FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Iterated Local Search for the Traveling Tournament Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Bong Min Kim
Matrikelnummer 0327177

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Priv.-Doz. Dr. Nysret Musliu

Wien, 20.08.2012

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Bong Min Kim
Schulzgasse 20, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First and most I want to thank my supervisor Nysret Musliu, whose advice and guidance were invaluable to me. I also want to express my gratitude to Prof. Andrea Schaerf for providing and discussing details of his work to support my thesis.

But ultimately, this thesis wouldn't have been possible without the support and encouragement of my parents, to whom I dedicate this work.

The research herein is partially conducted within the competence network Softnet Austria (http://www.soft-net.at/) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the City of Vienna in terms of the center for innovation and technology (ZIT).

Abstract

The problem of finding optimal schedules for professional sports leagues has attracted interests of many researchers in recent years. On the one hand, sport scheduling is an economically important class of combinatorial optimization applications, since sport leagues generate considerable amount of revenue for major radio and television networks, whose profits also depend on the quality of schedules. On the other hand, sport scheduling poses a very challenging optimization problem which involves issues of both feasibility and optimality.

The *Traveling Tournament Problem* (TTP) is a challenging sport scheduling problem abstracting the features of major league baseball (MLB) in the United States. The objective of the TTP is to find a *double-round-robin* tournament schedule minimizing the total distance traveled by the teams and satisfying at the same time the TTP-specific constraints.

To solve TTP, we propose a metaheuristic approach based on *Iterated Local Search* (ILS) framework. *Iterated Local Search* is a simple but yet powerful metaheuristic which has shown very good results for different classes of optimization problems. First, we develop a basic ILS algorithm for the TTP to assess the applicability of the ILS-principle to the TTP. Based on the insights gained by analyzing the basic variant, we further optimize and extend the basic version to improve the performance. One particularly important optimization is the definition of efficient algorithm for *incremental evaluation* which speeds up the computation considerably.

We conduct extensive computational experiments on selected TTP benchmark-sets and compare our results with those obtained by current state-of-the-art approaches in literature. For the *NL-x* benchmark-set, our ILS algorithm is able to solve the smaller instances $[NL_4, NL_6, NL_8]$ to optimality in only few seconds. For larger instances $[NL_{10}, NL_{12}, NL_{14}]$, our algorithm exhibits better average performance than most of other compared approaches being only second to the current best-performing *Simulated Annealing* approach. In general, our results show that our ILS algorithm is competitive with current state-of-the-art approaches in the literature.

Kurzfassung

Das Problem des Findens von optimalen Turnierplänen für professionelle Sportligen hat in letzter Zeit Aufmerksamkeit vieler Wissenschaftler erregt. Die Planung von optimalen Sportplänen ist eine wirtschaftlich wichtige Antwendung kombinatorischer Optimierung, weil die Sportligen heutzutage wichtige Einnahmequellen für große Fernsehsender und Turnierveranstalter darstellen, wobei der erzielte Profit auch von der Qualität der Spielpläne abhängt.

Das *Traveling Tournament Problem* (TTP) ist ein schwieriges Sportplanungsproblem, welches die besonderen Merkmale von amerikanischen Major League Baseball (MLB) abstrahiert. Das Ziel des TTP ist die Minimierung der gesamten Reisedistanz der teilnehmenden Teams, während TTP spezifische Einschränkungen erfüllt bleiben müssen.

Wir stellen ein metaheuristisches Verfahren basierend auf *Iterated Local Search* Framework zur Lösung des TTP vor. *Iterated Local Search* ist ein simples aber gleichzeitig mächtiges Heuristikverfahren, das für viele verschiedene Optimierungsprobleme gute Ergebnisse gezeigt hat. Wir entwickeln zuerst eine Basisvariante des ILS, mit der wir die Anwendbarkeit der ILS-Prinzipien fuer das TTP beurteilen. Basierend auf den Erkenntnissen aus der Analyse der ersten Variante verfeinern und optimieren wir den Basis-ILS weiter. Eine der wichtigsten Optimierungen ist dabei die Definition einer *inkrementellen* Evaluierungsfunktion, was die Geschwindkeit des Algorithmus deutlich steigert.

Wir führen ausführliche Experimente mit unserem Algorithmus durch und vergleichen die Resultate mit den aus der Literatur. Die Ergebnisse mit der *NL-x* Benchmark-set zeigen, dass unser Algorithmus die kleineren Instanzen $[NL_4, NL_6, NL_8]$ in wenigen Sekunden optimal lösen kann. Für größere Instanzen $[NL_{10}, NL_{12}, NL_{14}]$ zeigt unser Verfahren bessere und stabilere Durchschnittsleistung als die meisten Verfahren aus der Literatur, wobei nur das aktuell beste state-of-the-art Verfahren basierend auf *Simulated Annealing* klar bessere Ergebnisse als unsere Methode zeigt. Insgesamt zeigen die Ergebnisse, dass unser ILS-Algorithmus mit den besten state-of-the-art Algorithmen aus der Literatur für das TTP gut konkurrieren kann.

Contents

List of Algorithms

List of Figures

List of Tables

Part I

CHAPTER 1

Introduction

1.1 Motivation

The problem of finding optimal schedules for professional sports leagues has attracted interests of many researchers in recent years. On the one hand, the scheduling of sport leagues is an economically important class of combinatorial optimization applications, since sport leagues generate considerable amount of revenue for major radio and television networks and neither the sporting event organizers nor the participating teams want to waste their investments and resources due to the poor schedules of games. On the other hand, sport scheduling poses a very challenging optimization problem with multiple objectives and constraints combining issues of feasibility and optimality.

The *Traveling Tournament Problem* (TTP), which is proposed by Easton, Nemhauser and Trick [15] in year 2001, is a challenging sport scheduling problem abstracting the features of major league baseball (MLB) in the United States. The objective of the TTP is to find a *double-round-robin* tournament schedule minimizing the total distance traveled by the teams and satisfying at the same time the TTP-specific constraints. One can say that the TTP is a combination of the well-known *Traveling Salesman Problem* and the sport timetabling problem, for which already various effective solution techniques exist. But the combination of the both optimality- and feasibility-issues makes the TTP a much more difficult optimization problem than its individual underlying "sub-problems".

Since its introduction, the TTP has received considerable attention and numerous different approaches have been devised to tackle this hard optimization problem. The very first solving techniques proposed for the TTP were exact-methods like constraint programming and integer programming, but their limit was quickly reached even for the smallest instances. Then one

of the first successful metaheuristics approach using the *Simulated Annealing* technique was proposed by Anagnostopoulos et al. [4] introducing basic neighborhoods, which are used by nearly all the subsequent meta-heuristics researches for the TTP. In the following years, it was further enhanced to the current state-of-the-art meta-heuristics for solving the TTP.

Many more metaheuristics approaches followed using different techniques like *Tabu-Search* [17], *Ant Colony Optimization* [34] and *Hyper-heuristics* [24]. Based on the researches done so far, one can recognize that *single-solution* based metaheuristics (like *Simulated Annealing* and *Tabu-Search*) are performing particularly well for the TTP. The *Iterated Local Search* (ILS) is another *single-solution* based metaheuristics technique, which can exhibit very powerful performance if properly optimized, and it has been successfully applied to various optimization problems.

To the best of our knowledge, the ILS hasn't been closely analyzed in connection with the TTP yet and we believe that investigating ILS' applicability to the TTP would be an useful contribution to improve general understanding of the metaheuristics' effectiveness for solving this very challenging combinatorial optimization problem.

During the study of previous TTP-researches, we also identified some interesting questions regarding the neighborhoods, which are first introduced in [4] and used by many following metaheuristics. For instance, is it possible to define an efficient *incremental* evaluation function for these neighborhoods? Is the resulting search-space fully *connected* under these neighborhoods?

1.2 Aims of this thesis

The main goals set for this thesis are:

- Give a detailed description of the *Traveling Tournament Problem* and discuss various successful state-of-the-art approaches in the literature applied to this problem.
- Develop a novel metaheuristics approach based on *Iterated Local Search* to solve the TTP.
- Investigate some open issues regarding the widely used neighborhoods defined in [4].
- Implement the proposed algorithms and conduct extensive computational experiments comparing the results with the best results in the literature

1.3 Results

The main results obtained in the course of this thesis are:

- We have developed an ILS-based metaheuristics approach for solving the TTP. First, we start from the very basic version of the ILS algorithm. Based on the insights gained through analyzing the preliminary results, we further optimize and extend the basic version improving the performance significantly.
- The experimental results show that our ILS-based approach is very competitive with state-of-the-art approaches in the literature confirming the effectiveness of the ILS for solving the TTP.
- Further, using the neighborhoods from [4], we have given detailed description how to design and implement *incremental* evaluation function efficiently.
- We propose experimental approach for investigating the connectivity of the search-space under the neighborhoods of [4]. Our results support the hypothesis that the search-space is connected under the considered neighborhoods.

1.4 Organization

The rest of the thesis will be organized as follows. In Chapter 2, we give a formal description of the *Traveling Tournament Problem* and discuss current state-of-the-art approaches. The general principles of the *Iterated Local Search* will be discussed in Chapter 3. Our novel ILS-based approach for solving the TTP is presented in Chapter 4. Chapter 5 compares the experimental results of our approach with the results of the state-of-the-art approaches given in the literature. Chapter 6 concludes this thesis with closing remarks and the outlook on future works.

CHAPTER 2

The Traveling Tournament Problem (TTP)

2.1 Problem Description

The *Traveling Tournament Problem*, which is considered one of the most challenging sport scheduling problems to date, was originally introduced by Easton, Nemhauser and Trick [15], .

Given n teams with n even and an $n \times n$ symmetric distance matrix D, where $D(i, j)$ represents the distance between the cities of team T_i and T_j, the goal in solving the traveling tournament problem is to find a valid *double round robin* schedule, such that the total traveling distance of all teams is *minimized*. A schedule is valid for the traveling tournament problem, if it satisfies the following constraints:

1. *Double Round-Robin* constraint: Each team plays with each other team exactly two times, once in its own city and once in its opponent's city

2. *AtMost* constraint: Each team must play no more than u and no less than l *consecutive* games in or away from the home city

3. *NoRepeat* constraint: It is not allowed that two teams are playing each other in two consecutive rounds

We call a schedule *feasible*, if it satisfies all the constraints above, otherwise *infeasible*. Note that, if u is set to $n - 1$, then finding the schedule with the shortest traveling distance for one team T_i is equivalent to solving the *Traveling Salesman Problem*. It is somewhat misleading

to name the second constraint as the *AtMost* constraint, since we have both lower- and upper bounds for the number of consecutive home- and away-games, but for most of the benchmark instances available in the literature [3], the parameter l is set to 1, so that many researchers ([4], [17]) adopted the notion *AtMost* constraint, because they consider only the upper bound.

It is obvious that a double round-robin schedule for n teams (n even) consists of at least $2n-2$ rounds and for the regular TTP we shall only consider double round-robin schedules with this minimum number of rounds.

In this work, a schedule is represented by an $n \times (2n-2)$ matrix S of integer numbers (like in [4]), where the teams are assigned unique positive integer numbers from [1..n]. The entry $S_{i,j}$ of the schedule matrix (see Figure 2.1) represents the game, which is played by the team T_i in round R_j. The game entry $S_{i,j}$ is a positive integer number t, if the team T_i plays against a team, whose assigned number is t, in round R_j in its own home city. On the other hand if the game takes place in the opponent's home city, it will be represented with a negative integer number $-t$.

T/R	R_1	R_2	R_3	R_4	...	R_{2n-2}
T_1	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,4}$	...	$S_{1,(2n-2)}$
T_2	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$	...	$S_{2,(2n-2)}$
T_3	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,4}$	...	$S_{3,(2n-2)}$
T_4	$S_{4,1}$	$S_{4,2}$	$S_{4,3}$	$S_{4,4}$	...	$S_{4,(2n-2)}$
...						
T_n	$S_{n,1}$	$S_{n,2}$	$S_{n,3}$	$S_{n,4}$	...	$S_{n,(2n-2)}$

Figure 2.1: A double-round-robin schedule given in $n \times (2n-2)$ table

For instance, lets consider a valid TTP double-round-robin schedule with 6 teams. According to our representation, the schedule is represented as a 6×10 matrix of signed integer numbers:

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+6$	-2	$+4$	$+3$	-5	-4	-3	$+5$	$+2$	-6
T_2	$+5$	$+1$	-3	-6	$+4$	$+3$	$+6$	-4	-1	-5
T_3	-4	$+5$	$+2$	-1	$+6$	-2	$+1$	-6	-5	$+4$
T_4	$+3$	$+6$	-1	-5	-2	$+1$	$+5$	$+2$	-6	-3
T_5	-2	-3	$+6$	$+4$	$+1$	-6	-4	-1	$+3$	$+2$
T_6	-1	-4	-5	$+2$	-3	$+5$	-2	$+3$	$+4$	$+1$

Figure 2.2: Representation of a double-round-robin schedule with 6 teams

Using the representation given in Figure 2.2, the *Traveling Tournament Problem* can be formally formulated as follows (see also [15], [32]):

Problem-Instance: A 5-tuple (n, l, u, T, D), where

n: Number of participating teams

l: Lower bound for the *AtMost* constraint

u: Upper bound for the *AtMost* constraint

T: Mapping function, which assigns the integer numbers $[1..n]$ to the teams. We will denote the team, to which the number $i \in [1..n]$ is assigned, with T_i

D: $n \times n$ distance matrix, where $D(i, j)$ denotes the distance between the home city of T_i and the home city of T_j

Objective: Let $P = (n, l, u, T, D)$ be an instance of the *Traveling Tournament Problem*. The objective for solving the given instance is to find a schedule S, which minimizes the following objective function:

$$\sum_{i=1}^{n} \Gamma(T_i)$$

where

- $\Gamma(t) = \sum_{r=0}^{2n-2} \Phi(t, r)$
- $\Phi(t,r) = \begin{cases} 0 & \text{if } r = 0 \text{ and } S_{t,r+1} > 0 \\ D(t, |S_{t,r}|) & \text{if } r = 0 \text{ and } S_{t,r+1} < 0 \\ 0 & \text{if } 1 \leq r < (2n-2) \text{ and } S_{t,r} > 0 \text{ and } S_{t,r+1} > 0 \\ D(|S_{t,r}|, |S_{t,r+1}|) & \text{if } 1 \leq r < (2n-2) \text{ and } S_{t,r} < 0 \text{ and } S_{t,r+1} < 0 \\ D(|S_{t,r}|, t) & \text{if } 1 \leq r < (2n-2) \text{ and } S_{t,r} < 0 \text{ and } S_{t,r+1} > 0 \\ D(t, |S_{t,r+1}|) & \text{if } 1 \leq r < (2n-2) \text{ and } S_{t,r} > 0 \text{ and } S_{t,r+1} < 0 \\ 0 & \text{if } r = (2n-2) \text{ and } S_{t,r} > 0 \\ D(|S_{t,r}|, t) & \text{if } j = (2n-2) \text{ and } S_{t,r} < 0 \end{cases}$

Furthermore the following constraints must be satisfied:

$$C_1: \quad S_{t,r} = -S_{|S_{t,r}|,r} \qquad \forall t \in [1..n], \forall r \in [1..(2n-2)]$$

$$C_2: \quad \sum_{r=1}^{2n-2} \Psi(|S_{t,r}|, t') = 2 \qquad \forall (t,t') \in \{(t,t') \,|\, t \in [1..n],\, t' \in [1..n],\, t \neq t'\}$$

$$C_3: \quad \sum_{r=1}^{2n-2} S_{t,r} = 0 \qquad \forall t \in [1..n]$$

$$C_4: \quad |\sum_{k=0}^{u-1} \Omega(S_{t,r+k})| \leq u \qquad \forall t \in [1..n], \forall r \in [1..(2n-1-u)]$$

$$C_5: \quad |\sum_{k=0}^{l-1} \Omega(S_{t,r+k})| \geq l \qquad \forall t \in [1..n], \forall r \in [1..(2n-1-l)]$$

$$C_6: \quad |S_{t,r}| \neq |S_{t,r+1}| \qquad \forall t \in [1..n], \forall r \in [1..(2n-3)]$$

where

- $\Psi(g,t) = \begin{cases} 1 & \text{if } g = t \\ 0 & \text{otherwise} \end{cases}$
- $\Omega(g) = \begin{cases} 1 & \text{if } g > 0 \\ -1 & \text{if } g < 0 \\ 0 & \text{otherwise} \end{cases}$

The constraints C_1, $C2$ and $C3$ ensure that the solution is a valid double-round-robin tournament. The constraints C_4 and C_5 make sure that the *AtMost* constraint is satisfied, which constricts the number of consecutive home-games (homestand) and away-games (roadtrip) to be between l and u. Finally the constraint C_6 guarantees the fulfillment of the *NoRepeat* constraint that two teams can't play each other in two consecutive rounds.

