.474%
NL16	322 876	326 704.8	2 458.088	11 740.9	267 194	275 015.9	2 488.02	192 086.6	15.821%

Table 5.3. Comparison of our approach with TTSA on *NL* instances

	TTPMA				CNTS				
Instance	Min	Avg	Std. Dev	Avg. Time	Min	Avg	Std. Dev	Avg Time	Diff
NL10	65 560	66 069.0	288.769	1 466.3	59 876	60 424.2	823.90	7 056.7	8.544%
NL12	127 266	128 842.3	766.074	3 060.4	113 729	114 880.6	948.20	10 877.3	10.836%
NL14	227 921	230 984.1	1 528.669	7 361.0	194 807	197 284.2	2 698.50	29 635.5	14.590%
NL16	322 876	326 704.8	2 458.088	11 740.9	275 296	279 465.8	3 242.40	51 022.4	14.459%

Table 5.4. Comparison of our approach with CNTS on *NL* instances

	TTPMA				AFC-TTP				
Instance	Min	Avg	Std. Dev	Avg. Time	Min	Avg	Std. Dev	Avg Time	Diff
NL10	65 560	66 069.0	288.769	1 466.3	59 634	59 928.3	155.47	4 969.5	9.294%
NL12	127 266	128 842.3	766.074	3 060.4	112 521	114 437.4	895.70	7 660.1	11.180%
NL14	227 921	230 984.1	1 528.669	7 361.0	196 849	198 950.5	1 294.43	20 870.1	13.868%
NL16	322 876	326 704.8	2 458.088	11 740.9	278 456	285 529.6	3 398.57	35 931.3	12.603%

Table 5.5. Comparison of our approach with ACF on *NL* instances

	TTPMA				LHH				
Instance	Min	Avg	Std. Dev	Avg. Time	Min	Avg	Std. Dev	Avg Time	Diff
NL10	65 560	66 069.0	288.769	1 466.3	59 583	60 046.0	335.00	3 600.0	9.116%
NL12	127 266	128 842.3	766.074	3 060.4	112 873	115 828.0	1 313.00	3 600.0	10.101%
NL14	227 921	230 984.1	1 528.669	7 361.0	196 058	201 256.0	2 779.00	3 600.0	12.870%
NL16	322 876	326 704.8	2 458.088	11 740.9	279 330	288 113.0	4 267.00	3 600.0	11.812%

Table 5.6. Comparison of our approach with LHH on *NL* instances

5.4.3 Results for Super instances

Unlike with the *NL* instances, the *Super* instance family is used quite rarely in the literature. There are only two papers that we know of which state detailed results on this instance family: [6] (LHH) and [9] (TTILS$_{opt}$). We therefore compare our results to results of these two approaches.

The overview of results achieved by our algorithm on each of the *Super* instances is given in table 5.7.

	Total distance				Time			
Instance	Min	Avg	Max	Std. Dev	Min	Avg	Max	Std. Dev
Super4	63 405	63 405.0	60 405	0	0	0.3	1	0.160
Super6	130 365	130 365.0	130 365	0	3	21.7	92	29.120
Super8	182 409	182 441.1	182 594	69	441	1 311.2	1 917	479.194
Super10	330 183	334 460.6	337 996	2 130.892	378	1 868.2	2 949	982.235
Super12	498 747	501 755.8	504 782	2 506.531	1 413	3 529.3	6 395	1 827.260
Super14	684 415	693 386.0	697 739	6 057.813	10 376	12 011.1	13 500	1 310.908

Table 5.7. Results for the *Super* family of instances

Comparison of our results with those of LHH and TTILS$_{opt}$ is then presented in tables 5.8 and 5.9 respectively.

Notice that this time we listed all instances in the comparison. The reason is that it no longer holds that every approach solves the small ones to optimality every time as LHH reached sub-optimal solutions even on the smallest instance of just four teams. But on the other hand, the time required to find them was extremely short.

	TTPMA				LHH				
Instance	Min	Avg	Std. Dev	Avg. Time	Min	Avg	Std. Dev	Avg Time	Diff
Super4	63 405	63 405.0	0	0.3	63 405	71 033.0	12 283.00	0	-12.031%
Super6	130 365	130 365.0	0	21.7	130 365	130 365.0	0	1 800.0	0%
Super8	182 409	182 441.1	68.651	1 311.2	182 409	182 975.0	558.00	3 600.0	-0.293%
Super10	330 183	334 460.6	2 130.892	1 868.2	318 421	327 152.0	6 295.00	3 600.0	2.185%
Super12	498 747	501 755.8	2 506.531	3 529.3	467 267	475 899.0	5 626.00	3 600.0	5.153%
Super14	684 415	693 386.0	6 057.813	12 011.1	599 296	634 535.0	13 963.00	4 700.0	8.487%

