Informatics

Adaptive Large Neighbourhood Search for the Double-Round-Robin Sports Tournament Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Andreas Krystallidis, BSc
Matrikelnummer 11808586

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.techn. Nysret Musliu

Wien, 29. September 2023

[redacted]

Andreas Krystallidis

[redacted]

Nysret Musliu

Informatics

Adaptive Large Neighbourhood Search for the Double-Round-Robin Sports Tournament Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Andreas Krystallidis, BSc
Registration Number 11808586

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.techn. Nysret Musliu

Vienna, 29th September, 2023

[redacted]
Andreas Krystallidis

[redacted]
Nysret Musliu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Andreas Krystallidis, BSc

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.

Wien, 29. September 2023

Andreas Krystallidis

Acknowledgements

I would like to thank my advisor Associate Prof. Dr. Nysret Musliu for always taking the time to meet with me and discuss my various ideas surrounding this thesis sending me in the right direction at every step of the way. I would also like to thank my girlfriend Vanessa for her emotional support throughout this thesis as well as for proofreading it. Finally, I also want to thank all the organizers of the ITC2021 competition. Without their efforts in organizing an amazing event in which the problem discussed in this thesis was researched, I would have never gotten into this domain.

Kurzfassung

Sportturniere ziehen Millionen von Fans und Sportler auf der ganzen Welt in ihren Bann. Die erfolgreiche Organisation und Planung dieser Turniere spielt eine wichtige Rolle bei der Gewährleistung eines fairen Wettbewerbs, der Maximierung der Einnahmen und der Verbesserung des Gesamterlebnisses für die Zuschauer. Heutzutage bringen die Größe und Bedeutung solcher Veranstaltungen jedoch so viele verschiedene Faktoren mit sich, dass es für die Organisatoren fast unmöglich ist, bei der Erstellung von Zeitplänen für solche Turniere jedes Detail zu berücksichtigen. Aus diesem Grund wurden in den letzten Jahren viele verschiedene Ansätze entwickelt, um solche Spielpläne automatisch mit Hilfe von Computern zu erstellen.

Viele solcher Tools sind im Rahmen des International Timetabling Competition 2021 (ITC2021) entstanden, die sich speziell auf die Suche nach guten Heuristiken für schwierige Instanzen des zeitbeschränkten Double-Round-Robin-sports tournament (DRRST) Problem konzentrierte, welches ein sehr häufiges Format bei Sportveranstaltungen ist. Während viele Algorithmen bereits sehr gute Ergebnisse zeigen, hat der Wettbewerb auch deutlich gemacht, wie schwierig es ist, zufriedenstellende Zeitpläne für solche Turniere zu entwickeln. Viele der im Rahmen des Wettbewerbs entwickelten Ansätze mussten eine übermäßige Menge an Ressourcen einsetzen, um qualitativ hochwertige Lösungen zu erzeugen. Darüber hinaus haben bis heute nur 3 der 45 Instanzen, die während des Wettbewerbs vorgestellt wurden, Lösungen, die bewiesenermaßen optimal sind. In dieser Arbeit schlagen wir eine Adaptive Large Neighborhood Search vor, die weniger Rechenressourcen verbraucht als bisherige LNS-Ansätze und dennoch Ergebnisse erzielt, die dem Stand der Technik nahe kommen. Diese Effizienzsteigerung resultiert aus einer Kombination von multi-armed bandit-Methoden aus dem Reinforcement Learning, sechs neu entwickelten Nachbarschaftstypen sowie der Einführung neuer Heuristiken, die von der Tabu-Suche und Methoden der manuellen Optimierung inspiriert sind. Mit Hilfe dieser Techniken sind wir in der Lage, trotz unseres geringeren Ressourcenverbrauchs für 3 der 45 Instanzen des Wettbewerbes neue beste bekannte Lösungen zu finden. Unsere Forschung zeigt, wie wichtig der Einsatz adaptiver Methoden ist, wenn die Ressourcen nicht im Überfluss vorhanden sind. Schließlich zeigen wir auch, dass selbst bei ausschließlicher Betrachtung von LNS-basierten Ansätzen verschiedene Instanzen unterschiedliche Nachbarschaftstypen und Algorithmuskonfigurationen bevorzugen.

Abstract

Sports tournaments captivate millions of fans and athletes all around the world. The successful organization and scheduling of these tournaments play an important role in ensuring fair competition, maximizing revenue, and enhancing the overall spectator experience. However, nowadays the size and importance of such events introduce so many different factors that it becomes almost impossible for organizers to factor in every detail when creating schedules for such tournaments. For this reason in recent years, many different approaches have been developed to create such schedules computationally.