2.2 Current State-of-the-Art heuristics

Although the TTP is a relatively new problem, it has attracted interests of many researchers due to its practical relevance and its surprisingly high degree of difficulty, which results from the combination of two well-known problems of finding the shortest tour (optimality) and timetabling sport tournaments satisfying certain constraints (feasibility).

When the TTP has been introduced for the first time, the initial approaches proposed for solving the TTP were *exact methods* like integer programming, constraint programming [16]

and hybrid methods [7]. But even for small instances, it was extremely difficult for an exact algorithm to solve them in reasonable time.

The current most sophisticated exact algorithm for the TTP is proposed by Uthus [33]. It is based on branch-and-bound technique and is capable of solving the *National League* benchmark instances optimally up to the size of 10 teams.

To the best of our knowledge, one of the first successful metaheuristics for the TTP has been designed by Anagnostopoulos et al. [4] using the *Simulated Annealing* framework. It is one of the most successful heuristics approaches for the TTP and it has produced numerous best upper-bounds for most of the publicly available benchmark-sets. But the excellent solution quality comes with very long computation time, spending days of computation for larger instances.

The most valuable contribution of their work was the design and definition of the neighborhoods, which have been used nearly by all following metaheuristics for the TTP. The key idea of their neighborhoods is to distinguish between *hard constraints* and *soft constraints*. The hard constraints must be satisfied all the time during the search, whereas the soft constraints can be occasionally violated. This idea stems from the observation that some constraints in the TTP are extremely difficult to repair during the search, once they are violated.

After the Simulated Annealing approach, more single-solution-based metaheuristics followed. Another very successful metaheuristic based on *Tabu search* was developed by Di Gaspero and Schaerf [17]. Their algorithm uses composite neighborhoods based on the same neighborhoods of [4]. Through further fine-tuning of the moves and careful analytical study about the effectiviness of the different composite neighborhoods, they were able to obtain very good results, which are comparable to the best results in the literature.

An interesting hybrid metaheuristic approach was introduced by Lim et al. [20], which divides the search-space in two parts. The algorithm alternates between two components to improve the current solution. The first component, using a Simulated Annealing algorithm, tries to improve the solution by optimizing the timetable with a fixed team assignment, whereas the second component, which incorporates the hill-climbing technique, searches for better team assignment with a fixed timetable. So the fundamental idea in this approach is to improve the timetable, when a good team assignment has been found, and to search for a better team assignment, if the timetable looks promising.

Four years later after the first version of the Simulated Annealing approach [4] was proposed, a population-based extension has been proposed in [35]. This extension made the parallelization of the first SA algorithm possible and it produced the current upper-bounds for numerous benchmark instances running on a a cluster of 60 Intel-based, dual-core, dual-processor Dell Poweredge 1855 blade servers. For more details on this approach, we further refer to the original paper [35].

In recent years, other promising heuristics techniques based on *Ant Colony Optimization* and *Hyper-Heuristic* have been proposed for solving the TTP. In the past, there were already 2 ACO-based attempts ([9],[12]) for the TTP, but their results were relatively poor. The new ACO approach proposed by Uthus [34] in 2012, which incorporates some advanced extensions like *forward checking* and *conflict-directed backjumping algorithm* (see also [34]), is able to improve greatly on the solution quality compared to the previous ACO-based attempts. His new results are competitive with those of the state-of-the-art heuristics.

The Hyper-Heuristic method proposed by Misir et al. [24] also gives very promising performance. Their Hyper-Heuristic is composed of a simple selection mechanism based on a *learning automaton* and a novel acceptance mechanism, which they call as the *Iteration Limited Threshold Accepting* criterion. Despite of the simple and general nature of the Hyper-Heuristic, their method is able to produce very good solutions in relatively short amount of time.

2.3 Other variants of the Traveling Tournament Problem

The TTP has spawned some interesting variants of the original problem over the years. In this section we are going to briefly summarize some of them.

2.3.1 Mirrored Traveling Tournament Problem (mTTP)

For the *mirrored* version of the *Traveling Tournament Problem*, the *NoRepeat* constraint of the original problem is replaced with the new *Mirror* constraint:

$$S_{t,r} = -S_{t,r+n-1} \qquad \forall t \in [1..n], \forall r \in [1..n-1]$$

The *Mirror* constraint states that if a team T_i plays in its home city team T_j, then T_i should play T_j in T_j's home city exactly $n-1$ rounds later and vice versa. Also note that the *NoRepeat* constraint is implicitly satisfied if the *Mirror* constraint is satisfied. Figure 2.3 shows an example of a mirrored double-round-robin schedule.

T/R	R_1	R_2	...	R_{n-1}	R_n	R_{n+1}	...	R_{2n-3}	R_{2n-2}
T_1	$S_{1,1}$	$S_{1,2}$	...	$S_{1,n-1}$	$-S_{1,1}$	$-S_{1,2}$	...	$-S_{1,n-2}$	$-S_{1,n-1}$
T_2	$S_{2,1}$	$S_{2,2}$	...	$S_{2,n-1}$	$-S_{2,1}$	$-S_{2,2}$	...	$-S_{2,n-2}$	$-S_{2,n-1}$
T_3	$S_{3,1}$	$S_{3,2}$	...	$S_{3,n-1}$	$-S_{3,1}$	$-S_{3,2}$	...	$-S_{3,n-2}$	$-S_{3,n-1}$
...									
T_n	$S_{n,1}$	$S_{n,2}$	...	$S_{n,n-1}$	$-S_{n,1}$	$-S_{n,2}$	...	$-S_{n,n-2}$	$-S_{n,n-1}$

Figure 2.3: A mirrored double-round-robin schedule

The mTTP is very well worth to be studied, because the mirrored tournament structure is common in some countries (e.g. Latin America). The mTTP was first studied extensively by Riberio and Urrutia in [26]. In their work, they proposed an approach called *GRILS*, which is a combination of GRASP and Iterated Local Search framework, for solving the mTTP. Their experimental results showed that their algorithm was very fast compared to other existing heuristics for the non-mirrored TTP at that time and for some benchmark instances, it produced even better results than its non-mirrored counterparts.

2.3.2 Non-Round-Robin Tournament Problem

The *Non-Round-Robin Tournament Problem* [1] is a variant of the TTP, which was originally formulated by Douglas Moody.

In this variant, the wanted tournament schedule is no longer a double-round-robin schedule. Instead we are given a so-called "Matchup"-matrix M, which defines the exact number of visits for each team. Concretely the "Matchup"-matrix M is an $n \times n$ matrix, where the entry $M_{i,j}$ with $i \neq j$ defines the number of times T_i has to visit T_j.

As a result, the constraint of the original TTP that every team has to visit each other team exactly one time is modified so that every team T_i has to visit each other team T_j exactly $M_{i,j}$ times.

2.3.3 Relaxed Traveling Tournament Problem

This variant of the TTP, which was first proposed by Bao and Trick [2], relaxes the "compactness constraint" allowing that the solution can consist of more than $2 \times (n-1)$ rounds.

The relaxation permits the teams to have *byes* in their schedule, which means that they don't have to play in every round. The number of byes, the teams are allowed to have, is controlled by a parameter, which is denoted usually with K. If the K is set to 0, then it corresponds to the original TTP.

The byes are ignored when checking the feasibility of the *AtMost* constraint and the *NoRepeat* constraint.

2.4 Complexity

For a long time since the TTP has been first proposed, the complexity of the TTP remained an open issue. Many researchers believed that the TTP must be computationally hard, when even

alone the underlying sub-problem Traveling Salesman Problem is *NP-Hard*. But even the signs were very strong, there was no formal proof to prove the complexity of the TTP until recently.

The first NP-completeness proof has been given by Bhattacharyya [8] for a variant of the original TTP, where the constraint on consecutive home-games and away-games is left out.

The second attempt on TTP's complexity proof is made by Thielen and Westphal [31]. They showed that the TTP is *strongly NP-complete*, when the upper-bound of consecutive home-games and away-games is fixed to 3. It still doesn't prove the original TTP, where u and l can be arbitrary integer numbers, but nonetheless it is a big contribution in the analysis of the TTP's complexity. Furthermore, the authors also pointed out that with further refinement their proof can be probably generalized for the original TTP.

Part II

CHAPTER 3

The Iterated Local Search (ILS) framework

3.1 Iterated Local Search

There is an excellent overview of the Iterated Local Search framework given in [21], where they presents the main principles of the ILS in terms of the three main components. In this chapter, we will give brief descriptions about these components and discuss their impacts on the overall performance of the ILS.

The Iterated Local Search is both conceptually and practically very simple metaheuristics framework. The basic idea behind the ILS is to use the embedded local-search component *iteratively* restarting it from different *promising areas* in the search-space.

Then how can one actually identify the promising areas for the restarts? In one extreme end, we can determine the next restarting point in completely random fashion, where we then get a simple *Random-Restart* scheme. But for many problems, this scheme is very unlikely to perform well (see also [21],[18],[27]), because without using any information of the previous search the algorithm will most likely just stray "blindly" in the search-space. On the other extreme end, we can always restart from the "best position" found so far, but this strategy will increase the danger of getting easily stuck in local optima.

The ILS considers the embedded local-search heuristic as a kind of black box component and uses its output as a basis for determining the next starting point, trying to guide the search into the promising areas. In doing so, the nature and strength of the perturbation of the local-search's output is critical for the performance of the ILS. If the perturbation is too weak, meaning that not enough new attributes are introduced into the current search point, the algorithm risks getting

stuck early in local optima. On the other hand, if the perturbation is too strong, we will lose too much information from the previous search. In worst case, it will be then not better than just restarting the search from a random starting point.

Besides perturbation, there is another important aspect of the ILS, to which we should pay close attention, namely the criteria how to accept the local optima found by the embedded local-search component for the next iteration. To this end there are several different strategies to consider, which we will discuss in detail later on.

In summary, we can modularize the ILS framework into following three main components as described in [21]:

- Perturbation
- Local-search component
- Acceptance criterion

Having these individual components cleanly modularized reduces the complexity of the framework and makes it easier to optimize the overall performance by fine-tuning the components independently. Of course, it should be clear that the components can not function completely independent from each other. In order to achieve maximum performance, we also should carefully study and understand their correlations and impacts on each other, which will vary from problem to problem.

The main "work-flow" of the ILS framework can be depicted as in Figure 3.1.

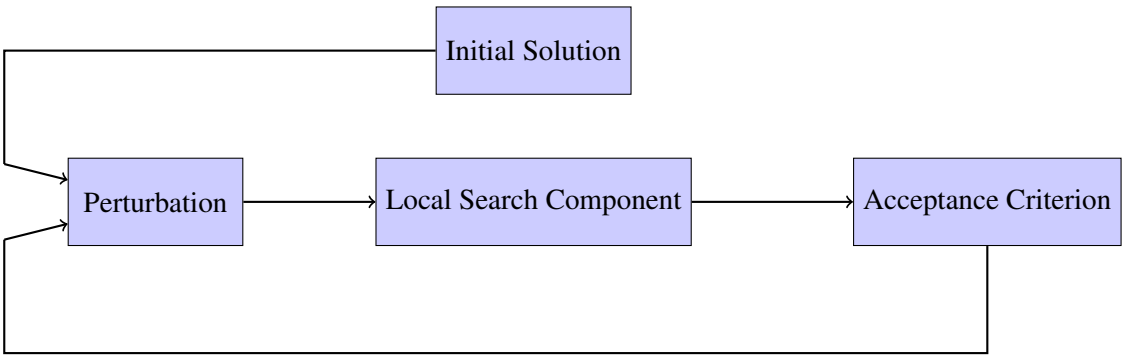


Figure 3.1: Main work-flow of the ILS framework

We start with an initial solution, which is usually generated using some random constructive procedures. Then we use the local-search component to obtain a local optimum, which is either accepted or discarded according to the chosen acceptance criterion. Then the local search component restarts with a new starting point, which is obtained by perturbating the current accepted solution. The ILS-template, formulated as a pseudo-code, is given in Algorithm 1.

Algorithm 1 Template of the ILS [21]

```
1:  s = generateInitialSolution()
2:
3:  while !stopCondition do
4:     s' = perturbate(s)
5:
6:     s'' = localSearch(s')
7:
8:     if acceptanceCriterion(s, s'') then
9:        s = s''
10:    end if
11: end while
12:
13: return s
```

3.1.1 Initial Solution

Until now, we have somewhat neglected the question how to generate initial solutions for the ILS and what influence they have on the overall performance.

Simply stated, one can either start from fully random initial solutions or may try to use greedy procedures in order to construct good-quality start solutions.

But in general, there is not always a clear best choice regarding the initial solutions for the ILS. Sometimes greedy initial solutions appear to be recommendable when one has to obtain good-solutions quickly. For instance, some experiments ([21]) have shown that for *certain problems* the ILS performs in average better with greedy initial solutions, when short computation time is given.

For much longer running-time, the meaning of the initial solution may become less relevant, since in most cases much of the initial properties will get lost during long search. Here, the user may choose the strategy, which is easiest to implement.

3.1.2 Perturbation

The perturbation component is a crucial component, which allows the ILS to escape from local optima. The ILS tries to modify the local optimum in a certain way, so that the local-search component can jump to another promising region in the next iteration.

At this point, we introduce the notion *perturbation strength*, which specifies how strong the current local optimum will be modified. Obviously we should be very careful in choosing the appropriate perturbation strength for the ILS. If the perturbation is too strong, we run the risk of losing good properties found in the previous searches, which is against the concept of the ILS. On the other hand, if we are too "petty" with the perturbation, the chance to successfully escape from local optima will be very low. So as you can see, one of the most important aspects in fine-tuning the ILS will be the task of finding a nice balance for the perturbation strength considering the points mentioned above.

For example, there are different strategies proposed in the past to handle the perturbation strength during the search:

- **Static**: the perturbation length is fixed a priori before the search and is no longer modified during the search.
- **Dynamic**: the perturbation length is modified dynamically during the search without taking the search history into account (can be random variations of the perturbation length in a certain interval).
- **Adaptive**: the perturbation length is modified dynamically during the search exploiting the information (i.e. about the shape of the landscape) gathered during the search.

Finding effective perturbation methods is a highly "problem-specific" matter and depends also on the used embedded local-search heuristic. One important aspect to consider is, that the perturbation shouldn't be easily undone by the local-search component, otherwise one will fall back into the local-optimum just visited. Furthermore, one should try to exploit as much problem-specific properties as possible in the perturbation component complementing possible shortcomings of the local-search component.

3.1.3 Acceptance Criterion

Alongside the perturbation component, the acceptance criterion will also have a great influence on the effectiveness of the ILS framework. We consider the ILS as an heuristic approach, which "random-walks" in the search-space S^* consisting of local optima defined by the embedded local-search component. The perturbation mechanism together with the local-search component

defines the *transition* from one local optimum $s' \in S^*$ to the "neighboring" local optimum $s'' \in S^*$ and the acceptance criterion determines whether the neighbor s'' will be accepted or not for the next iteration.

The chosen acceptance criterion has a critical influence on the balance between *intensification* and *diversification* of the search. On the one hand, we can define the acceptance criterion to accept only better local optima than the current one. We call such strategy as the **Better** acceptance criterion ([21]), which can be defined for the minimization problem as follows:

$$\text{Better}(s', s'') = \begin{cases} s' & \text{if Cost}(s') < \text{Cost}(s'') \\ s'' & \text{otherwise} \end{cases}$$

As you can intuitively see, this criterion is an extreme one, which clearly advocates strong intensification.

At the opposite extreme, one can work with a strategy called **Random-Walk** (RW) acceptance criterion [21], which always chooses the *most recently* visited local optimum, irrespective of its cost:

$$\text{RW}(s', s'') = s''$$

This criterion strongly favors diversification over intensification, because every solution in S^* is accepted for the next step.

Obviously in order to find an appropriate balance between these two extremes, we need to find a way to encourage both intensification and diversification in an adequate manner. One of the very successful acceptance criteria applied to the ILS was a simulated annealing type acceptance criterion, which we will denote as the **LSMC** acceptance criterion, reminiscent of the term *large step Markov chains* used for one of the first ILS algorithms [22] with this type of acceptance criterion.

The LSMC criterion accepts always s'', if it is better than the current local optimum s'. Otherwise, if s'' is worse than s', a certain probability p, with which s'' will be accepted, is calculated based on the difference in qualities of s' and s''. The bigger the gap between s' and s'' is, the less the chance that s'' will be accepted. Given s' and its qualitative worse neighbor s'', the acceptance probability p can be calculated as

$$e^{\frac{Cost(s') - Cost(s'')}{T}}$$

, where T is a parameter called temperature, which controls the balance between intensification and diversification.