Table 5.8. Comparison of our approach with LHH on *Super* instances

	TTPMA				$TTILS_{opt}$				
Instance	Min	Avg	Std. Dev	Avg. Time	Min	Avg	Std. Dev	Avg Time	Diff
Super4	63 405	63 405.0	0	0.3	?	?	?	?	?
Super6	130 365	130 365.0	0	21.7	130 365	130 365.0	0	0.1	0%
Super8	182 409	182 441.1	68.651	1 311.2	182 409	182 409.0	0	77.3	0.018%
Super10	330 183	334 460.6	2 130.892	1 868.2	318 007	318 225.5	262.97	2 313.6	4.854%
Super12	498 747	501 755.8	2 506.531	3 529.3	469 290	472 002.1	1 228.14	2 514.4	5.930%
Super14	684 415	693 386.0	6 057.813	12 011.1	594 388	600 533.6	5 187.21	1 911.6	13.391%

Table 5.9. Comparison of our approach with $TTILS_{opt}$ on *Super* instances

5.4.4 Discussion

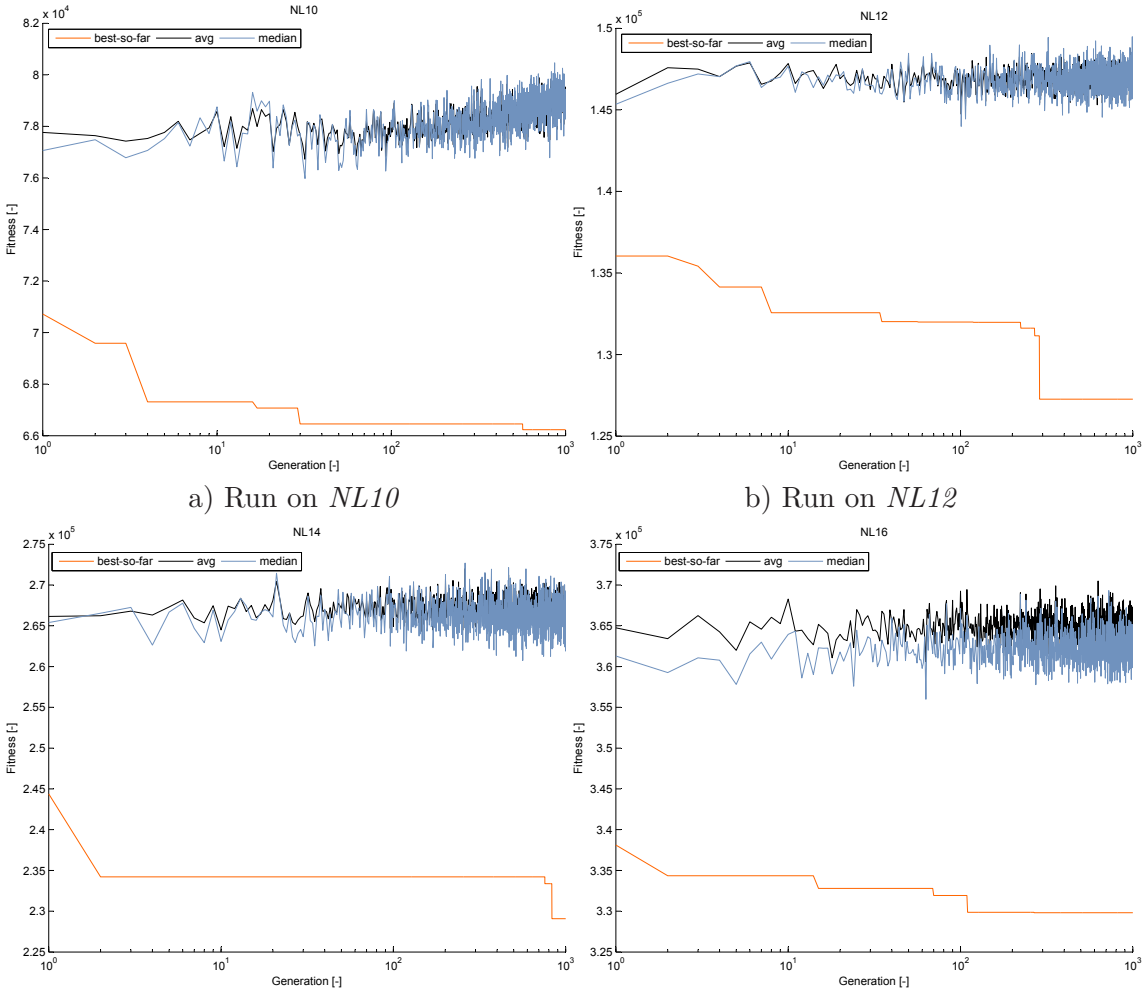
One thing is clear from results presented in previous sections: our approach does not outperform any of the approaches we compared it to. Let us now present our hypothesis of why it is so.

As we said earlier, we measured few additional metrics besides the best-of-run fitness and time. Namely the average, median and worst fitness in the population. After we analyzed this data, we came to single conclusion: the algorithm as a whole behaved very much like a random search.