Many such tools have emerged in the International Timetabling Competition in 2021 (ITC2021) that focused specifically on finding good heuristics for difficult instances of the time-constraint double-round-robin sports tournament (DRRST) problem which is a very common format in sporting events. While many algorithms already show very good results, the competition has also highlighted just how hard it is to come up with satisfactory schedules for such tournaments. Many of the approaches developed during the competition had to use an excessive amount of resources to find high-quality solutions. Furthermore, to this date, only 3 out of the 45 instances featured during the competition have solutions that were proven to be optimal. In this thesis, we propose an Adaptive Large Neighborhood Search, that uses fewer computational resources than previous LNS approaches while still achieving results close to the state of the art. This increase in efficiency stems from a combination of multi-armed bandit methods from reinforcement learning, six newly developed neighborhood types as well as the introduction of new heuristics that are inspired by tabu search and methods of manual optimizations. With the help of those techniques, we are able to find new best-known solutions to 3 out of the 45 competition instances despite our lower resource usage. Our research highlights the importance of using adaptive methods when resources are not abundant. Finally, we also show that even when only considering LNS-based approaches different instances favor different neighborhood types and algorithm configurations.

Contents

CHAPTER 1

Introduction

Competition is part of human nature and has existed as long as mankind. Of course, the ways in which we compete have developed over time from hunting and fighting towards more civilized sports and games. Nowadays, competitions usually follow strict rules and formats to ensure fairness for all involved parties while also providing a good experience to fans who follow such events. Creating a schedule for a sports competition is a hard computational task because of the many factors that come into play which range from reserving venues to balancing out home advantages.

In this thesis, we focus on a specific tournament format called the time-constraint double-round-robin sports tournament (DRRST) which has received recent attention through the ITC2021 [VG23b]. The DRRST format is amongst the most popular formats for sports tournaments and is probably best known through various football leagues. Because of the ITC2021 competition many different approaches [LFMSP21, RPGS22, POW21, FT22] for finding close-to-optimal schedules have been developed and it has become apparent that ILP solvers are a very good tool for this problem. However, only one team [POW21] has tried to combine the promising aspects of a large neighborhood search (LNS) with elements from reinforcement learning to create a more adaptive form of the LNS commonly called adaptive large neighborhood search (ALNS) [RP06]. While this approach by Phillips et al. [POW21] only achieved mediocre results in most instances from the competition, the general idea of using an ALNS seems very promising considering that the first place in the competition, which was achieved by Lamas-Fernandez et al. [LFMSP21], used a variant of LNS that essentially exhaustively searched for possible improvements without using any forms of learning except for changing the sizes of neighborhoods used.

The competition also highlighted the general difficulty of the DRRST. Only 3 out of 45 instances were solved to optimality even though the instances only used between 16 and 20 teams which is a realistic amount for a real sports tournament. This difficulty combined with the fact that the competition imposed no limit on time and computational resources used (except a final deadline to send in solutions) results in most teams using very long runtimes, in some cases combined with a lot of computing power to achieve competitive results.

Finding ways to acquire good solutions to the DRRST problem automatically will make it much easier for sports tournaments in the future to create more optimal schedules. But considering that not only big tournaments that have a lot of money to spend on lots of computational power are interested in holding well-organized events, looking into ways of making the search for optimal schedules more adaptive and therefore more efficient in regards to time and resources is essential.

1.1 Aims of This Thesis

Our main objective for this thesis is to develop and analyze a new ALNS approach for solving the time-constraint DRRST that does not rely on excessive computational power or time.

This goal entails researching the following topics:

- Finding and comparing new methods for adaptive selection of neighborhoods.
- Improving upon previous methods to more efficiently generate feasible solutions for the problem instances.
- Developing new neighborhoods that can be used to improve LNS based algorithms.
- Statistically evaluating our results with the help of tools for automated parameter tuning and comparing them to the current state-of-the-art.

1.2 Contributions

The main contributions of this thesis are:

- A new multi-stage algorithm for generating feasible solutions to the DRRST problem.
- Six new neighborhood types for ILP-based methods to the DRRST problem.
- An examination of pre-existing and new neighborhood types to find out which are most effective.
- An ALNS algorithm that finds good solutions to DRRST faster than other state-of-the-art ILP-based methods.
- A new heuristic that helps to escape local minima in LNS-based approaches for the DRRST problem and potentially other scheduling problems.
- New best-known solutions for three of the instances of the ITC2021 [VG23b].