3.1.4 Local-Search Component

In many overview articles for the ILS, the local search component is described as a "blackbox" module, for which we can use practically any existing *single-solution based metaheuristics*. This gives us two advantages using the Iterated Local Search framework.

- Since we treat the embedded local-search as a blackbox component, we don't have many "dependency problems" between the framework and the local-search component. Therefore, if necessary we can just swap the local-search component without altering the whole framework again.
- As mentioned above, we can use any existing metaheuristics as the embedded local-search component. If there already exists a well performing heuristic for the given problem, then we can quickly develop a potentially better performing ILS-version reusing the existing local-search algorithm as the embedded local search component.

At this point, we want to reemphasize the main principle of the ILS. Roughly speaking, the ILS is nothing more than a "simple" walk in S^*, which can be seen as a subset of the original search-space S consisting of *local optima* produced by the embedded local search. So, we can think of the local-search component as a *mapping function*, which maps the original search-space S into the subset of local optima S^*:

$$S^* := \{\, \text{localSearch}(s) \mid s \in S \,\}$$

Note also that, no explicit neighborhood is defined for the walk in S^*, but instead the components *perturbation* and *acceptance criterion* determine the next neighbor to visit.

You may have already noticed that ideally S^* should be a small compact set of local optima, which contains the global optimum. In order to get a high quality mapping, one, of course, needs a powerful local-search component, which returns high quality local optima. In general, we can assume the better the embedded local-search, the better the corresponding ILS. For example in case of the TSP, the Lin-Kernighan heuristic is better than the 3-opt local-search. Researches have shown that the ILS embedding the Lin-Kernighan heuristics gives better results than the ILS using the 3-opt local-search ([18], [29]), confirming the aforementioned assumption. But high quality comes usually with a high price, namely long running-time. If the computation-time is heavily limited, it would be probably better idea to use a less powerful but faster embedded local-search in order to get useful results more quickly.

As already mentioned for the perturbation, an important aspect to consider when choosing the local-search component is the "collaboration" between the local-search and the perturbation component. The rule of thumb is that local-search shouldn't systematically undo the changes made by the perturbation component ([21]).

3.2 Applications of Iterated Local Search

The Iterated Local Search framework is very simple but at the same time very powerful concept. If properly tuned and optimized, it can often become even a state-of-the-art algorithm. In this section we will give you a quick overview of some combinatorial problems, to which the ILS approach has been successfully applied.

3.2.1 Solving the Traveling Salesman Problem with ILS

The *Traveling Salesman Problem* is probably one of the oldest and most well-studied combinatorial problems in computer science. Not surprisingly one of the first variants of the ILS concept was applied to the Traveling Salesman Problem.

According to [21], the very first attempt to apply ILS concept for the TSP came from Baum [6]. In his approach (called *iterated descent* method at the time), he used 2-opt local-search as the embedded local-search component and a simple perturbation scheme, where he interchanges 3 cities randomly. Although his results were not particularly impressive, it made an useful contribution for further researches with ILS concept.

The next major improvement for the ILS was achieved by Martin, Otto and Felten [22] using a better acceptance criterion and the Lin-Kernighan algorithm for the embedded local-search. In order to be more diversifying, they used a simulated annealing like LSMC acceptance criterion. Besides the improved acceptance criterion, it was also the new perturbating move, so called *double-bridge move*, which contributed much to the improvement of the ILS's performance for solving the Euclidean TSP.

Finally it should be mentioned that the ILS is today among the best performing metaheuristics for the TSP, one concrete example being the *Chained Lin-Kernighan* by Applegate et al. [5]. It's especially interesting to read about their experimental tests, where they compare the effectiveness of fine-tuning different components.

3.2.2 Solving the Scheduling Problems with ILS

Another area, where ILS performs well, is solving scheduling problems. Over the years, the ILS has been applied successfully to various forms of scheduling problems:

- Single Machine Total Weighted Tardiness Problem (SMTWTP)
- Single and parallel machine scheduling
- Flow shop scheduling

There is an excellent overview included in [21] about applications of ILS to the aforementioned scheduling problems. So here we only give a brief summary of important aspects and skip the most of the details. For more details, we refer the reader to the respective original works, which are also referenced in [21].

Studying the various ILS applications for the scheduling problems, we can recognize that in order to be successful with ILS, one has to exploit the problem-specific details as much as possible.

For instance, the ILS implementation by Congram [11] et. al for the SMTWTP shows an interesting way to incorporate problem-specific knowledge into the ILS algorithm. Without going too much into details, they exploited the property of the SMTWTP (a detailed problem description can also be found in [11]) that there exists an optimal solution, where non-late jobs are sequenced in non-decreasing order of the due dates. This property is then successfully exploited to design an effective perturbation move and to improve the computation speed of the local-search component.

For the *Flow Shop Problem*, Stützle proposed an ILS approach [30], which is rather of simple nature. Incorporating simple *hill climbing* local-search component, random perturbation scheme and variant of the LSMC acceptance criterion with constant temperature, he was able to achieve comparable results to the state-of-the-art results of that time.

Besides the TSP and scheduling problems, there are also many other problems, to which the ILS has been successfully applied. For example, the Graph Coloring Problem [10], the Quadratic Assignment Problem [28] and the Tree Decomposition Problem [25], just to name a few.

3.3 Summary

In this chapter we've given a detailed description of the ILS framework. The ILS framework consists of three main components, namely *embedded local search*, *perturbation component* and *acceptance criterion*.

In general, each of these components can be optimized individually, but if we want to achieve maximum performance out of ILS, we should also try to gain deeper understanding how they influence each other and fine-tune them together "globally".

We have presented some selected successful applications of the ILS to various combinatorial optimization problems. Most notably, the ILS framework seems to be well suited both for the TSP and the scheduling problems. This gives us an extra motivation and hope in choosing the ILS framework for solving the TTP.

CHAPTER 4

Applying ILS to the TTP

In this chapter, we propose a metaheuristic approach based on ILS framework for solving the *Traveling Tournament Problem*. First, we will give a detailed description of the neighborhoods, we are going to use for the embedded local-search component.

As mentioned earlier, one of the first successful applications of a single-solution based metaheuristic to the TTP came from Anagnostopoulos et al. [4]. One of the key contributions of their work was the definition of local-move neighborhoods, which are used more or less by all successive metaheuristics approaches for the TTP.

For instance, Di Gaspero and Schaerf designed an approach based on *Tabu Search*, which is one of the current state-of-the-art approaches, using the basic neighborhoods adapted from those of [4]. Gaspero and Schaerf [17] also give an extensive analysis of these neighborhoods providing us deeper insight about their individual properties and the resulting search space. It is especially important to study the relationships between the individual basic neighborhoods, if one wants to work with composite-neighborhoods.

In this work, we are going to use the "well-established" neighborhoods introduced by [4] as well. In doing so, we also want to look into some interesting issues like definition of an efficient incremental evaluation function and investigation of the connectivity of the search-space.

The rest of this chapter is organized as follows. First, we give detailed descriptons of the individual neighborhoods. Then we investigate their connectivity by means of experimental tests and present algorithms to implement *incremental* evaluation function. Finally, we propose at first a basic ILS algorithm, which is optimized and extended further resulting in our final ILS algorithm for solving the TTP.

4.1 Neighborhoods for the Local-Search Component

4.1.1 The Search-Space

As already mentioned, we have defined our neighborhoods based on the ideas introduced in [4]. In particular, we consider five different types of neighborhoods, which can be categorized into three different classes regarding the magnitude of introduced changes:

- N_1: SwapHomes (micro move)
- N_2, N_3: SwapRounds, SwapTeams (macro moves)
- N_4, N_5: PartialSwapRounds, PartialSwapTeams (generalized moves)

But before we continue with the definition of the neighborhoods, let's first take a quick look at the TTP's search-space. To the best of our knowledge, the size and structure of TTP's search space is still under investigation and it is still not clear how big exactly the search-space of *valid* TTP-solutions is.

Nevertheless, if we want to make a rough estimation about the approximate size of the search-space, the obvious upper-bound for a TTP-instance of size n can be given as $\prod_{t=1}^{n}(2n-2)! = ((2n-2)!)^n$, since for each of n teams, there are $(2n-2)!$ permutation possibilities to arrange the order of the matches. But this is actually too big for an estimation, because it also includes all the invalid schedules violating the *Double-Round-Robin* constraint and the TTP-specific *At-Most* and *No-Repeat* constraints.

In order to refine our estimation, let's consider a double-round-robin(DRR) schedule with 6 teams given in Figure 4.1.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

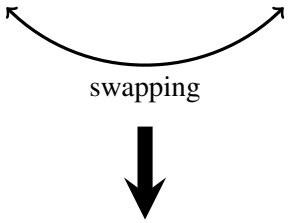
Figure 4.1: DRR-schedule with 6 teams

Looking at this feasible DRR-schedule, we can think of some modification "moves", that we can apply to the schedule and maintain at the same time the feasibility of the *Double-Round-Robin* constraint. For the time being, we will leave the *At Most* and *No Repeat* constraints out.

If we just look at the columns of the schedule, it's easy to recognize that swapping the columns (rounds) doesn't violate the DRR-feasibility at all.

For example, look at the rounds R_3 and R_6:

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$



T/R	R_1	R_2	R_6	R_4	R_5	R_3	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$-T_4$	$+T_3$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$+T_3$	$-T_6$	$+T_4$	$-T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$-T_2$	$-T_1$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$+T_1$	$-T_5$	$-T_2$	$-T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$-T_6$	$+T_4$	$+T_1$	$+T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$+T_5$	$+T_2$	$-T_3$	$-T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

As you can see, the modified schedule is still a valid DRR-schedule. This means, given a certain DRR-schedule, we have already $(2n-2)!$ possibilities to permutate the order of the rounds producing *different* valid DRR-schedules.

In addition, we can also recognize that the swapping of two teams doesn't violate the DRR-feasibility either. Considering the schedule given in Figure 4.1, we can imagine this time the teams T_1,T_2,...,T_n as kind of placeholders, to which arbitrary teams can be assigned. Then obviously there are $n!$ different team-assignments to consider.

Again, let's look at one concrete example:

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

New team assignment, where the teams T_2 and T_4 are swapped:

$$
\begin{aligned}
T_1 &\longrightarrow T_1\\
T_2 &\longrightarrow T_4\\
T_3 &\longrightarrow T_3\\
T_4 &\longrightarrow T_2\\
T_5 &\longrightarrow T_5\\
T_6 &\longrightarrow T_6
\end{aligned}
$$

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_4$	$+T_2$	$+T_3$	$-T_5$	$-T_2$	$-T_3$	$+T_5$	$+T_4$	$-T_6$
T_4	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_2$	$+T_3$	$+T_6$	$-T_2$	$-T_1$	$-T_5$
T_3	$-T_2$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_4$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_2	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_4$	$+T_1$	$+T_5$	$+T_4$	$-T_6$	$-T_3$
T_5	$-T_4$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_2$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$-T_2$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_4$	$+T_3$	$+T_2$	$+T_1$

We have now seen some possibilities to produce different DRR-schedules from a given DRR-schedule. So as you can see, even if the search-space contains only *valid* DRR-schedules, we would still have to deal with at least $(2n-2)!$ possible solution candidates.

In summary, the search-space would consist of total $((2n-2)!)^n$ possible candidates, if we allow that all the three constraints can be violated. If we include all the invalid schedules in the search-space, then the large size of the search-space will make it very difficult to find even valid solutions satisfying all the three constraints, let alone optimizing the travel-distance. Therefore, our neighborhoods consist of only *valid* DRR-schedules, whereas the other two constraints can be violated. We will discuss this later in more details when we describe our ILS algorithm.

4.1.2 Neighborhoods

4.1.2.1 Swap-Homes Neighborhood

From the five neighborhoods mentioned earlier, the *Swap-Homes* neighborhood offers the "smallest local-move", meaning that the number of changes caused by this move is minimal.

Given a valid DRR-schedule, this move swaps the *home/away* states of the teams T_i and T_j, where $i \neq j$. Let's say that team T_i plays T_j in round R_l at home and team T_j plays T_i in round R_k at home, where $i \neq j$ and $k \neq l$. Then after swapping the home/away states of T_i and T_j, team T_i plays T_j in round R_k at home and team T_j plays T_i in round R_l at home. You can easily recognize that there are $O(n^2)$ possible neighbors in this neighborhood. Also notice that the "move-strength" is always 4, since only 4 games are affected by this move.

The procedure for applying the *Swap-Homes* move is depicted in Figure 4.2.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

Swapping the home/away roles of teams T_2 and T_4

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$-T_4$	$+T_3$	$+T_6$	$+T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$+T_2$	$+T_1$	$+T_5$	$-T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

Figure 4.2: Swap-Homes Move

4.1.2.2 Swap-Rounds Neighborhood

Actually, we have already introduced this move in the previous section when we discussed the structure of the search-space. The *Swap-Rounds* move simply swaps two rounds R_k and R_l in the given configuration, where $k \neq l$.

Swapping two rounds is considered to be a "macro" move, meaning that the changes introduced by this move are of quite disruptive nature. The number of affected games by this move is obviously $2 * n$ and there are $O(n^2)$ possible neighbors in this neighborhood. Also we should recall that after swapping two rounds, the new resulting schedule is still a valid DRR-schedule, so no further repair action is needed.

The procedure for applying the *Swap-Rounds* move is depicted in Figure 4.3.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

Swapping the rounds R_2 and R_6

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_4$	$+T_4$	$+T_3$	$-T_5$	$-T_2$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_3$	$-T_3$	$-T_6$	$-T_4$	$+T_1$	$+T_6$	$+T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$-T_2$	$+T_2$	$-T_1$	$+T_6$	$+T_5$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_1$	$-T_1$	$-T_5$	$+T_2$	$+T_6$	$+T_5$	$-T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_6$	$+T_6$	$+T_4$	$+T_1$	$-T_3$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$+T_5$	$-T_5$	$+T_2$	$-T_3$	$-T_4$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

Figure 4.3: Swap-Rounds Move

4.1.2.3 Swap-Teams Neighborhood

Similar to the *Swap-Rounds* move, swapping two teams is a "macro" move, which introduces up to $4 * (2n - 4)$ changes in the given schedule, and is the most disruptive move of the five neighborhoods.

Given two teams T_i and T_j, the *Swap-Teams* move swaps the games of T_i and T_j at every round, except when they play against each other. Obviously the number of affected rounds is $2n - 4$ and at each round, the number of changed games is always 4.

This move is also similar to the *Swap-Rounds* move in the aspect that it doesn't violate the DRR-feasibility after the application either.

The procedure for applying the *Swap-Teams* move is depicted in Figure 4.4.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_2$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

Swapping the teams T_2 and T_4

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_4$	$+T_2$	$+T_3$	$-T_5$	$-T_2$	$-T_3$	$+T_5$	$+T_4$	$-T_6$
T_2	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$+T_4$	$+T_1$	$+T_5$	$-T_4$	$-T_6$	$-T_3$
T_3	$-T_2$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_4$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_4	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$-T_2$	$+T_3$	$+T_6$	$+T_2$	$-T_1$	$-T_5$
T_5	$-T_4$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_2$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$-T_2$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_4$	$+T_3$	$+T_2$	$+T_1$

Figure 4.4: Swap-Teams Move

4.1.2.4 Swap-Partial-Rounds Neighborhood

So far, we have discussed neighborhoods, which are straight-forward and simple to understand. But the local-moves given by these neighborhoods are not sufficient for an effective search resulting only in limited search-space. Therefore, the authors of [4] also introduced the "partial-swapping-moves", which generalize the aforementioned moves. These moves expand the search-space considerably and make the search-space more connected.

Let's first look at the *Swap-Partial-Rounds* neighborhood. Like the *Swap-Rounds* move, we need two parameters R_i and R_j, which specify the rounds that are being swapped. But in addition we also need a team T_k as the third parameter, from which the games at rounds R_i and R_j should be swapped. Obviously, swapping just the two games will violate the DRR-constraint of the schedule, but there is a deterministic way to define a sequence of "repairing movements" in order to restore the DRR-feasibility after the swapping.