We derived this conclusion from the way the algorithm worked. More specifically, we observed how often the best-so-far fitness decreased and what were the values of mean and median fitness throughout each run. What we saw was that at the very beginning of each run, the best-so-far fitness decreased every couple of generations. That's good, that is how a genetic algorithm should behave. But soon after, the algorithm pretty much stopped improving the best-so-far fitness and it remained the same for many generations.

Interestingly enough, we cannot say it *converged* to that fitness as both mean and median fitness were quite far from the best-so-far value and the population usually consisted of individuals which had completely unique fitness values. Additionally, we noticed the mean (as well as median) fitness practically didn't change with passing generations. That indicates that the population was diverse enough, but it was not converging to the best-so-far fitness. Figure 5.1 illustrates this problem. It shows four plots, each one depicting a progression of mean, median and best-so-far fitness during one run of the algorithm over *NL* instances of 10 and more teams. Please note that the x axis in all of the plots is in logarithmic scale.

For illustration purposes, we just picked one from 10 runs over each instance to make these plots, but the trend was the same in all runs over all instances: the best-so-far fitness decreased over time, exactly as it should, but the average and median fitness did not. Instead, they oscillated around some specific value and remained more or less the same during whole runtime of the algorithm.



a) Run on *NL10* b) Run on *NL12*

c) Run on *NL14* d) Run on *NL16*

Figure 5.1. Progression of average, median and best-so-far fitness on instances *NL10*, *NL12*, *NL14* and *NL16*

That clearly indicates that the algorithm is unable to gradually shift the population towards more promising regions. If it was able to do so, the mean (and median) fitness would decrease over time as well.

We think that there is only one possible explanation for this behavior: the expansion operator is *too* complex. By complex we mean that it is difficult to say how the secondary representation will look like just by looking at the genotype. In case of our expansion operator, it is not, in fact, easy. There are distances between teams involved and portions of the secondary representation can be even constructed using CSP solver... and that's the problem.

If the interconnection between genotype and the secondary representation is intricate, the GA component of the algorithm has hard time identifying promising parts of genotypes in individuals. In fact, it might not be able to do so at all. In that case, it mutates and recombines individuals "blindly" which results in random-like search. And that is exactly what we observed.

This is caused by the fact that in our case, even slight change in genotype can have (and usually has) noticeable effect on produced phenotype. That prevents crossover operator from effectively combining parts of genotypes of two (or more) parents and it

also prohibits mutation from searching local neighborhood of an individual via small changes in genotype.

To support this conclusion, we conducted numerous additional short-termed tests with various input parameters to verify that our results were not caused by poorly set parameters. These experiments shown that our original experimental setup was not the cause of this random-like behavior as the algorithm exhibited the very same symptoms regardless of combination of input parameters. It didn't work exactly the same every time, of course, but it did always work according to the same general scheme: it decreased the best-so-far fitness several times in the first couple of iterations, but sooner or later it was not able to improve it anymore while the mean fitness was not gradually getting lower.

We are certain that this problem degraded the overall performance of our algorithm severely. And we also think that because of this, the algorithm, in its current state, would not be able to reach noticeably better solutions even if provided with much longer runtime.

We suspect that the shape of search space is rather complicated and according to our observations, it consists of feasible regions separated by infeasible areas. This would explain why the algorithm was able to decrease the best-so-far fitness several times in the beginning: it just (almost randomly) explored the feasible part of the search space it currently operated on, but once it explored it thoroughly, it was unable to cross any of the infeasible regions to move further and explore other parts of the search space. That again was caused by its random-like nature.

On the other hand, we must say that even though the expansion operator prevented the GA unit of our algorithm from performing optimally, the operator itself worked quite well. It was able to create solutions of reasonable quality from almost random genotypes. The difference from state-of-the-art heuristics was consistently between 10% and 15%. That is, in fact, not a bad result at all, as we can safely assume that the local optimizer was not able to improve solutions generated by the expansion operator more than slightly and most of the credit therefore goes to the expansion operator. All the more so as the state-of-the-art approaches usually utilize extensive local search while working with direct representation of the tournament which makes it substantially easier for them to navigate through the search space.

Chapter 6
Conclusion and future work

This work aimed at proposing novel approach for solving the challenging *Traveling Tournament Problem*.

We first reviewed current state-of-the art heuristics and discovered that most of them are methods based on local search. We also reviewed few approaches based on *evolutionary algorithms*.

Then, we proposed, implemented and evaluated a brand new metaheuristic approach for solving TTP based on *memetic algorithm*.

The most interesting and important part of our algorithm is so-called *expansion* operator which is a routine that transforms simplified representation (used by *genetic algorithm* inside our method) to fully developed, valid TTP solutions. The process is based on simple nearest-neighbor heuristic and uses some features of *constraint programming*. In some cases it even employs a CP solver to generate portions of these solutions.

We designed and conducted series of computational experiments and analyzed their results. We concluded that the expansion operator itself works very well and is able to generate TTP solutions of reasonable quality. On the other hand, the expansion process seems to be too complicated for the genetic algorithm to work efficiently.

We discovered that due to intricate interconnection between genotype and transformed representation of individuals, the GA part of our algorithm is incapable of identifying promising portions of genotypes in these individuals and that, in turn, makes it behave very much like a random search. That severely impacts the overall performance of the algorithm. But even with GA component working non-optimally, the algorithm was able to find solutions only 10% to 15% longer than those found by current state-of-the-art approaches.

Despite the fact that this algorithm didn't outperform current state-of-the-art heuristics, we still think that genetic algorithms are more than capable of solving TTP efficiently. We thing that with some improvements, it might as well be our algorithm which would achieve great results on TTP.

But before that happens, there are few improvements to be done which plan for future work:

- define different primary representation and/or simpler expansion operator
- limit usage of the expansion operator proposed in this work only to generating initial population of individuals of high quality
- explore possibility of working over individuals which would *not* represent strictly valid TTP solutions (e.g. tournaments violating *noRepeat* and *atMost* constraints) and examine suitable evaluation functions to be used to assign fitness to such individuals
- introduce more sophisticated local optimizer to improve generated individuals even further

References

[1] Kelly Easton, George Nemhauser, and Michael Trick. The Traveling Tournament Problem Description and Benchmarks. 580.

[2] A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. *Journal of Scheduling*. 2006, vol. 9 (issue 2), 177-193.

[3] Pascal Van Hentenryck, and Yannis Vergados. *Population-based simulated annealing for traveling tournaments*. In: *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*. 2007. 267.

[4] Luca Di Gaspero, and Andrea Schaerf. A composite-neighborhood tabu search approach to the traveling tournament problem. *Journal of Heuristics*. 2007-2-28, vol. 13 (issue 2), 189-207.

[5] Jin Ho Lee, Young Hoon Lee, and Yun Ho Lee. *Mathematical modeling and tabu search heuristic for the traveling tournament problem*. 2006.

[6] Mustafa Mısır, Tony Wauters, Katja Verbeeck, and Greet Vanden Berghe. *A new learning hyper-heuristic for the traveling tournament problem*. In: *Proceedings of the 8th Metaheuristic International Conference (MIC'09). Hamburg: Germany*. 2009.

[7] David C Uthus, Patricia J Riddle, and Hans W Guesgen. *An ant colony optimization approach to the traveling tournament problem*. In: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 2009. 81–88.

[8] Fabrício Lacerda Biajoli, and Luiz Antonio Nogueira Lorena. *Mirrored traveling tournament problem: an evolutionary approach*. 2006.

[9] Bong Min Kim. *Iterated local search for the traveling tournament problem*. Citeseer, 2012.

[10] A. Lim, B. Rodrigues, and X. Zhang. A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operational Research*. 2006, vol. 174 (issue 3), 1459-1478.

[11] KS Narendra, and MAL Thathachar. Learning automata: an introduction. 1989. *Printice-Hall, New York*.