The procedure for applying the *Swap-Partial-Rounds* move is depicted in Figure 4.5.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_2$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_4$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_1$	$-T_5$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_6$	$-T_3$
T_3	$-T_4$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_4	$+T_3$	$+T_6$	$-T_3$	$-T_6$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_1$	$-T_5$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_2$	$+T_1$

Partial-swapping the rounds R_2 and R_9 for the team T_2

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$+T_4$	$+T_2$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$-T_2$	$-T_6$
T_2	$+T_5$	$-T_6$	$-T_1$	$-T_5$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$+T_1$	$-T_3$
T_3	$-T_4$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_4	$+T_3$	$-T_1$	$-T_3$	$-T_6$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$+T_6$	$-T_5$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$+T_2$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$-T_4$	$+T_1$

Figure 4.5: Swap-Partial-Rounds Move

As you can see, the *Swap-Partial-Rounds* move doesn't swap the "whole columns", but only the parts which are necessary to maintain the DRR-validity. As already, the parts that are needed to be swapped can be determined in a deterministic fashion.

Let's take a closer look at the above example. Team T_2 plays against team T_1 and T_6 at rounds R_2 and R_9, which therefore should be swapped. Note that the home/away states of the games are irrelevant in this case, so we concentrate only on teams. After swapping the relevant games of T_2, one can see that the games of T_1 and T_6 also should be swapped at rounds R_2 and R_9, since they are affected by the first swap. Swapping games of T_1 and T_6 further affects the team T_4 and gives us a total set of teams $\{T_1, T_2, T_4, T_6\}$, whose games must be swapped at rounds R_2 and R_9 in order to repair the violation of the DRR-constraint.

The authors of [17] introduced the term *repair-chain*, which depicts the way of determining the parts to be swapped in a deterministic fashion. See Figure 4.6.

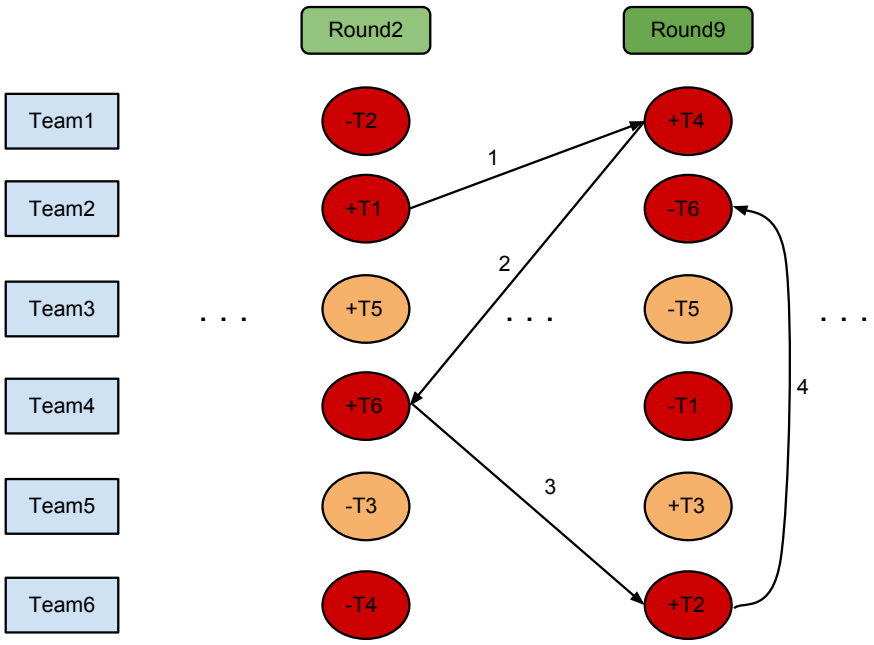


Figure 4.6: Repair chain of the "Partial Swap-Rounds" move

4.1.2.5 Swap-Partial-Teams Neighborhood

In a similar fashion like the *Swap-Partial-Rounds* neighborhood, the *Swap-Partial-Teams* neighborhood is the further generalization of the *Swap-Teams* neighborhood. Given two teams T_i, T_j and round R_k, the *Swap-Partial-Teams* move swaps the games of T_i and T_j at the round R_k and repairs the schedule afterwards so that it becomes a valid DRR-schedule again. In addition, there is an important precondition that T_i doesn't play against T_j at the round R_k.

Similar to the *Swap-Partial-Rounds* move, the repair-chain can be determined in a deterministic way. The move-strength varies from case to case and is given by the actual length of the repair-chain. In the extreme case, the repair-chain can have the length $(2n-4)$, in which case the move equals to the respective *Swap-Teams* move. The repair-chain of the *Swap-Partial-Teams* move can be determined in a similar way as for the *Swap-Partial-Rounds* move, see Figure 4.8.

The procedure for applying the *Swap-Partial-Teams* move is depicted in Figure 4.7.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_2$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_4$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_1$	$-T_5$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_6$	$-T_3$
T_3	$-T_4$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_4	$+T_3$	$+T_6$	$-T_3$	$-T_6$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_1$	$-T_5$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_2$	$+T_1$

Partial swapping the teams T_2 and T_4 at the round R_9

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_4$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_2$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_3$	$-T_6$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_1$	$-T_5$
T_3	$-T_4$	$+T_5$	$+T_2$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_4$
T_4	$+T_3$	$+T_6$	$-T_1$	$-T_5$	$-T_2$	$+T_1$	$+T_5$	$+T_4$	$-T_6$	$-T_3$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_4$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_2$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_4$	$+T_1$

Figure 4.7: Swap-Partial-Teams Move

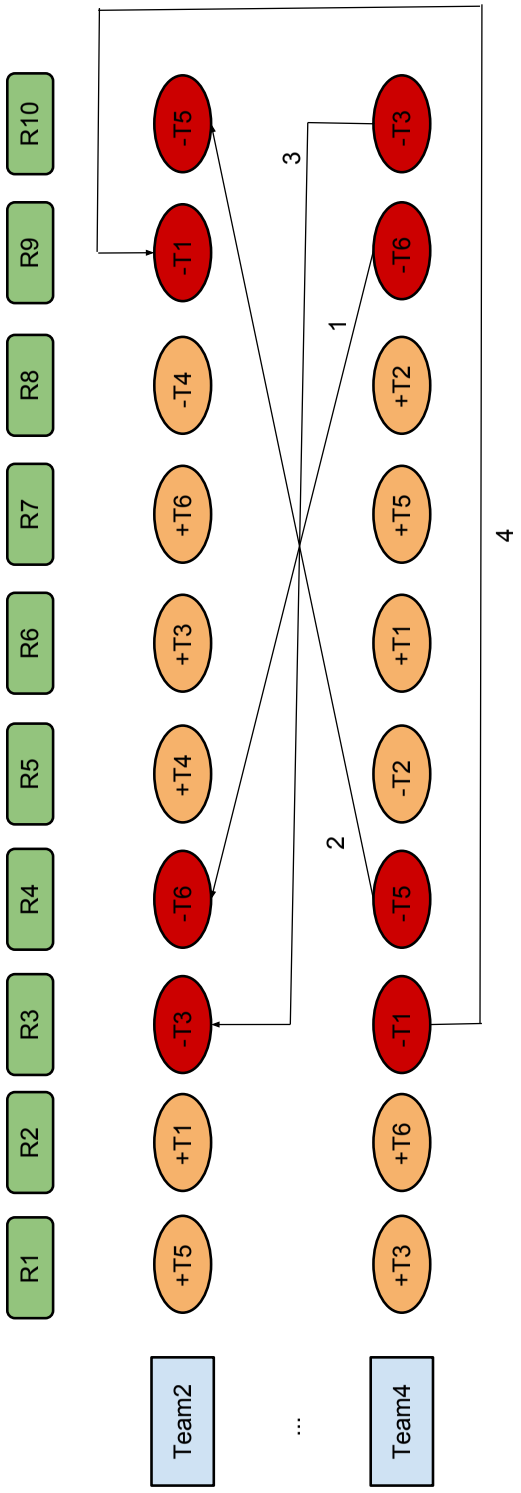


Figure 4.8: Repair chain of the *Swap-Partial-Teams* move

4.1.3 Connectivity of the neighborhoods

Having defined the neighborhoods for our embedded local-search component, we look into the next interesting question regarding the connectivity of our neighborhoods. It is indeed an important issue to investigate, since we want to know, if it is theoretically possible to reach every valid solution in the search-space with the local-moves given by our neighborhoods.

From here on, let's denote our neighborhoods with following abbreviations:

N_1: *Swap-Homes* Neighborhood

N_2: *Swap-Rounds* Neighborhood

N_3: *Swap-Teams* Neighborhood

N_4: *Swap-Partial-Rounds* Neighborhood

N_5: *Swap-Partial-Teams* Neighborhood

To the best of our knowledge, there isn't a formal proof yet proving whether the search-space is connected or not under the neighborhoods $[N_1, \ldots, N_5]$, but there have been already some interesting approaches. For instance, the authors of [17] attempted to investigate this matter with an experimental approach. They performed an experiment on the NL-instance with team-size 8, where there exists some structural information about its search-space and the optimal solution is known. For eight teams, the number of *non-isomorphic* tournament patterns is only six [36]. Using each of these patterns, they generated 6 different initial solutions and started their solvers (with various compositions of $[N_1, \ldots, N_5]$) from those initial configurations. By doing so, they observed that each of their solvers could reach the same known optimal solution for the NL_8 instance. From this observation they suggested it is somewhat likely that the search-space is indeed connected under the neighborhoods $[N_1, \ldots, N_5]$.

In this thesis, we want to propose another experimental approach, which will test the connectivity-hypothesis also on the instances without explicit structural information. We have designed our experiment as follows:

1. Generate a *random* DRR-schedule s.
2. Generate another *random* DRR-schedule s'.
3. Test if we can reach the schedule s' starting from the schedule s using only local transformations given by the neighborhoods $[N_1, \ldots, N_5]$.

The question, how we can generate a random DRR-schedule, will be discussed later in details.

Our approach to test the reachability between two random DRR-schedules is simple. Just devise a simple heuristic, which tries to minimize the *hamming-distance* between the starting schedule s and the target schedule s' by applying only the moves from $[N_1, \ldots, N_5]$. In other words, the objective function is the *hamming-distance* function, which calculates the number of differences between s and s'.

To this end, we propose a simple *Iterated Local Search* algorithm, which is described in the Algorithm 2.

Algorithm 2 ILS for testing the reachability of two configurations

```
INPUT:
        s: starting configuration
        s': target configuration

s'' = s

while !stopCondition do
   //perturbation
   for i = 1 → k do
      N = chooseRandomNeighborhood([N_1, ..., N_5])
      s''' = applyRandomMove(s'', N)
   end for

   //hill-climbing with the composite neighborhood
   //N_1 ∪ N_2 ∪ N_3 ∪ N_4 ∪ N_5 and
   //hamming distance as the objective function
   s''' = hillClimbSearch(s''')

   //acceptance criterion
   if hammingDist(s''', s') < hammingDist(s'', s') then
      s'' = s'''
   end if
end while

if hammingDist(s'', s') == 0 then
   return SUCCESS
else
   return FAIL
end if
```

We have run our experiments for team-sizes $[6, 8, 10, 12]$ testing each team-size with 1000 random DRR-schedule pairs. Interestingly for every team-size we have tested, we could always

reach 100% success rates. Even with the very basic ILS procedure, the test could report success in relatively small amount of computation-time.

In conclusion based on our own experimental results, we can only further support the hypothesis from [17] claiming that the search-space seems to be connected under the investigated neighborhoods. Obviously, it is just a hypothesis based on experimental results and doesn't prove the connectivity. The real challenge for the future would be proving the connectivity formally which is not a trivial task. It would be also interesting to experiment on the largest instances with much more runs.

4.1.4 Analysis of the neighborhoods

Before we begin with the design of an ILS algorithm for the TTP, we should first pay closer attention to the individual properties of our previously introduced neighborhoods and analyze what relationships they have with each other.

For the neighborhoods $[N_1,N_2,N_3]$, it is straightforward to see that they contain no duplicate-neighbors and that $N_1 \cap N_2 \cap N_3 = \emptyset$. Let's call $M(N_i)$ the set of N_i's moves which lead to the corresponding neighbors in N_i. Then, we say that a neighborhood N_i contains duplicates, if two *different* moves $m' \in M(N_i)$ and $m'' \in M(N_i)$ applied to a specific solution s lead to respective solutions $s' \in N_i$ and $s'' \in N_i$, where $s' = s''$ and $m' \neq m''$.

On the contrary, the partial-swapping neighborhoods N_4 and N_5 contain duplicates, which is an important fact to notice for efficiency, since we don't want to visit the same state more than once. This is not such a big deal for approaches like *Simulated Annealing*, since it doesn't search the whole neighborhood in each iteration but only chooses a random move from it. But for methods like *Hill-Climbing* or *Tabu Search*, which have to scan the whole neighborhood in every iteration, it could have considerable impacts on the performance w.r.t. the search-speed.

Given a schedule s, we denote $M(N_4)$ as a set of moves which lead to s's neighbors in N_4. Then, a move $m \in M(N_4)$ can be represented as a triple <T_i,R_j,R_k> with $j < k$, which swaps the T_i's games at the rounds R_j and R_k. If the move <T_i,R_j,R_k> induces the repair-chain RC, then all the moves <T,R_j,R_k> with $T \in RC$ will lead to same neighbor-schedule $s' \in N_4$.

For N_5, the case is the same as with N_4. If a move <R_i,T_j,T_k> from $M(N_5)$ induces repair-chain RC, then all moves <R,T_j,T_k> with $R \in RC$ are equivalent.

Let's look at one concrete example for N_4, where R_2 and R_9 are being swapped for T_2:

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$-T_2$	$+T_2$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$+T_4$	$-T_6$
T_2	$+T_5$	$+T_1$	$-T_1$	$-T_5$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$-T_6$	$-T_3$
T_3	$-T_4$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_4	$+T_3$	$+T_6$	$-T_3$	$-T_6$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$-T_1$	$-T_5$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$-T_4$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$+T_2$	$+T_1$

The repair-chain of the move in the example consists of teams $[T_1, T_2, T_4, T_6]$. One can now easily recognize that all the moves <T_1,R_2,R_9>,<T_2,R_2,R_9>, <T_4,R_2,R_9> and <T_6,R_2,R_9> will lead to same schedule:

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+T_6$	$+T_4$	$+T_2$	$+T_3$	$-T_5$	$-T_4$	$-T_3$	$+T_5$	$-T_2$	$-T_6$
T_2	$+T_5$	$-T_6$	$-T_1$	$-T_5$	$+T_4$	$+T_3$	$+T_6$	$-T_4$	$+T_1$	$-T_3$
T_3	$-T_4$	$+T_5$	$+T_4$	$-T_1$	$+T_6$	$-T_2$	$+T_1$	$-T_6$	$-T_5$	$+T_2$
T_4	$+T_3$	$-T_1$	$-T_3$	$-T_6$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$+T_6$	$-T_5$
T_5	$-T_2$	$-T_3$	$+T_6$	$+T_2$	$+T_1$	$-T_6$	$-T_4$	$-T_1$	$+T_3$	$+T_4$
T_6	$-T_1$	$+T_2$	$-T_5$	$+T_4$	$-T_3$	$+T_5$	$-T_2$	$+T_3$	$-T_4$	$+T_1$

As stated earlier, the neighborhoods $[N_1,N_2,N_3]$ are disjunctive to each other: $N_1 \cap N_2 \cap N_3 = \emptyset$. But since $[N_4, N_5]$ are the respective generalizations of $[N_2, N_3]$, there can be some possible overlappings between them.

Lets consider N_2 and N_4. If a move <T_i,R_j,R_k> $\in M(N_4)$ has a repair-chain of the length n (team-size), then it is easy to see that the both moves <R_j,R_k> $\in M(N_2)$ and <T_i,R_j,R_k> $\in M(N_4)$ are equivalent, meaning that they both lead to same neighbor-schedule. On the other hand, if a move $m \in M(N_4)$ has a repair-chain of the length 2, then there exists a move $m' \in M(N_1)$ such that m and m' are equivalent. So as you can see, it is possible that $N_2 \cap N_4 \neq \emptyset$ and N_1 is in fact a subset of N_4. Again for N_3 and N_5, the situation is similar as with N_2 and N_4, but note also that $N_1 \cap N_5 = \emptyset$ because of the special precondition for N_5.

It's important that we identify and take notice of these properties, since they offer useful information we can exploit to iterate over the neighborhoods more efficiently. A similar but much more detailed analysis regarding the neighborhoods can also be found in [17].

4.1.5 Incremental Evaluation

One of the most performance-critical part of a local-search algorithm is the *evaluation* of the neighboring solutions. After definition of the neighborhoods, the very first thing one should look

into is the definition of an *incremental* (delta) evaluation function, which efficiently evaluates only the changed parts of the neighbor-solution instead of reevaluating it from scratch.