Appendix A
Experiment with mutations

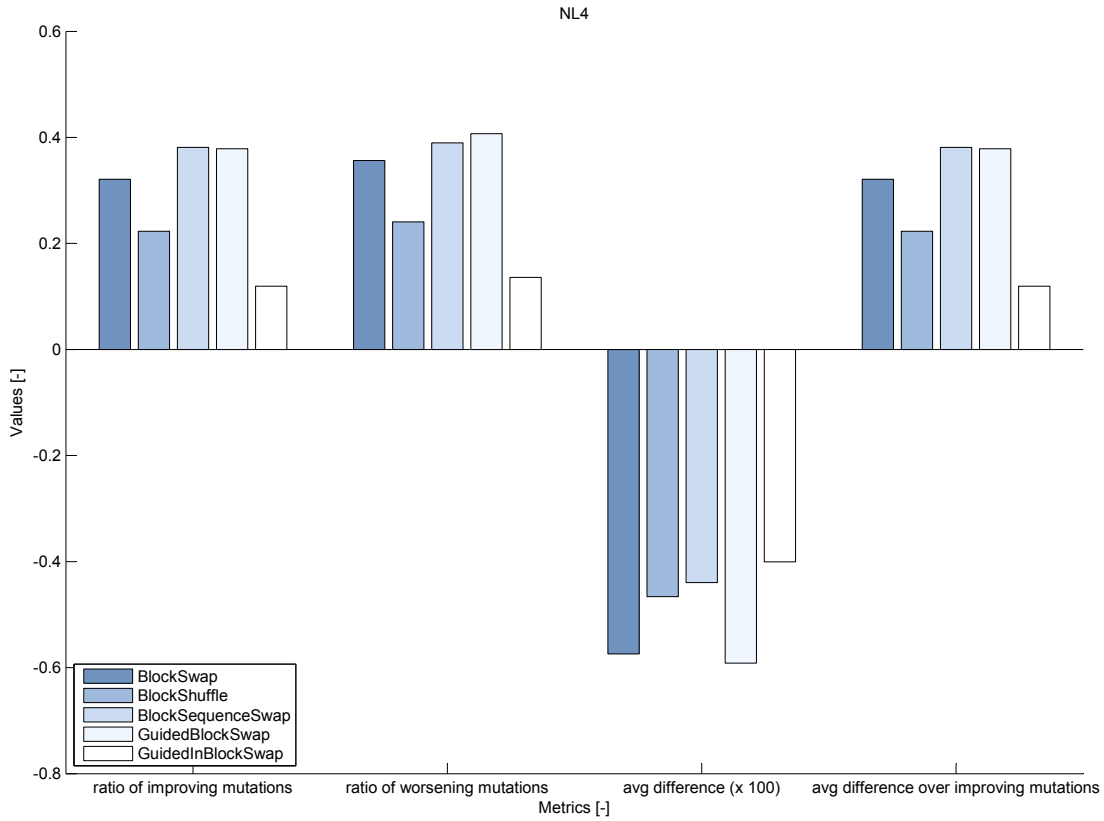


Figure A.1. Result of experiment with mutations for instance NL4

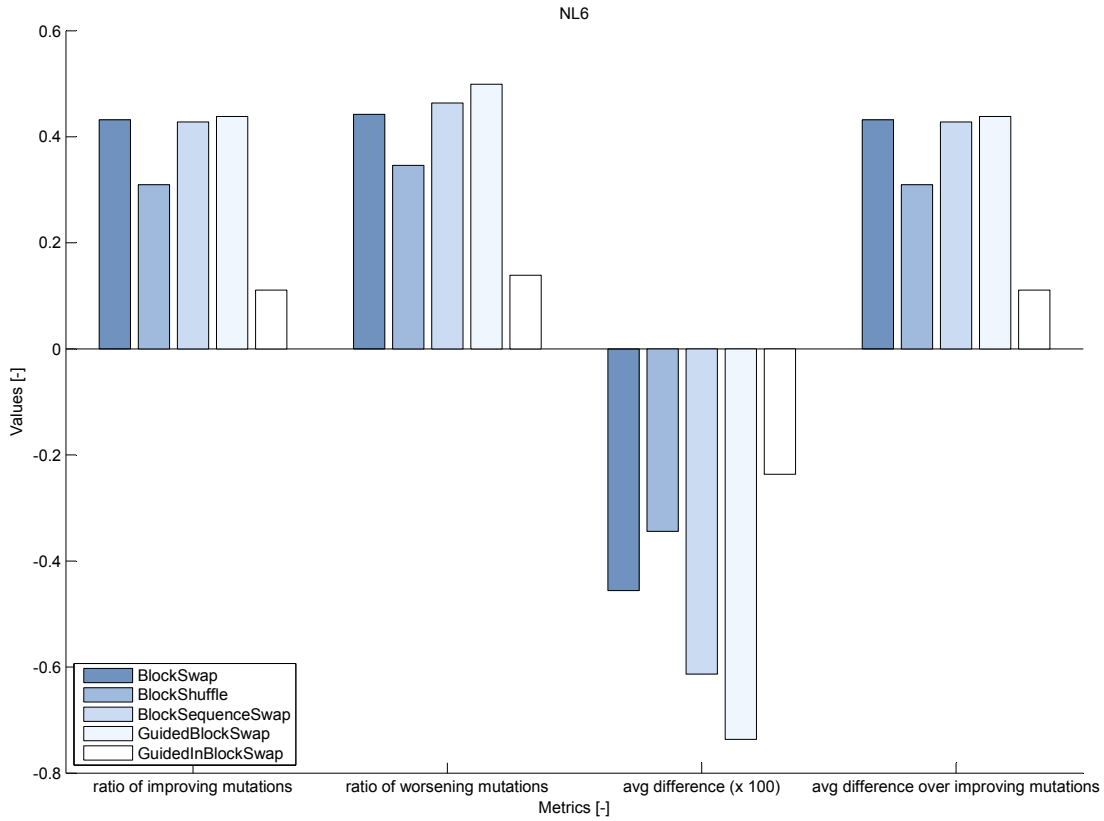


Figure A.2. Result of experiment with mutations for instance NL6

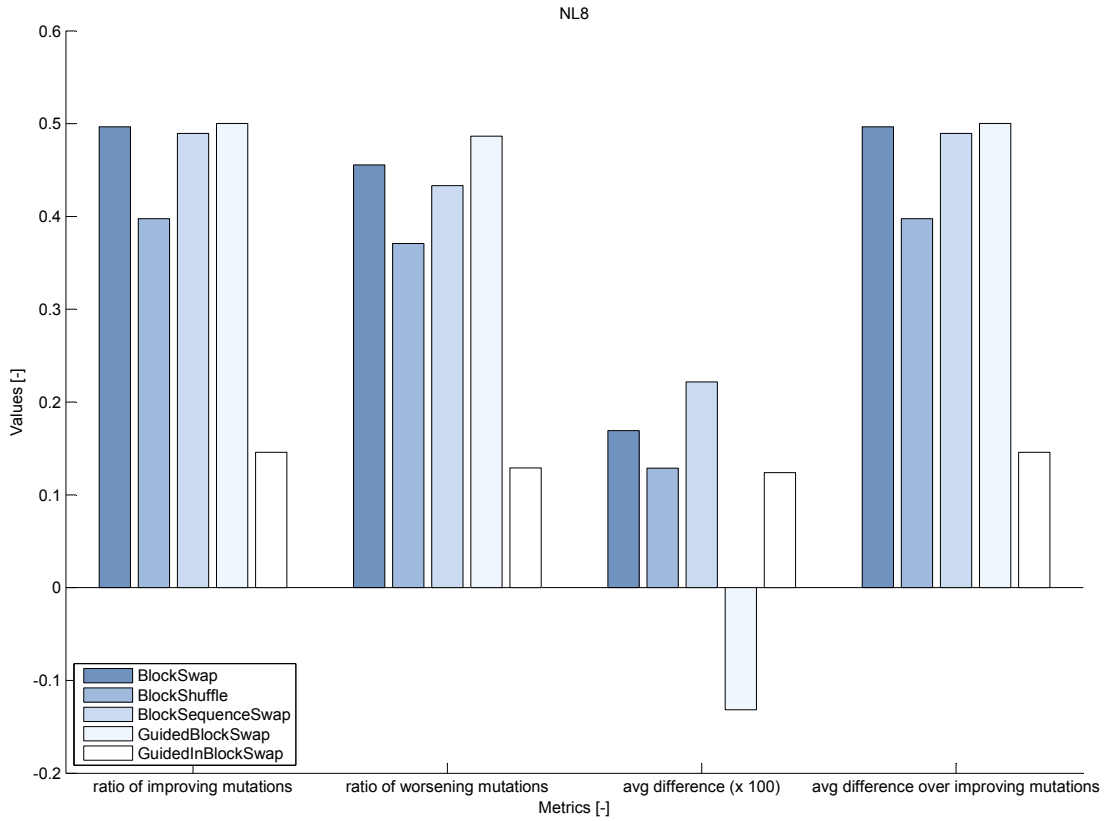


Figure A.3. Result of experiment with mutations for instance NL8

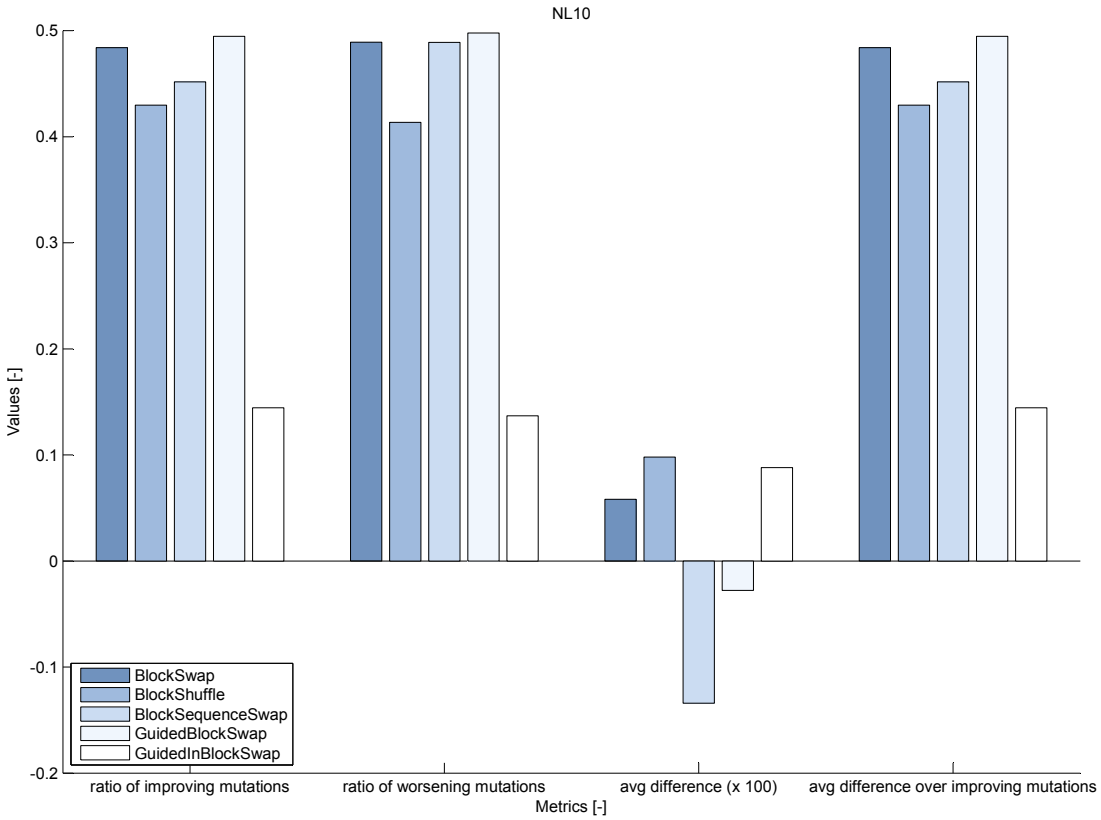


Figure A.4. Result of experiment with mutations for instance NL10