For the TTP, a naive approach of evaluating everything from scratch can be costly, since we'll have $O(n^2)$ performance cost each time we evaluate a neighbor-solution. Therefore one of our contribution goals in this thesis is to investigate if it is possible to define an efficient incremental evaluation procedure, which is better than the naive approach.

A valid solution for the TTP is a schedule, which satisfies all three constraints

C_1: *Double-Round-Robin* constraint

C_2: *AtMost* constraint

C_3: *NoRepeat* constraint

and the optimal solution for the TTP is the schedule, which minimizes the travel-distances of each involved teams.

A DRR-schedule satisfies always the C_1 constraint but can possibly violate the C_2 and C_3 constraints. Therefore we call the constraint C_1 a *hard-constraint* and the constraints C_2 and C_3 *soft-constraints* (see also [4]). Please recall that the search-space defined by our neighborhoods encompasses only valid DRR-schedules, i.e. only schedules, which already satisfy C_1 constraint. So in order to evaluate a TTP solution candidate in our search-space, we need values of three cost-components:

- the travel-distance
- the number of *AtMost* violations
- the number of *NoRepeat* violations

4.1.5.1 Delta-Evaluation of the Traveling Distance

If we imagine the schedule as a two-dimensional matrix S (which is indeed our chosen data-structure), where $S_{i,j}$ represents the game of the team T_i at round R_j, a change in the schedule S equals an update of a matrix-entry, i.e. deleting the old game and inserting a new game. All the moves defined by our neighborhoods can be seen as a *sequence* of those changes. For instance, the *Swap-Homes* move causes exactly 4 updates in the schedule-matrix, since its move-strength is 4.

Calculating the delta-value of the travel-distance, caused by an single entry update, is relatively easy. One just has to subtract the distance value of the old deleted game and add the distance value of the newly inserted game. The procedure is given in Algorithm 3. We also want

to notice that some special cases (regarding the first and last round) are not considered here for the sake of brevity, but they can be, of course, augmented in a trivial manner.

Algorithm 3 Calculate delta-travel-cost for single change

```
INPUT:
        S: schedule
        T_i: team
        R_j: round
        G: new game

//let's say that the function distance(t, g_1, g_2)
//calculates the distance between two games g_1 and g_2 for the team t

δ = 0

//delete the old game
δ = δ - distance(T_i, S[T_i][R_j − 1], S[T_i][R_j]) + distance(T_i, S[T_i][R_j], S[T_i][R_j + 1])
//insert the new game
δ = δ + distance(T_i, S[T_i][R_j − 1], G) + distance(T_i, G, S[T_i][R_j + 1])

return δ
```

At first glance, the delta-function looks to be easily definable by calculating all the changes caused by a move individually using the procedure given above. Unfortunately this doesn't work quite yet. Too see why, just consider a local-move, which introduces changes to the two *consecutive* games of a team. If we were about to calculate the changes individually one after another, the calculated delta-value wouldn't be correct, because the second calculation wouldn't be able to "see" the new changed game from the first update.

The problem described above can be trivially solved if we actually update the intermediate states of the schedule as we go along, so that the previous changes would become "visible" to the subsequent delta calculations. But this comes with a performance cost. Not only we have to perform the actual updates, but we also have to undo the changes (or if you work on a copy, the copying itself would be actually more expensive than undoing the move) in order to visit the next neighbor.

But there is a better way to address this problem. We can easily generalize the previous idea of calculating the delta-value for a *single change*. Instead of working with one single change at a time, we can actually work with one *contiguous change-block* at a time. A contiguous change-block is an *interval of consecutive games* of a team affected by a move and can be represented with a triple $< T_i, R_i, R_j >$, where T_i is the team and R_i, R_j are the beginning- and end-round of the interval.

An example of a contiguous change-block $< T_4, R2, R4 >$ for a schedule with 6 teams is given in Figure 4.9. As you have probably already noticed, adjusting the Algorithm 3 to calculate the delta-value of a contiguous change-block is not difficult. The adjusted procedure is given in Algorithm 4.

Obviously the complexity of the Algorithm 4 is $O(m)$, where m is the length of the change-block. By applying this procedure for every affected change-block sequentially, we can calculate the total delta-value of the travel-distance in $O(p)$, where p is the *total number* of changes.

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
...										
T_4	$+T_3$	$-T_1$	$-T_3$	$-T_6$	$-T_2$	$+T_1$	$+T_5$	$+T_2$	$+T_6$	$-T_5$
...										

Figure 4.9: A contiguous change-block

Algorithm 4 Calculate delta-travel-cost for a contiguous change-block

INPUT:
    S: schedule
    T_i: team
    R_i: beginning of the change-block
    R_j: end of the change-block
    $G[]$: sequence of new games

//let's say that the function *distance(t, g_1, g_2)*
//calculates the distance between two games g_1 and g_2 for the team t

δ = 0
k = 0

δ = δ - distance(T_i, $S[T_i][R_i - 1]$, $S[T_i][R_i]$)
δ = δ + distance(T_i, $S[T_i][R_i - 1]$, $G[0]$)

while $R_i + k < R_j$ **do**
    δ = δ - distance(T_i, $S[T_i][R_i + k]$, $S[T_i][R_i + k]$)
    δ = δ + distance(T_i, $G[k]$, $G[k + 1]$)

    k = k + 1
end while

δ = δ - distance(T_i, $S[T_i][R_j]$, $S[T_i][R_j + 1]$)
δ = δ + distance(T_i, $G[k + 1]$, $S[T_i][R_j + 1]$)

return δ

4.1.5.2 Delta-Evaluation of the *NoRepeat* violations

Calculating the delta-value for the *NoRepeat* cost-component is as straight forward as it is for the travel-distance. Again we can break down the changes caused by a local move in contiguous change-blocks, which are described in the previous section, and calculate their delta-values independently. The algorithm for this procedure is similar to the Algorithm 4 and is presented in the Algorithm 5 without further explanations.

Algorithm 5 Calculate delta-NoRepeat-cost for a contiguous change-block

```
1:  INPUT:
2:        S: schedule
3:        T_i: team
4:        R_i: beginning of the block
5:        R_j: end of the block
6:        G[]: sequence of new games
7:
8:  //lets say that the function noRepeatViolation(g_1, g_2)
9:  //returns 1 if the two games g_1 and g_2 have same opponents
10: //otherwise 0.
11:
12: δ = 0
13: k = 0
14:
15: δ = δ - noRepeatViolation(S[T_i][R_i − 1], S[T_i][R_i])
16: δ = δ + noRepeatViolation(S[T_i][R_i − 1], G[0])
17:
18: while R_i + k < R_j do
19:    δ = δ - noRepeatViolation(S[T_i][R_i + k], S[T_i][R_i + k])
20:    δ = δ + noRepeatViolation(G[k], G[k + 1])
21:
22:    k = k + 1
23: end while
24:
25: δ = δ - noRepeatViolation(S[T_i][R_j], S[T_i][R_j + 1])
26: δ = δ + noRepeatViolation(G[k + 1], S[T_i][R_j + 1])
27:
28: return δ
```

An attentive reader might have noticed that the code given above is not fully optimized yet. If we consider the moves defined by our neighborhoods, all of them are moves, which incorporate sequence of swapping. So in case of *NoRepeat* cost-component, when we swap two entire change-blocks, the only positions, that can influence the delta-value, are the beginning and

T/R	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	$+6$	-2	$+4$	$+3$	-5	-4	-3	$+5$	$+2$	-6
T_2	$+5$	$+1$	-3	-6	$+4$	$+3$	$+6$	-4	-1	-5
T_3	-4	$+5$	$+2$	-1	$+6$	-2	$+1$	-6	-5	$+4$
T_4	$+3$	$+6$	-1	-5	-2	$+1$	$+5$	$+2$	-6	-3
T_5	-2	-3	$+6$	$+4$	$+1$	-6	-4	-1	$+3$	$+2$
T_6	-1	-4	-5	$+2$	-3	$+5$	-2	$+3$	$+4$	$+1$

This schedule with 6 teams yields following homeaway-sequence-table, where [] denotes home-sequence and () denotes away-sequence

T_1	[1,1]	(2,2)	[3,4]	[3,4]	(5,7)	(5,7)	(5,7)	[8,9]	[8,9]	(10,10)
T_2	[1,2]	[1,2]	(3,4)	(3,4)	[5,7]	[5,7]	[5,7]	(8,10)	(8,10)	(8,10)
T_3	(1,1)	[2,3]	[2,3]	(4,4)	[5,5]	(6,6)	[7,7]	(8,9)	(8,9)	[10,10]
T_4	[1,2]	[1,2]	(3,5)	(3,5)	(3,5)	[6,8]	[6,8]	[6,8]	(9,10)	(9,10)
T_5	(1,2)	(1,2)	[3,5]	[3,5]	[3,5]	(6,8)	(6,8)	(6,8)	[9,10]	[9,10]
T_6	(1,3)	(1,3)	(1,3)	[4,4]	(5,5)	[6,6]	(7,7)	[8,10]	[8,10]	[8,10]

Figure 4.10: A homeaway-sequence table

end of the block. This makes it actually unnecessary to recompute the delta-value for the whole block, but it would be sufficient to look only at the beginning and end of the change-block.

But nevertheless, these are low-level optimization issues, which can be fine-tuned during the actual implementation. Either way, the complexity of the Algorithm 5 is *linear* to the length of the change-block and again we can calculate the total delta-value of the *NoRepeat* cost-component in time linear to the number of *changes*.

4.1.5.3 Delta-Evaluation of the AtMost violations

So far, calculating delta-values for the travel-distance and the *NoRepeat* cost-component has been a relatively straight-forward affair. But for the final cost-component, namely the *AtMost* cost-component, more complex strategies are needed as we will see it shortly.

In order to evaluate the number of *AtMost* violations, we need to know all the *sequences* of the *consecutive* home-games and away-games for each team. We call such a sequence simply as *homeaway-sequence*. To save the information of the *homeaway-sequences*, we have to define a secondary data-structure, which we will call as *homeaway-sequence-table*.

Maybe the best way to explain it is to look at one concrete example. The homeaway-

sequence-table shown in the Figure 4.10 should be self explanatory. As you can see, given the schedule S and its homeaway-sequence-table H, the entry $H[T_i][R_j]$ contains the *homeaway-sequence*, to which the game $S[T_i][R_j]$ belongs. A *homeaway-sequence* is represented as a pair of numbers, whose first number is the round, where the sequence begins, and the second number the round, where it ends.

Using the homeaway-sequence-table H, the calculation of the delta-value for a *single* change is now simple. It should be also clear that a change only occurs if the the old game and the new game have *different* home-away assignments, i.e. there is no change if the old game and the new game are both away-games or both home-games.

When calculating the delta-value for the single change at $S[T][R]$, we distinguish four different cases. We denote the respective homeaway-sequence as $(i, j) = H[T][R]$ and its neighboring sequences as $(g, h) = H[T][i - 1]$ and $(k, l) = H[T][j + 1]$. We also define a function $atMost(i, j)$, which calculates the number of the *AtMost* violations for the given homeaway-sequence (i, j).

1. $i = R = j$ (R is both the beginning and end of the interval):
   $\delta = atMost(g, l) - atMost(g, h) - atMost(k, l)$

2. $i < R < j$ (R is in the middle of the interval):
   $\delta = atMost(i, R - 1) + atMost(R + 1, j) - atMost(i, j)$

3. $i = R < j$ (R is the beginning of the interval):
   $\delta = atMost(g, i) - atMost(g, h) - atMost(i, j) + atMost(i + 1, j)$

4. $i < R = j$ (R is the end of the interval):
   $\delta = atMost(j, l) - atMost(k, l) - atMost(i, j) + atMost(i, j - 1)$

As with the travel-distance, the unpleasant difficulty arises, when we have to deal with *multiple* changes. If we want to *chain* the calculation for the single change, then we actually have to *update* all the intermediate states of the homeaway-sequence-table H.

Updating the intermediate states of H can cause a serious performance-penalty if done in a naive manner. But luckily, there is an efficient way to make the intermediate changes "visible" to the subsequent calls of the single delta-calculation. In contrast to the previous two cost-components, we're going to work here with a complete *list of changed rounds* for each team instead of change-blocks. In addition, we demand as a precondition that the list of changed rounds is *sorted* in ascending order. Then we just iterate over the list calculating the delta-cost for each changed round with the procedure described above.

Since we go through the *sorted* list of rounds, the only homeaway-sequence updates, which have to be "visible" to the next calculation for the round R, are the 2 "rightmost" homeaway-sequences *up to* the round R. The concrete details will become clearer when looking at the implementation given in Algorithm 6. We want to notice again that some special-cases regarding the first and last round are omitted in Algorithm 6 for the sake of brevity.

Clearly, the complexity of the Algorithm 6 is *linear* to the number of changed rounds, if the list of changed rounds is already sorted. But if you actually need to sort it, then the complexity will rise to $O(m \log m)$ (e.g. for the merge-sort) considering the sorting-cost, where m is the length of the change-list.

4.1.5.4 Summary

In this section, the strategies for incremental-evaluation of the delta-values are introduced for all three cost-components of a TTP-solution

- Travel-distance
- NoRepeat violations
- AtMost violations

The importance of incremental evaluation is critical for all kinds of metaheuristics. Some may argue that if a move is too disruptive, then it is better to calculate everything from scratch. This may be true for small-instances, but nevertheless you will certainly notice more and more difference with increasing size of the instances.

Therefore we hope that our effort to find efficient ways of incremental evaluation would be a positive contribution to further researches using similar neighborhoods.

Algorithm 6 Calculate delta AtMost violation for all changed rounds of a team

```
1: INPUT:
2:         S: schedule
3:         H: venue-sequence table
4:         T_i: team
5:         R[]: sorted list of changed rounds
6:
7: //lets say that the function atMost(i, j)
8: //returns the number of AtMost violations of the given homeaway-sequence
9: //(i,j)
10:
11: haSeq1 = (∞, ∞)
12: haSeq2 = (∞, ∞)
13: δ = 0
14:
15: for r ∈ R do
16:   if r ∈ haSeq2 then
17:     (i, j) = haSeq2
18:     (g, h) = haSeq1
19:     (k, l) = H[T_i][j + 1]
20:   else
21:     (i, j) = H[T_i][r]
22:     if i − 1 ∈ haSeq2 then
23:       (g, h) = haSeq2
24:     else
25:       (g, h) = H[T_i][i − 1]
26:     end if
27:     (k, l) = H[T_i][j + 1];
28:   end if
29:
30:   if i = r = j then
31:     δ = atMost(g, l) - atMost(g, h) - atMost(k, l)
32:     haSeq2 = (g, l)
33:     haSeq1 = H[T_i][g − 1]
34:   else if i < r < j then
35:     δ = atMost(i, r − 1) + atMost(r + 1, j) - atMost(i, j)
36:     haSeq2 = (r + 1, j)
37:     haSeq1 = (r, r)
38:   else if i = r < j then
39:     δ = atMost(g, i) - atMost(g, h) - atMost(i, j) + atMost(i + 1, j)
40:     haSeq2 = (i + 1, j)
41:     haSeq1 = (g, i)
42:   else if i < r = j then
43:     δ = atMost(j, l) - atMost(k, l) - atMost(i, j) + atMost(i, j − 1)
44:     haSeq2 = (j, l)
45:     haSeq1 = (i, j − 1)
46:   end if
47: end for
48:
49: return δ
```

4.2 Iterated Local Search for the TTP

We now have discussed all the necessary preliminaries to tackle the "main quest" of designing an ILS algorithm for the TTP.

We find that designing an ILS algorithm is an *iterative* process, where we start from very basic choices of the components and improve them based on the insights gained from analyzing the previous choices.

To recap, the main components of the ILS framework are:

- Initial solution
- Local Search
- Perturbation
- Acceptance Criterion

The local-search component is the main driving part , which is responsible for finding the local optima, whereas the components *Perturbation* and *Acceptance Criterion* are responsible for escaping the local optima balancing the search between *intensification* and *diversification*.

Normally, one would assume that using a high-quality local search component is always better, but you should also be aware of the high computation cost it comes with. Sometimes it can be also very effective to use a simple but very *fast* embedded local-search, which, for example, has been demonstrated in [25]. Tuning the perturbation and acceptance criterion components are also very essential for the success of the ILS. The perturbation component should introduce enough changes to maintain adequate chance of escaping local optima, whereas too strong perturbation could destroy too much of the "good properties" found in previous searches. The acceptance criterion normally should favor the better results, but doing it in a too biased way can lead the algorithm to be stuck permanently in "difficult" local optima.

In the next following sections, we describe our choices for the individual components.

4.2.1 Basic ILS for the TTP ($TTILS_{basic}$)

4.2.1.1 Initial Solution

In order to kick off the ILS algorithm, we first need a valid initial solution, which is a valid double-round-robin schedule. There already exist different strategies to generate an initial DRR-schedule. For instance, see [4],[17], [20] and [26].

Our first choice was initially a *semi-random* strategy (similar to the one used in [17]), which works as follows:

1. For a given number of teams n, find an *1-factorization* [23] of a complete graph K_n, i.e. partition the graph K_n into $n-1$ *1-factors* each containing $n/2$ edges, where the *1-factors* represent the rounds and the edges represent the games between two teams. The resulting *1-factorization* of K_n is a valid *single-round-robin* schedule-pattern for n teams. We call it a pattern, because the nodes of K_n can be seen as placeholders, to which concrete teams can be assigned afterwards. So, the *1-factorization* of K_n can be completed to a SRR-schedule by determining home- and away-states for each game and assigning concrete teams to the nodes of K_n,

   Now we can generate a second SRR-schedule by simply mirroring the first one and by appending the second SRR-schedule to the first one, we get a valid *mirrored double-round-robin* schedule. The *1-factorization* can be easily computed deterministically in linear time [13].

2. After generation of a valid *mirrored double-round-robin* schedule S, we can now apply several *random* moves from $[N_1,\ldots,N_5]$ to randomize S further.

But we realized that this strategy wasn't actually the best choice if we consider its "semi-randomness". Since our generation of the initial *1-factorization* is a deterministic procedure ($canonicalpattern$) and we don't know with certainty the connectivity of our neighborhoods $[N_1,\ldots,N_5]$, it is desirable to devise a strategy to generate "more" random initial solutions.

Therefore we propose a second strategy which is based on a simple heuristic method. The method works as follows:

1. Generate a random initial schedule S, such that every team T_i plays each team $T_j (j \neq i)$ exact twice, once at home and once in the opponent's city. This is easy to achieve, since we only have to assign each team a random permutation of the sequence $[-T_n, \ldots, T_n] \setminus [-T_i, T_i]$. Obviously the resulting schedule can violate the DRR-constraint.

2. In the second phase, we try to eliminate the violations of the DRR-constraint from the initial schedule S. For that purpose, we devise a simple *Hill-Climbing heuristic* using a local-move, which simply swaps two games of a team. You can imagine this move as our *Swap-Partial-Rounds* move, only without applying the repair-chain afterwards. The objective function is the number of violations violating the DRR-constraint and the algorithm stops if the number of violations drops to zero. Despite of its simplicity, the proposed heuristic is able to find a valid DRR-schedule fairly quickly.

At this point, we also want to mention that we decided not to employ any kinds of greedy constructing mechanism during the generation of the initial solution, since the precedent re-

searches make us to believe that the greediness in the initial solution is not of a significant importance in the long run for the TTP. So our initial schedule is a fully random schedule generated using the second “full-random" method.

4.2.1.2 Local Search

Here, we describe our basic choice for the embedded local-search component. We have to keep in mind that the quality alone is not the sole critical decisive factor to consider, but the effectiveness of the local-search component also depends on its computation-time to reach high quality solutions.

It is relatively often the case that a simple *Hill-Climbing* heuristic is used as the embedded local-search component in the ILS framework. The main skeleton of the *Hill-Climbing* heuristic for the *minimization* problem is given in the Algorithm 7.

Algorithm 7 Hill Climbing

```
1: INPUT:
2:         x: initial solution
3:         f: evaluation function
4:         N: neighborhood structure
5:
6: stop = false;
7:
8: while !stop do
9:    x' = y ∈ N(x) with f(y) ≤ f(y'), ∀y' ∈ N(x)
10:   if f(x') < f(x) then
11:      x = x'
12:   else
13:      stop = true;
14:   end if
15: end while
16:
17: return x;
```

As you can see, the *Hill-Climbing* heuristic improves the current solution, as long as it can find improving neighbors. Finding an improving neighbor for the next iteration can be done in two different ways:

- Find *first* improving neighbor
- Find *best* improving neighbor

The first strategy continues with next iteration as soon as the first improving neighbor is found, whereas the second one searches the full neighborhood and continues with the best improving neighbor. Obviously if the neighborhood is very large, using the *first improving* strategy could speed up things a little bit.

In order to apply the *Hill-Climbing* heuristic to the TTP, we first have to define an appropriate neighborhood for it. In the previous sections, we've introduced five different neighborhoods $[N_1, \ldots, N_5]$ for the TTP. At first, our initial thought was to combine all the five neighborhoods into one big composite neighborhood. But simply joining them would be careless, since we've seen that our neighborhoods $[N_1, \ldots, N_5]$ are *not disjunct* to each other. After some initial experiments, we've decided to go with the following composition:

$$\mathbb{N} = N_2 \cup N_3 \cup N_4^* \cup N_5^*$$

,where N_4^* is the N_4 neighborhood reduced by those moves, whose repair-chain has the length n (team-size) and N_5^* is the N_5 neighborhood reduced by the moves, whose repair-chain has the length $2n - 4$. We omitted the neighborhood N_1 completely, since it is the subset of N_4.

The obvious advantages of embedding the *Hill-Climbing* heuristic into the ILS are:

- easy, straight-forward implementation
- no critical parameters to optimize
- fast speed, if the ILS has already reached a high-quality region

But in contrast, we should also point out some of its significant disadvantages, which have to be compensated by other components:

- poor local optima quality compared to more advanced local searches
- can easily be stuck in local optima, if the perturbation is not strong enough

Despite of some disadvantages, we have decided to use *Hill-Climbing* as the embedded local-search for $TTILS_{basic}$ because of its simplicity regarding the implementation and parameter-tuning. Also, some researches show that even with a simple embedded local-search like *Hill-Climbing* very competitive results can be achieved ([25],[28]).

4.2.1.3 Perturbation

A simple way to perform perturbation is to apply *high-order* random moves, i.e. apply a random move from a randomly selected neighborhood for a predefined k times. This is exactly how our initial perturbation is implemented for $TTILS_{basic}$.

From the five neighborhoods $[N_1, \ldots, N_5]$, one is chosen randomly, a random move is generated from it and applied. This step is repeated k-times, where k is a parameter to be defined. Since there are some overlapping moves in the neighborhoods, one should also take care that the previous perturbation step is not reversed by the successive equal move.

Another important question is how to choose the value for the parameter k. If it is too big, the perturbation will deteriorate the solution quality too much, whereas if it is kept too small, the effectiveness of escaping local optima will be reduced. Instead of keeping it constant, we decided to make k a *dynamic* parameter, which will vary during the search in some predefined interval (k_{min}, k_{max}). Our approach is here very simple. We just increment k by 1 until it reaches k_{max} and then it is reset to k_{min}.

This kind of perturbation has been already successfully applied in other ILS-applications ([25], [28]) as well and despite of its simplicity the results were very good when it was combined with right components.

4.2.1.4 Acceptance Criterion

Initially we've decided to experiment with 2 basic acceptance criteria for the $TTILS_{basic}$:

- *Better* acceptance criterion
- *Random Walk* acceptance criterion

As the name already suggests, the *Better* acceptance criterion accepts only *improving* solutions.

$$\text{Better}(s', s'') = \begin{cases} s' & \text{if Cost}(s') < \text{Cost}(s'') \\ s'' & \text{otherwise} \end{cases}$$

This clearly advocates strong intensification and it will push the search *quickly* towards a local optimum. One major drawback of this strategy is that it is very difficult to escape difficult local optima, once the search is stuck there.

On the other hand, the *Random Walk* strategy always accepts *new* solutions found by the local-search component.

$$\text{RW}(s', s'') = s''$$

In contrast to the *Better* acceptance criterion, this strategy strongly advocates diversification. With this acceptance criterion, the ILS will be able to explore many different search regions, but it will have difficulties to "go deeper" in one particular promising region.

Our initial experiments suggest that the *Better* acceptance criterion seems to perform better than the *Random Walk* acceptance criterion for $TTILS_{basic}$.

4.2.1.5 Objective function and Strategic Oscillation

The main objective in solving the TTP is to find a schedule, which minimizes the total travel-distance satisfying at the same time various constraints. Our neighborhoods are defined in a way that certain *soft-constraints* can be violated during the search and in order to push the search towards valid solutions we have to add some penalties to those violations in the main objective function.

Therefore we define the objective function f, which adds the *weighted* number of violations to the total travel-distance. The objective function f can be formally expressed as

$$f(S) = \sum_{T=1}^{n} dist(S,T) + \omega * (atmost(S,T) + norepeat(S,T))$$

,where $dist()$, $atmost()$ and $norepeat()$ are the respective cost-component functions of a single team.

As shown in previous researches using the same neighborhoods, it is of great importance to spend some time in *infeasible* regions during the search, i.e. we have to alternate the search between *feasible* and *infeasible* regions. Similar to [4] and [17], we can achieve that by means of a mechanism resembling the *strategic oscillation* incorporated successfully into the Tabu-Search [14]. The idea is to increase and decrease the weight parameter ω based on search history, so that the search spends an appropriate amount of time in both the feasible and infeasible regions.

We can build this strategy into the ILS by examining the feasiblity of the solutions produced by the local-search component during k iterations. If the majority of them are infeasible/feasible, then we can increase/decrease the weight ω by multiplying/dividing it with δ, thus penalizing the violations stronger/weaker. For the simplicity, we've set k to 1.

The initial value of ω can be set initially to any appropriate value (e.g. the average distance of the distance matrix), since the weight will be adjusted "reactively" during the search.

4.2.1.6 Discussion

After some preliminary experiments with $TTILS_{basic}$, we could observe that even the simple basic "ILS-configuration" was capable of producing pretty good results in short amount of time. But looking closely, some obvious problems emerged to the surface.

First, we noticed that the perturbation of $TTILS_{basic}$ was too disruptive causing the local-search component to spend long time to reach another local-optimum. The disturbance is the worst when multiple consecutive perturbation moves are selected from the neighborhoods N_2 and N_3.

Second, both the *Better* and the *Random Walk* acceptance criteria were "too biased" or "too random". When comparing the two directly, we could notice that the *Better* acceptance criterion

was in general the superior one, but still the search gets too easily trapped in local optima with *Better* acceptance criterion.

And finally, we find that the initial composite neighborhood for the local-search component is still seems to be too big. Since the embedded *Hill-Climbing* heuristic is not so powerful and its main advantage lies in *fast sampling* of the local optima, we should try to reduce the size of the neighborhood as much as possible without losing too much quality for the local-optima.

4.2.2 Tuning the basic ILS for the TTP ($TTILS_{opt}$)

In this section, we optimize the $TTILS_{basic}$ by trying to improve the problematic issues described in the previous section resulting in final version of our ILS algorithm $TTILS_{opt}$.

4.2.2.1 Local Search

Regarding the local-search, we keep using the *Hill-Climbing* heuristic as our embedded local-search component. But we experimented with further possibilities to composite the neighborhoods and we noticed that omitting the neighborhoods N_2 and N_3 didn't have too much of a negative impact on the quality of the local-search. This also has been confirmed somewhat by the results given in [17], where they used similar composition-neighborhood for their best solver.

So our new reduced composite neighborhood for $TTILS_{opt}$ is:

$$\mathbb{N} = N_4 \cup N_5$$

Also note that we've removed the constraints on maximum repair-chain-length for N_4 and N_5.

4.2.2.2 Perturbation

The main idea of making *high-order* random moves for perturbation remains the same as for the $TTILS_{basic}$, but we've further constrained the neighborhoods from which a random move can be selected. Concretely, we have completely discarded the moves from $[N_2, N_3]$. The length of these moves are just too long and even a small number of these moves can cause too much disruption in the solution quality. We also further constrained the "selectable" moves from the neighborhoods N_4 and N_5 to those with repair-chain length equal or less than 6.

In summary, the perturbation component of $TTILS_{opt}$ selects random moves from

$$\mathbb{N} = N_1 \cup N_4^* \cup N_5^*$$

,where N_4^* and N_5^* are the reduced "versions" of N_4 and N_5, which contain only moves with repair-chain length equal or less than 6.

4.2.2.3 Directed Perturbation

In this section, we want to briefly describe a more advanced idea for the perturbation component, which we didn't use in the end due to the poor initial results. Although we couldn't fully investigate the idea, we find it worth mentioning, since we believe that properly tuned it could be an interesting approach for the future researches.

The main idea was to apply the local-search component for the perturbation with the *modified objective* function, so that the local-search concentrates only on the *subset* of the teams. Concretely, the "perturbating" steps for the given schedule S would be:

1. Choose m teams randomly from $T_1, \ldots, T_n$
2. Modify the objective function f to f', which optimizes only the chosen teams
3. Apply the local-search component to S with modified objective function f'
4. Return the local-optimum found in the previous step as the perturbated solution

The rationale behind this idea can be summarized as follows:

- Optimizing a subset of the teams will push the corresponding parts of the schedule structure towards optimum, thus introducing new different "good" attributes into the schedule without deteriorating the overall quality too much.
- The perturbation can not be easily undone by the embedded local-search component with the original objective function.

All of this sounds quite promising and for future works, it would be interesting to investigate, if this idea can be successfully incorporated into the perturbation component.

4.2.2.4 Acceptance Criterion

To find an appropriate balance between the two previous basic variants, we consider a simulated annealing type acceptance criterion, which we will denote as the *Large Step Markov Chains* (LSMC) acceptance criterion (as in [21]). This type of acceptance criterion was successfully applied in one of the first ILS algorithms proposed for the TSP [22] and it significantly improved the performances of the ILS algorithms for the TSP.

The LSMC acceptance criterion has been already briefly introduced in Chapter 3. To recap, the LSMC acceptance criterion tries to find a balance between intensification and diversification

by allowing the acceptance of qualitative inferior solutions with certain probabilities, which will get lower with decreasing qualities. The acceptance probability p can be formally expressed as

$$p = e^{\frac{Cost(s') - Cost(s'')}{T}}$$

,where T is the so-called temperature, which regulates the degree of "punishment" of inferior solutions. Obviously, the search becomes more diversifying with high temperature T and if T is low, then the search is more intensifying.

In order to "regulate" the transitions between diversification and intensification, we have to decide for a "cooling" scheme how the temperature is lowered during the search. For $TTILS_{opt}$, we have chosen a *non-monotonic* cooling scheme, where the temperature T is lowered during the search by multiplying it with $0 < \delta < 1$ at each iteration until it seems the intensification is no longer useful. Then the temperature is reset to T_{max} to allow diversification again for a limited time, where T_{max} is a parameter to tune. This would be most effective, if we could do this in an automatic manner taking the search history into account.

Therefore we have devised following strategy to determine an appropriate moment for the next diversification phase: We simply check in every λ iterations how many worse solutions got accepted. If the number of accepted worse solutions is smaller than α, then we assume that the intensification is too strong and resets the temperature to T_{max}. In this way, we can end the intensification phase in a "timely" manner taking the search history into account. This idea was inspired from the LSMC application in [28].

As it turns out, the parameter T_{max} is one of the most important parameters to tune in order to reach good performance. In Chapter 5, we'll discuss which initial candidate values for T_{max} we've experimented with and the reason why we've chosen them.

The results from the initial experiments also suggest that *fast* temperature cooling-rate seems to be more favorable, i.e. it seems to be better when the diversification phase is rather short and the intensification is quickly resumed.

4.2.2.5 Additional Features

In the course of our preliminary experiments, we have noticed one particular problem when applying the LSMC with *non-monotonic* cooling schedule. Concretely, we often could observe the problem that the search systematically "wanders off" from very promising regions, when it is desirable to stay there longer and search more thoroughly.

For example, let's consider the following scenario, where the search has reached a very high-quality solution S and the temperature is reset to T_{max}. Then due to diversification, a (much) worse solution S' gets accepted, which also happens to be a "difficult" local-optimum (since we are in a high-quality region), where the search is stuck until the next temperature reset occurs.

After the temperature is reset again while the search is stuck at S', now a new difficult local-optimum S'', which is worse than S', gets accepted. One can now imagine where this is going. If this happens multiple consecutive times, then the search will stepwise "back off" from the promising region around the first high-quality solution S.

To address this problem, we incorporate a *restoring mechanism*, which restores the *global best* solution found so far when there is a sign that the search wanders off too much from the current promising region. The "restoring" can be triggered, when the difference between the global best solution and currently accepted solution exceeds a certain threshold. To keep things simple, in $TTILS_{opt}$ we just restore the global best solution every time when the temperature is reset to T_{max}.

Obviously one also has to consider the danger that the repeated restoring of global best solution can cause the search to be trapped in the same search-region forever. Therefore, we use additionally a *soft restarting* mechanism as means for "macro-diversification", when we think that the search is trapped for too long in the same region. To recognize a potential stalemate of the search, we count the number of *temperature resets* without improvements for the current best solution and if the number exceeds certain threshold, we simply restart the algorithm. In doing so, we don't start entirely from scratch, but we construct the initial solution for the next round by applying *high-order* random perturbation-move to the current best solution, so that some of the good properties are passed on.