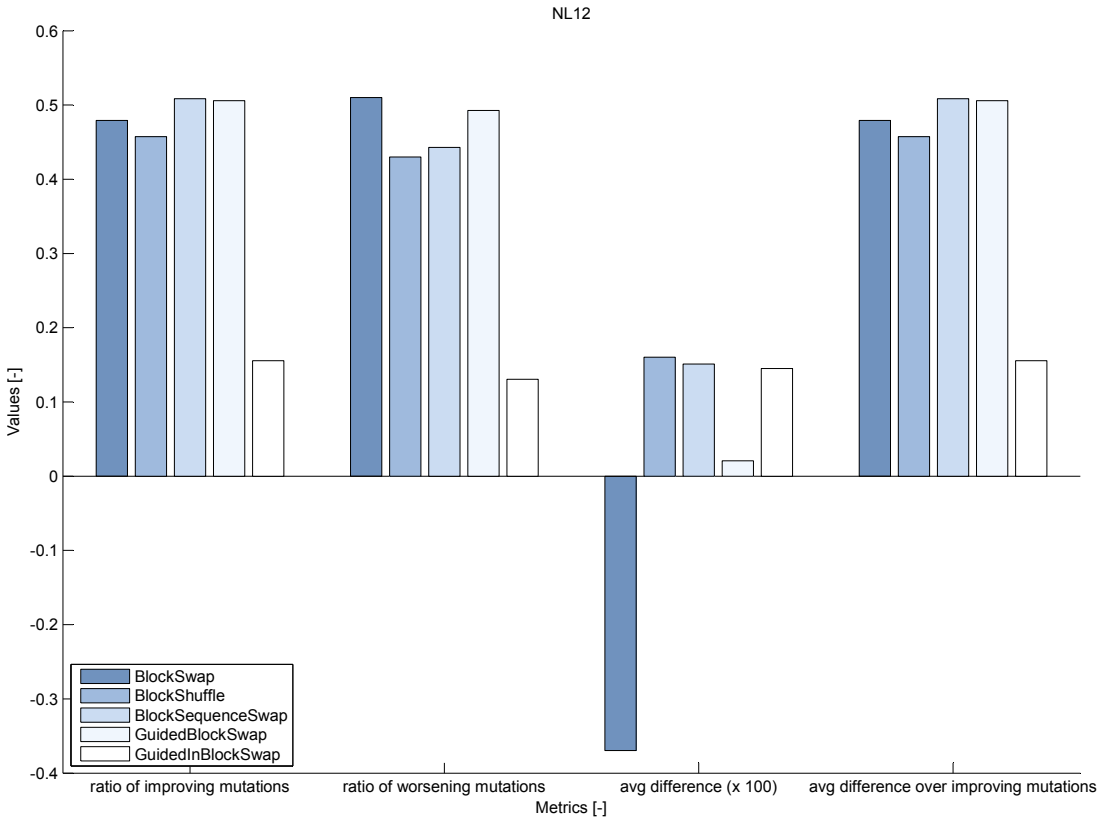


Figure A.5. Result of experiment with mutations for instance NL12

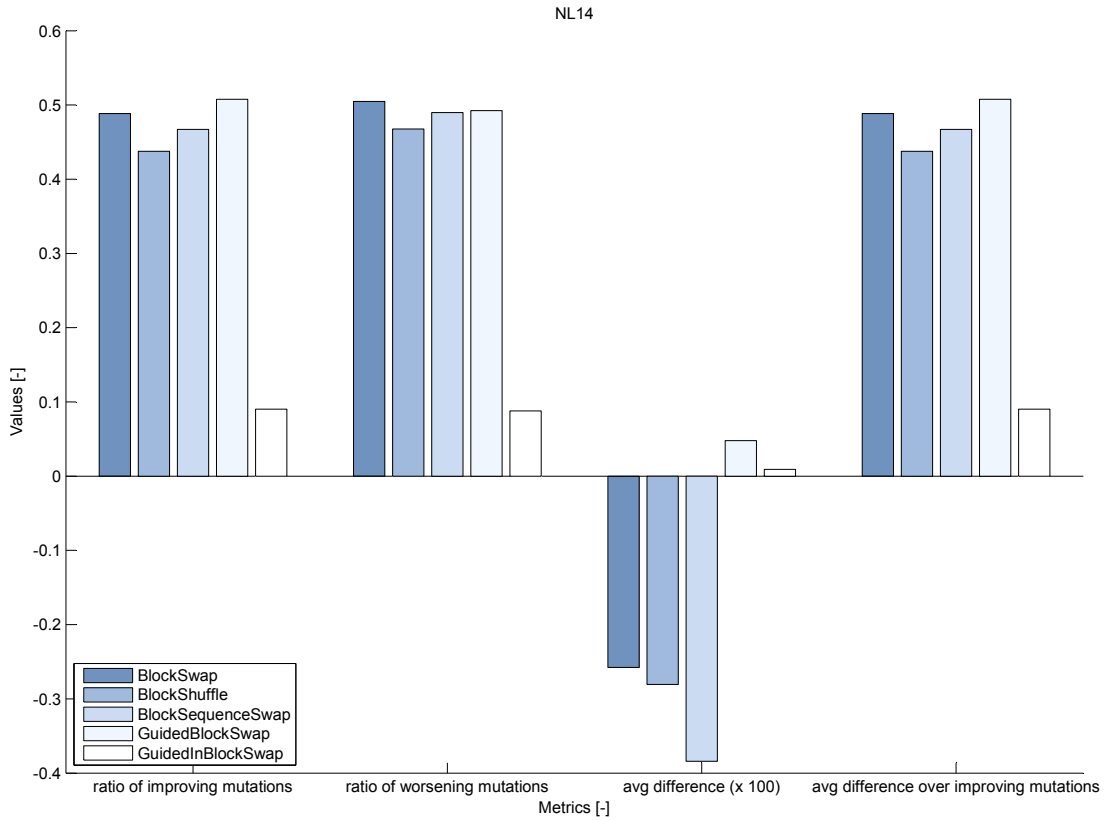


Figure A.6. Result of experiment with mutations for instance NL14