Another noteworthy detail to mention is that we actually keep two global best solutions, one for the feasible solutions and the other for the infeasible solutions. Keeping a "promising" infeasible solution is valuable, since it can lead to very good feasible solutions in next iterations. But we have to make sure that they are close to the feasibility, i.e. the number of violations is within a certain limit, otherwise the probability it can be "repaired" to the equally good feasible one is very low. Therefore, we manage only best infeasible solutions, whose total number of violations is smaller than σ.

Since we keep both globally best feasible and globally best infeasible solutions, we have to choose between the two global best solutions when we trigger the restoring-mechanism. In our approach, we simply take one with 50% probability.

Part III

CHAPTER 5

Experimental Results

In this chapter, we present the computational results of $TTILS_{opt}$ on selected benchmark instances from [3]. The code is implemented from scratch in C++ and compiled using `g++ (SUSE Linux) version 4.5.1 20101208` with flag `-O3`. All experiments are carried out on a Linux machine with `Intel(R) Xeon(R) CPU E5345 @ 2.33GHz` and 47GB main memory. It should be noted that the implementation of $TTILS_{opt}$ is strictly single threaded and only one core is used during the computation. The memory usage of $TTILS_{opt}$ is not noteworthy and occupies less than 1GB during the entire computation.

5.1 Benchmark instances

Trick, who originally proposed the Traveling Tournament Problem, personally administer the TTP benchmark instances on his website [3].

One of the very first benchmark instances published by Trick was the family of *NL-x* instances based on real data of the US National Baseball League, where the x is an even number of teams. The family of *NL-x* instances is probably the most well-researched TTP benchmark-family and virtually all researches studying the TTP publish their computational results with *NL-x* instances.

Over the years, more and more instances are added, where the *Super-x* and *Galaxy-x* families added by Uthus belong to the most recent ones. The *Super-x* instances are from the Super 14 Rugby League, composed of teams from New Zealand, Australia, and South Africa. The *Galaxy-x* family uses distances that arise from taking distances between stars (in light years).

For our computational experiments, we've used all three families with following instance-sizes.

- *NL-x* family with team-sizes from 6 to 16.
- *Super-x* family with team-sizes from 6 to 14.
- *Galaxy-x* family with team-sizes from 6 to 16.

5.2 Parameter setting

Despite of the relatively simple ILS-components, there are actually some parameters to define for the $TTILS_{opt}$. The embedded local-search component is a simple *Hill-Climbing* heuristic without any parameters to tune. The parameters for the perturbation component define the interval in which the perturbation strength varies *dynamically*. At this point, one should notice that $TTILS_{opt}$ tries to achieve diversification mainly by occasionally accepting worse solutions and not so much by very high perturbation strength. Therefore the $TTILS_{opt}$'s main "tool" for escaping local-optima is the LSMC acceptance criterion, whose parameters have critical impacts on the performance and we've spent considerable effort determining best values for them.

In summary, there are following parameters to determine for $TTILS_{opt}$:

k_{min}: the minimum perturbation strength

k_{max}: the maximum perturbation strength

T_{max}: the maximum(initial) temperature for LSMC acceptance criterion

λ: number of iterations before the next worse-solution-acceptance-rate is checked

α: if the number of accepted worse solutions is smaller than α, then the temperature is reset to T_{max}

δ: the rate with which the temperature is lowered

ω: the rate with which the penalty weight of the soft constraints is increased/decreased.

σ: the upper-bound for the allowed number of violations in global best infeasible solution

After extensive try&error experiments, we could determine reasonable values for most of the parameters, which are stable across all benchmark-instances, except for the T_{max}. T_{max} is surely one of the most influential parameters for $TTILS_{opt}$, because if it is too high or too low, the balance between diversification and intensification is easily distorted. Furthermore we believe that the best value for T_{max} actually depends on the individual instance-families and the instance-sizes.

As it has been already pointed out in [20], it is reasonable to take the *average distance* between two team sites into consideration when experimenting with initial temperature parameters for simulated-annealing type acceptance criteria. The average distances calculated for each of our benchmark-families are given in Table 5.1.

Family	$Inst_6$	$Inst_8$	$Inst_{10}$	$Inst_{12}$	$Inst_{14}$	$Inst_{16}$
NL	649	623	621	790	1094	1194
Galaxy	35	38	45	51	57	61
Super	4198	4167	4130	3910	3954	-

Table 5.1: Average distances between two team sites

Considering the average distances given above, we've experiment with the following temperature candidates for T_{max}:

{10, 20, 30, 50, 200, 400, 600, 800, 1000, 2000, 1000, 3000, 5000}

As expected, the best T_{max} values varied from instance to instance, probably depending on the individual instance-sizes and the average distance between two team sites. It is also noticeable that T_{max} doesn't have critical impacts on the solution quality for smaller instances, because they are already solved optimally by the embedded local search.

The final parameter settings, which we have determined based on the various experiments with T_{max} and extensive try&error experiments for other parameters, are given in Table 5.2. They are used for all of our computational benchmarks.

As you can see, all values for k_{max} are rather small values, so it seems to be best when the perturbation doesn't get too disruptive. As expected, the parameter ω should be set to a moderate value, so that the penalty weight doesn't fluctuate too much once it is settled. At the first glance, the cooling speed δ seems to be too low, but since we update the temperature at each iteration, this is actually a reasonable value. The parameter λ controls how often the acceptance-rate of the worse solutions is checked and it seems that the value 500 for λ is stable for most of the instances. The parameter α doesn't actually appear to have significant impacts as long as it is set to a reasonably small value.

As far as the soft-restarting mechanism is concerned, we just restart the algorithm, when during the last 15 temperature-resets no improvements could be achieved. In doing so, the initial solution for the next round is obtained by applying 5 random moves selected from the neighborhoods $[N_1, \ldots, N_5]$ to the current best solution.

Instance	k_{min}	k_{max}	ω	δ	λ	α	σ	T_{max}
NL_6	2	3	1.1	0.999	500	3	3	200
NL_8	2	4	1.1	0.999	500	3	3	300
NL_{10}	2	5	1.1	0.999	500	3	5	400
NL_{12}	2	6	1.1	0.999	500	3	6	500
NL_{14}	2	7	1.1	0.999	500	3	6	500
NL_{16}	2	8	1.1	0.9995	1000	3	7	550
GL_6	2	3	1.1	0.999	500	3	3	10
GL_8	2	4	1.1	0.999	500	3	3	20
GL_{10}	2	5	1.1	0.999	500	3	5	30
GL_{12}	2	6	1.1	0.999	500	3	6	30
GL_{14}	2	7	1.1	0.999	500	3	6	30
GL_{16}	2	8	1.1	0.9995	1000	3	7	30
Sup_6	2	3	1.1	0.999	500	3	3	400
Sup_8	2	4	1.1	0.999	500	3	3	400
Sup_{10}	2	5	1.1	0.999	500	3	5	500
Sup_{12}	2	6	1.1	0.999	500	3	6	500
Sup_{14}	2	7	1.1	0.999	500	3	6	1000

Table 5.2: Final parameter settings for each benchmark-instances (n denotes the team-size)

5.3 Computational Results

Using the final parameter settings presented in the previous section, we've conducted extensive computational experiments on aforementioned three benchmark-families. In the following, our evaluation results will be presented for each benchmark-family individually and compared with results of the different state-of-the-art approaches in the literature.

However it should be also noted that not all authors, who reported their best results on the Trick's TTP benchmark homepage [3], give detailed experimental results of their used methods. For instance, there are no publicly available data regarding detailed benchmark results and experimental settings from Langford, who obtained current best results for some instances of the *Super-x* and *Galaxy-x* families. To the best of our knowledge, his only publication for the TTP came recently in year 2010 [19]. There he presents his new improved neighborhood derived from *Partial-Swap-Teams* neighborhood of [4] and reports only his best results for *larger Galaxy-x* instances, which he has obtained using slightly modified version of the *Simulated Annealing* algorithm of [4]. No further details regarding experiments or benchmarks are given.

5.3.1 Results for the *NL-x* family

As already mentioned, the *NL-x* family is the most well-studied benchmark-set, for which numerous computational results from different methods are reported. Among different heuristics approaches in the literature, we've picked 5 *leading methods* (including the current state-of-the-art metaheuristics) to compare our results with:

- Simulated Annealing (TTSA) [4]
- Composite-neighborhood Tabu Search (CNTS) [17]
- Hybridization of Simulated Annealing and Hill-Climbing (SAHC) [20]
- Ant Colony Optimization (AFC-TTP) [34]
- Learning Hyper-Heuristic (LHH) [24]

For the *NL-x* family, we've conducted 2 separate evaluations with different timeout settings. For the first experiment, we tested our $TTILS_{opt}$ 10 times for each NL-instance and we adjusted the timeouts corresponding to the average time-values reported in [24]. In doing so, we've also tried to take the machine-difference into account, but, of course, the difference can be only roughly approximated.

For the second experiment, we again ran our $TTILS_{opt}$ 10 times for each NL-instance and this time we set the timeouts corresponding to the average time-values reported in [17] resulting

	Distance				Best Time				
Instance	Min	Avg	Max	Std.Dev	Min	Avg	Max	Std.Dev	Timeout
NL_6	23916	23916.0	23916	0	0	0.4	2	0.6663	10
NL_8	39721	39721.0	39721	0	4	60	198	56.896	300
NL_{10}	59583	59632.6	59727	60.592	489	2652	4305	1167.318	4700
NL_{12}	113360	114391.7	115289	708.349	410	3088	4688	1140.322	4700
NL_{14}	197230	199182.4	201638	1436.473	1873	3739.5	4688	874.921	4700
NL_{16}	281644	286178.0	289547	2199.686	2225	3686.8	4673	751.776	4700

Table 5.3: Experimental results of $TTILS_{opt}$ on the *NL-x* family with short timeout

	Distance				Best Time				
Instance	Min	Avg	Max	Std.Dev	Min	Avg	Max	Std.Dev	Timeout
NL_{10}	59583	59655.8	59910	119.33	70	1605.7	3816	1277.917	4600
NL_{12}	112960	113820.8	115586	798.296	3740	4979.3	6937	1940.038	7000
NL_{14}	194802	197185.1	201132	1688.060	5178	12577	18881	4721.297	19000
NL_{16}	277088	280868.3	283951	1943.884	10140	24433.2	283951	8216.410	32700

Table 5.4: Experimental results of $TTILS_{opt}$ on the *NL-x* family with long timeout

in much longer runs than in the first experiment. The results of both evaluations are given in Table 5.3 and 5.4, where all time values are given in seconds.

The results from the first evaluation (Table 5.3) are compared with the results of [24]. The results obtained from the second experiment with longer timeouts (Table 5.4) are compared with the results reported by [17], [34], [20] and [4]. We omit the results for NL_6 and NL_8 when comparing the results from the longer experiment, since we always get *optimal* solutions for these small instances.

It should be also noted that the comparisons with [24], [17] and [34] are made under the roughly *same* experimental conditions regarding the computation-time and the number of runs per instance. However the results of [20] and [4] are obtained from much longer test-runs and [4] also carried out 50 runs per instance. The results of the individual comparisons are given in Table 5.5, Table 5.6, Table 5.8, Table 5.7 and Table 5.9.

The evaluations and comparisons show very promising and favorable results for $TTILS_{opt}$. $TTILS_{opt}$ is always able to solve the small instances NL_6 and NL_8 to optimality in very short amount of time.

The comparison with LHH shows that $TTILS_{opt}$ is capable of producing very good solutions in short amount of time for all NL-instances and $TTILS_{opt}$ exhibits throughout better

	LHH				$TTILS_{opt}$				
Instance	Min	Avg	Std.Dev	Avg(Time)	Min	Avg	Std.Dev	Avg(Time)	Avg.Dif
NL_6	**23916**	**23916**	0	300	**23916**	**23916.0**	0	10	0
NL_8	**39721**	39801	172	1800	**39721**	**39721.0**	0	300	-0.2
NL_{10}	**59583**	60046	335	3600	**59583**	**59632.6**	60.6	4700	-0.7
NL_{12}	**112873**	115828	1313	3600	113360	**114391.7**	708.3	4700	-1.2
NL_{14}	**196058**	201256	2779	3600	197230	**199182.4**	1436.5	4700	-1.0
NL_{16}	**279330**	288113	4267	3600	281644	**286178.0**	2199.7	4700	-0.7

Table 5.5: Comparison of $TTILS_{opt}$ with LHH [24] on *NL-x* family

	AFC-TTP				$TTILS_{opt}$				
Instance	Min	Avg	Std.Dev	Avg(Time)	Min	Avg	Std.Dev	Avg(Time)	Avg.Dif
NL_{10}	59634	59928.3	155.47	4969.51	**59583**	**59655.8**	119.3	4600	-0.5
NL_{12}	**112521**	114437.4	895.7	7660.07	112960	**113820.8**	798.3	7000	-0.5
NL_{14}	196849	198950.5	1294.43	20870.07	**194802**	**197185.1**	1688.1	19000	-0.9
NL_{16}	278456	285529.6	3398.57	35931.27	**277088**	**280868.3**	1943.9	32700	-1.6

Table 5.6: Comparison of $TTILS_{opt}$ with AFC-TTP [34] on *NL-x* family

	SAHC				$TTILS_{opt}$				
Instance	Min	Avg	Std.Dev	Avg(Time)	Min	Avg	Std.Dev	Avg(Time)	Avg.Dif
NL_{10}	59821	60375.0	552.72	67619.6	**59583**	**59655.8**	119.3	4600	-1.2
NL_{12}	115089	116792.3	1069.59	82322.0	**112960**	**113820.8**	798.3	7000	-2.5
NL_{14}	196363	197769.9	731.52	96822.4	**194802**	**197185.1**	1688.1	19000	-0.3
NL_{16}	**274673**	**278477.9**	1885.53	111935.2	277088	280868.3	1943.9	32700	0.8

Table 5.7: Comparison of $TTILS_{opt}$ with SAHC [20] on *NL-x* family

	CNTS				$TTILS_{opt}$				
Instance	Min	Avg	Std.Dev	Avg(Time)	Min	Avg	Std.Dev	Avg(Time)	Avg.Dif
NL_{10}	59876	60424.2	823.9	7056.7	**59583**	**59655.8**	119.3	4600	-1.3
NL_{12}	113729	114880.6	948.2	10877.3	**112960**	**113820.8**	798.3	7000	-0.9
NL_{14}	194807	197284.2	2698.5	29635.5	**194802**	**197185.1**	1688.1	19000	-0.05
NL_{16}	**275296**	**279465.8**	3242.4	51022.4	277088	280868.3	1943.9	32700	0.5

Table 5.8: Comparison of $TTILS_{opt}$ with CNTS [17] on *NL-x* family

	TTSA				$TTILS_{opt}$				
Instance	Min	Avg	Std.Dev	Avg(Time)	Min	Avg	Std.Dev	Avg(Time)	Avg.Dif
NL_{10}	**59583**	**59605.96**	53.36	40268.62	**59583**	59655.8	119.3	4600	0.08
NL_{12}	**112800**	113853.00	467.91	68505.26	112960	**113820.8**	798.3	7000	-0.02
NL_{14}	**190368**	**192931.86**	1188.08	233578.35	194802	197185.1	1688.1	19000	2.2
NL_{16}	**267194**	**275015.88**	2488.02	192086.55	277088	280868.3	1943.9	32700	2.1

Table 5.9: Comparison of $TTILS_{opt}$ with TTSA [4] on *NL-x* family

average solution qualities and more stability than the LHH. However, LHH produced better minimum results than $TTILS_{opt}$ for instances NL_{12}, NL_{14} and NL_{16}.

Our results also compare favorably with other approaches with longer computation-times. In general, $TTILS_{opt}$ shows slightly better average performance than CNTS and SAHC for instances NL_{10}, NL_{12} and NL_{14}, even when SAHC has much longer computation-time for each instance. $TTILS_{opt}$ also finds better minimum solutions than CNTS and SAHC for instances NL_{10}, NL_{12} and NL_{14}. $TTILS_{opt}$ outperforms AFC-TTP both in average performance and in terms of minimum solution qualities for all instances. Only for NL_{12}, AFC-TTP finds slightly better minimum solution.

Compared with TTSA, which is the current best metaheuristics for the TTP, we can claim that our results are at least comparable, where the biggest gap in average solution quality is 2.2%. For NL_{12}, we even reach slightly better average performance.