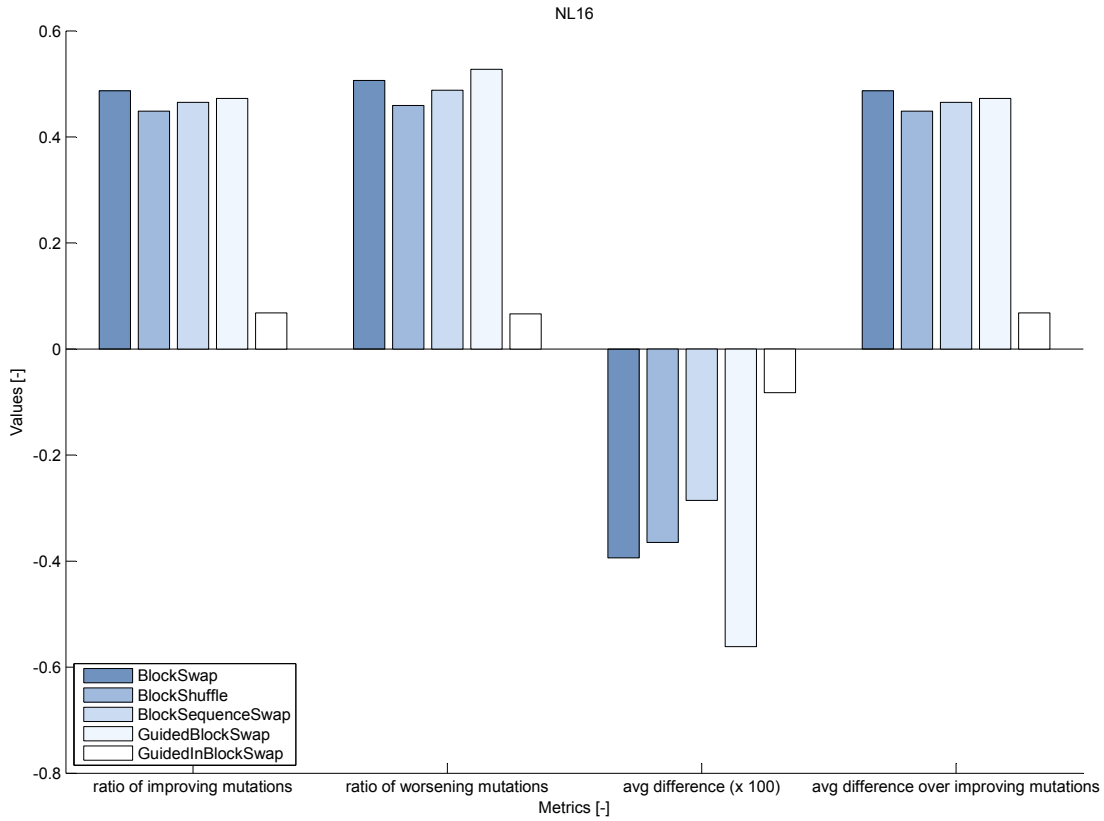


Figure A.7. Result of experiment with mutations for instance NL16

Appendix B
CD contents

```
ROOT

...\thesis\
...\...\sources\
...\...\thesis.pdf

...\implementation\
...\...\sources\
...\...\TTPMA.jar
```