We are somewhat disappointed from the results for the instance NL_{16}. Though comparable with other methods, the results for the NL_{16} were worse than originally anticipated. First we suspected that $TTILS_{opt}$ could have problems with the large size of 16 teams, but it seems to be rather an instance-specific problem, because $TTILS_{opt}$ produces very good results for the instance $Galaxy_{16}$ with the same team-size of 16. In the future, we will investigate more thoroughly if we can obtain better results for NL_{16} by investing even more effort in parameter-tuning.

5.3.2 Best results for the *NL-x* family

In this section, we present our global *best* results for *NL-x* family, which we have obtained during our research and compare them with the best results in the literature.

It should be noted that the best results for instances NL_{12} and NL_{14} don't come from the systematic evaluations presented before but from the various individual experiments we have conducted in the course of this thesis. They are obtained *without* the soft-restarting mechanism and the parameter settings, with which we have obtained them, are given in Table 5.10.

Instance	k_{min}	k_{max}	ω	δ	λ	α	σ	T_{max}	Time
NL_{12}	2	6	1.1	0.999	500	3	5	600	2700
NL_{14}	2	7	1.1	0.999	500	5	3	800	32204

Table 5.10: Parameter settings for best NL_{12}-NL_{14}-results

Our best results for *NL-x* family are given in Table 5.11, which also lists all the best results reported in the past by different authors. The comparison table for *NL-x* family is adapted from [9].

As apparent from the comparison tables, our $TTILS_{opt}$ produces results comparable to the current upper-bounds, despite we didn't conduct many experiments with very long computation-time. Our best results are slightly better than those obtained by Lim et al. [20] (except for the largest NL_{16} instance) and are slightly worse than the best results of [17] for $NL_{12}, NL_{14}, NL_{16}$ instances. Clearly, the current best performing metaheuristic for the TTP is the *Simulated Annealing* approach TTSA and its extensions [4],[35]. But as you can see, our best results are not far away from their best results either. All in all we can claim that $TTILS_{opt}$ produces very competitive results and shows great potential for future researches.

Authors	Method	NL_4	NL_8	NL_6	NL_{10}	NL_{12}	NL_{14}	NL_{16}
Easton et al.	Linear Programming (LP)	**8276**	**23916**	44113				312623
Benoist et al.	constraint programming and lagrange relaxation	**8276**	**23916**	42517	68691	143655	301113	437273
Cardemil	Tabu Search	**8276**	**23916**	40416	66037	125803	205894	308413
Zhang	Unknown (data from TTP website)	**8276**	24073	39947	61608	119012	207075	293175
Shen and Zhang	Greedy big step meta-heuristics			39776	61679	117888	206274	281660
Lim et al.	Simulated Annealing and Hill Climbing	**8276**	**23916**	**39721**	59821	115089	196363	274673
Langford	Unknown (data from TTP website)				**59436**	112298	190056	272902
Crauwels and Oudheusden	Ant Colony Optimization with local improvement	**8276**	**23916**	40797	67640	128909	238507	346530
Anagnostopoulos et al.	Simulated Annealing	**8276**	**23916**	**39721**	59583	111248	188728	263772
Gaspero and Schaerf	Composite Neighborhood Tabu Search Approach				59583	111483	190174	270063
Chen et al.	Ant-Based Hyperheuristic	**8276**	**23916**	40361	65168	123752	225169	321037
Van Hentenryck and Vergados	Population-based Simulated Annealing	**8276**	**23916**	**39721**	**59436**	**110729**	**188728**	**261687**
Mustafa Mısır et al.	Learning Automata Hyperheuristics	**8276**	**23916**	**39721**	59583	112873	196058	279330
Uthus et al.	Ant Colony Optimization	**8276**	**23916**	**39721**	59583	112521	195627	278456
This work	Iterated Local Search	**8276**	**23916**	**39721**	59583	112478	194384	277088

Table 5.11: Best results for NL instances (adapted from [9])

5.3.3 Results for the *Super-x* family

To the best of our knowledge, the only publicly available experimental results on the *Super-x* benchmark-set are reported in [24]. As we've done it for the *NL-x* family, we adjust the computation-time of our experiment to be comparable with those reported in [24] and test our algorithm 10 times for each instance. Our results are reported in Table 5.12 and compared with the results of LHH in Table 5.13.

As you can see from the results, $TTILS_{opt}$ is able to solve the small instances Sup_6 and Sup_8 always to the optimality and LHH [24] is again outperformed by $TTILS_{opt}$. Additionally, we give the gaps between our experiment's best values and the current best results for the *Super-x* family reported on the TTP-benchmark-website [3] in Table 5.14.

As mentioned already, the current best results for the *Super-x* family are reported by Langford and Uthus, but, to the best of our knowledge, there are no further information available what concrete experiments they have conducted and what the average solution-quality and running-time of their approaches are.

	Distance				Best Time				
Instance	Min	Avg	Max	Std.Dev	Min	Avg	Max	Std.Dev	Timeout
Sup_6	130365	130365.0	130365	0	0	0.1	1	0.3	10
Sup_8	182409	182409.0	182409	0	3	77.3	286	80.96	300
Sup_{10}	318007	318225.5	318691	262.973	186	2313.6	4104	1147.795	4700
Sup_{12}	469290	472002.1	115289	1228.140	150	2514.4	4646	1464.588	4700
Sup_{14}	594388	600533.6	610824	5187.210	538	1911.6	3005	939.88	4700

Table 5.12: Experimental results of $TTILS_{opt}$ on *Super-x* family

	LHH				$TTILS_{opt}$				
Instance	Min	Avg	Std.Dev	Avg(Time)	Min	Avg	Std.Dev	Avg(Time)	Avg.Dif
Sup_6	**130365**	**130365**	0	300	**130365**	**130365.0**	0	10	0
Sup_8	**182409**	182975	558	1800	**182409**	**182409.0**	0	300	-0.3
Sup_{10}	318421	327152	6295	3600	**318007**	**318225.5**	262.973	4700	-2.7
Sup_{12}	**467267**	475899	5626	3600	469290	**472002.1**	1228.140	4700	-0.8
Sup_{14}	599296	634535	13963	3600	**594388**	**600533.6**	5187.210	4700	-5.3

Table 5.13: Comparison of $TTILS_{opt}$ with LHH [24] on *Super-x* family

Instance	$TTILS_{opt}$	Best	Dif(%)
Sup_6	**130365**	**130365**	0
Sup_8	**182409**	**182409**	0
Sup_{10}	318007	**316329**	0.5
Sup_{12}	469290	**463876**	1.2
Sup_{14}	594388	**571632**	3.98

Table 5.14: Comparison with best results of the *Super-x* family

5.3.4 Results for the *Galaxy-x* family

In this section, we report our experimental results for the *Galaxy-x* benchmark-set. Unfortunately due to novelty of the *Galaxy-x* family, there are no extensive computational results available for this benchmark-set yet. As mentioned previously, author Langford [19] reported most of the current best results for *Galaxy-x* family using modified version of TTSA from [4], but didn't give further details about how he obtained them and what experiments he has conducted to benchmark his approach.

In the following, we present our results for the *Galaxy-x* family in Table 5.15, which are obtained under the same experimental settings as for the *Super-x* family. Table 5.16 lists the gaps between our minimum values and the current best values of Langford showing that $TTILS_{opt}$ is also able to produce very good solutions for the *Galaxy-x* family in short amount of time.

	Distance				Best Time				
Instance	Min	Avg	Max	Std.Dev	Min	Avg	Max	Std.Dev	Timeout
GL_6	1365	1365.0	1365	0	0	0.1	1	0.3	10
GL_8	2373	2373.0	2373	5	3	31.1	73	20.554	300
GL_{10}	4535	4547.4	4585	16.704	562	2351.1	4374	1253.793	4700
GL_{12}	7318	7395.6	7502	56.075	1935	3591.3	4618	3591.3	4700
GL_{14}	11324	11434.5	11545	69.717	1439	3089.6	4665	1198.567	4700
GL_{16}	15172	15733.0	16173	258.457	1939	4119.7	4677	772.589	4700

Table 5.15: Experimental results of $TTILS_{opt}$ on the *Galaxy-x* family

Instance	$TTILS_{opt}$	Best	Dif(%)
GL_6	**1365**	**1365**	0
GL_8	**2373**	**2373**	0
GL_{10}	**4535**	**4535**	0
GL_{12}	7318	**7197**	1.7
GL_{14}	11324	**10918**	3.7
GL_{16}	15172	**14900**	1.8

Table 5.16: Comparison with best results of the *Galaxy-x* family

CHAPTER 6

Conclusion and future work

In this thesis, we have proposed a novel metaheuristics approach based on the *Iterated Local Search* framework for solving the very challenging *Traveling Tournament Problem*.

First, we developed a basic ILS approach $TTILS_{basic}$ to assess the applicability of the ILS-principle to the TTP. The initial results of $TTILS_{basic}$ were promising and based on the insights gained by analyzing $TTILS_{basic}$, we further optimized and extended $TTILS_{basic}$, which led to our final version $TTILS_{opt}$.

The proposed $TTILS_{opt}$ incorporates perturbation mechanism, which uses random moves from higher-order neighborhoods to perturbate solutions and varies its strength dynamically. $TTILS_{opt}$ embeds simple *Hill-Climbing* heuristic as the local-search component, where we have experimented with different compositions of neighborhoods. In order to escape difficult local optima, $TTILS_{opt}$ uses an advanced simulated annealing type acceptance criterion with *non-monotonic* cooling schedule. $TTILS_{opt}$ incorporates also other additional extensions like *strategic-oscillation* and *soft-restarting mechanism*.

We have implemented the proposed algorithm $TTILS_{opt}$ and conducted extensive computational experiments on selected benchmark-families publicly available from Trick's TTP-benchmark-website. The results of our experiments are compared with other state-of-the-art metaheuristics proposed for the TTP in the literature.

The comparison on the *NL-x* benchmark-set with other approaches showed particularly favorable results for $TTILS_{opt}$. $TTILS_{opt}$ is able to solve the smaller instances $[NL_4, NL_6, NL_8]$ to optimality in only few seconds and for larger instances $[NL_{10}, NL_{12}, NL_{14}]$, $TTILS_{opt}$ exhibits better average solution qualities and robustness than most of other compared approaches, being only second to the current best-performing *Simulated Annealing* approach TTSA [4].

These results confirm the potential and effectiveness of the ILS-based metaheuristics approach for solving the TTP.

Furthermore, we investigated the possibility of defining an efficient *incremental* evaluation function when using the five neighborhoods $[N_1, \ldots, N_5]$ introduced in [4], which are also the underlying basic neighborhoods for $TTILS_{opt}$. We were able to propose incremental algorithms with better complexity than the naive approach to calculate delta-values for all of the three cost-components efficiently.

The connectivity of the neighborhoods $[N_1, \ldots, N_5]$ is also an interesting issue, since, to the best of our knowledge, it is still unknown whether the search-space is connected or not under the aforementioned neighborhoods. We propose an *experimental* approach to investigate this issue, where the reachability between two *random* solutions is tested by means of heuristic optimization. The preliminary results of our experiment on *NL-x* benchmark-set further support the *hypothesis* that the search-space is connected under the considered neighborhoods.

For the future, several issues could be further investigated to improve the performance of $TTILS_{opt}$:

- Investigate if a more powerful embedded local-search component like Tabu-Search or Simulated Annealing can improve the solution quality.
- Search for some useful problem-specific properties, which can be exploited in a more advanced perturbation scheme.
- Explore novel ideas for designing new neighborhoods. For instance, it would be interesting to see if it is feasible to search in a much bigger search-space, where the *double-round-robin* constraint is also a *soft-constraint*.
- Reevaluate $TTILS_{opt}$ with much longer computation-time and investigate how much can longer computation-time improve the solution quality of $TTILS_{opt}$.

Bibliography

[1] Non-round robin traveling tournament problem. http://mat.gsia.cmu.edu/TOURN/nonrr/, 2012.

[2] Relaxed traveling tournament problem. http://mat.gsia.cmu.edu/TOURN/relaxed/, 2012.

[3] TTP benchmark. http://mat.gsia.cmu.edu/TOURN/, 2012.

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

[5] D. Applegate, W. Cook, and A. Rohe. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, 2003.

[6] E.B. Baum. Iterated descent: A better algorithm for local search in combinatorial optimization problems. Technical report, Caltech, Pasadena, CA, 1986.

[7] T. Benoist, F. Laburthe, and B. Rottembourg. Lagrange relaxation and constraint programming collaborative schemes for travelling tournament problems. In *In CP-AI-OR'2001, Wye College*, pages 15–26, 2001.

[8] R. Bhattacharyya. A note on complexity of traveling tournament problem. *Optimization Online*, 2009.

[9] P.C. Chen, G. Kendall, and G.V. Berghe. An ant based hyper-heuristic for the travelling tournament problem. In *Computational Intelligence in Scheduling, 2007. SCIS'07. IEEE Symposium on*, pages 19–26, 2007.

[10] M. Chiarandini, T. Stützle, et al. An application of iterated local search to graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, 2002.

[11] R.K. Congram, C.N. Potts, and S.L. Van De Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1):52–67, 2002.

[12] H. Crauwels and D. Van Oudheusden. A generate-and-test heuristic inspired by ant colony optimization for the traveling tournament problem. In *Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling*, pages 314–315, 2002.

[13] D. de Werra. Scheduling in sports. *North-Holland Mathematics Studies*, 59:381–395, 1981.

[14] J.A. Dıaz and E. Fernández. A tabu search heuristic for the generalized assignment problem. *European Journal of Operational Research*, 132(1):22–38, 2001.

[15] K. Easton, G. Nemhauser, and M. Trick. The traveling tournament problem description and benchmarks. In *Principles and Practice of Constraint Programming—CP 2001*, pages 580–584, 2001.

[16] K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: a combined integer programming and constraint programming approach. *Practice and Theory of Automated Timetabling IV*, pages 100–109, 2003.

[17] L.D. Gaspero and A. Schaerf. A composite-neighborhood tabu search approach to the traveling tournament problem. *Journal of Heuristics*, 13(2):189–207, 2007.

[18] D.S. Johnson and L.A. McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, pages 215–310, 1997.

[19] G. Langford. An improved neighbourhood for the traveling tournament problem. *Arxiv preprint arXiv:1007.0501*, 2010.

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

[21] H. Lourenco, O. Martin, and T. Stützle. Iterated local search. *Handbook of metaheuristics*, pages 320–353, 2003.

[22] O. Martin, S.W. Otto, and E.W. Felten. Large-step markov chains for the TSP incorporating local search heuristics* 1. *Operations Research Letters*, 11(4):219–224, 1992.

[23] E. Mendelsohn and A. Rosa. One-factorizations of the complete graph—a survey. *Journal of Graph Theory*, 9(1):43–65, 1985.

[24] M. Misir, T. Wauters, K. Verbeeck, and G. Vanden Berghe. A new learning hyper-heuristic for the traveling tournament problem. In *Proceedings of the 8th Metaheuristics International Conference (MIC 2009), Hamburgh Germany.*, 2009.

[25] N. Musliu. An iterative heuristic algorithm for tree decomposition. *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 133–150, 2008.

[26] C.C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 179(3):775–787, 2007.

[27] F. Schoen. Stochastic techniques for global optimization: A survey of recent advances. *Journal of Global Optimization*, 1(3):207–228, 1991.

[28] T. Stützle. Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539, 2006.

[29] T. Stützle and H.H. Hoos. Analyzing the run-time behaviour of iterated local search for the tsp. In *III Metaheuristics International Conference*, pages 1–6, 1999.

[30] Thomas Stützle. Applying iterated local search to the permutation flow shop problem. Technical Report AIDA–98–04, FG Intellektik, FB Informatik, Technische Universität Darmstadt, August 1998.

[31] C. Thielen and S. Westphal. Complexity of the traveling tournament problem. *Theoretical Computer Science*, 412(4):345–351, 2011.

[32] M. Trick. Formulations and reformulations in integer programming. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 833–836, 2005.

[33] D. Uthus, P. Riddle, and H. Guesgen. Dfs* and the traveling tournament problem. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 279–293, 2009.

[34] D.C. Uthus, P.J. Riddle, and H.W. Guesgen. An ant colony optimization approach to the traveling tournament problem. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 81–88, 2009.

[35] P. Van Hentenryck and Y. Vergados. Population-based simulated annealing for traveling tournaments. In *Proceedings of the 22nd national conference on Artificial intelligence-Volume 1*, pages 267–272, 2007.

[36] W.D. Wallis, A.P. Street, and J. Seberry. *Combinatorics: Room squares, sum-free sets, Hadamard matrices*, volume 292. Springer-Verlag, 1972.