1.3 Structure of the Thesis

The following thesis is split into four chapters. In Chapter 2 we present an overview of the DRRST including a problem definition and a summary of the state-of-the-art. In Chapter 3 we describe our novel ALNS approach in detail including six new neighborhood types, a new approach to generating feasible solutions as well as new strategies that increase adaptivity. In Chapter 4 we evaluate our ALNS and compare both the end results and intermediate stages with other approaches and variants of our ALNS. Finally, in Chapter 5 we will sum up our findings and give an outlook on possible future research directions.

CHAPTER 2

The Double Round Robin Sports Tournament Problem

2.1 Problem Definition

Sports Tournaments are celebrated all around the world attracting large viewerships. Scheduling the individual games of such events can be very challenging [Bri08], especially since there is no one-size-fits-all solution that is ideal for every tournament. Each event comes with different constraints that can range from very general constraints, like the number of consecutive games that can be played in the home stadiums of the respective teams, to very specific constraints like team A not being allowed to play against team B in the first three days of the tournament. To manage the different formats and constraints of sports tournaments Van Bulck et al. [VBGSG20] have created a framework called RobinX that makes it possible to encode the various tournaments in a unified format. Because of the large interest in optimized solutions to the sports tournament problem, the International Timetabling Competition in 2021 (ITC2021) [VG23b] has hosted a competition for the time-constrained double-round-robin sports tournament (DRRST). The time-constrained (also called compact) DRRST is a subset of the general sports tournament problem that only considers tournaments with a schedule where each team plays all other teams exactly twice and every team plays exactly one match per day (this implies an even number of teams). Furthermore, they specify the constraints that are relevant to the competition.

The following categories and corresponding constraints are used in the ITC2021 [VG23b]:

- Capacity Constraints: Consists of four individual constraints that limit the number of home/away games during certain time slots and the amount of games teams (or sets of teams) can play against each other during certain time slots (or sets of time slots). The four individual constraints are referred to as CA1, CA2, CA3, and CA4.

- CA1: Limits the maximum amount of home or away games for a single team during a set of slots. An example of this could be that team 1 is allowed to play at most 2 home games in the first 5 timeslots of the tournament.
- CA2: Limits the maximum amount of home games, away games, or both a single team is allowed to play against a set of other teams during a set of time slots. An example of this could be that team 1 is allowed to play at most 1 away game against teams 3 and 4 in the last 7 timeslots of the tournament. However, in this example, team 1 can play away games in all 7 timeslots as long as not more than 1 is against teams 3 or 4.
- CA3: This constraint is similar to CA2 except that we don't specify any timeslots but instead use a parameter called "intp". This parameter is an integer that specifies that in any interval of intp consecutive timeslots, the team can only have a limited amount of home games, away games, or both against a set of other teams.
- CA4: With this constraint, we use two sets of teams (teams1 and teams2) and limit the total amount of home games, away games, or both between those sets. We can either specify certain slots like in CA1 and CA2 or this constraint is applied globally so that the constraint has to hold on each individual slot.

• Game Constraints: Contains a single constraint (GA1) that imposes restrictions on specific matches (pairs of two teams + location).

  - GA1: Given a set of time slots and matches only a given amount of those matches can be played during those time slots. One example of this could be the match between team 1 and team 2 with team 1 being the home team (written as (1, 2)) and the match (3, 4) can not both happen on the third timeslot of the tournament.

• Break Constraints: Is made up of two different constraints that limit the number of breaks. A break is defined as playing either at home or away multiple days in a row. We differentiate between home breaks and away breaks depending on if the break occurs because of consecutive home or away games. The two constraints of this category are called BR1 and BR2.

  - BR1: Limits the total number of home breaks, away breaks, or both for a single team during certain time slots. E.g. team 1 can not have consecutive games at home on the first 3 days of the tournament.
  - BR2: This constraint is very similar to BR1 except that it will not look at an individual team but a set of teams and it does not differentiate between home and away breaks but always considers both. E.g. teams 1, 2, and 3 cant have more than 5 breaks (either at home or away) in the first half of

the tournament. In practice, this constraint is often used to limit the total amount of breaks in the whole tournament.

- Fairness Constraints: Only consists of a single constraint (called FA2) that enforces balance on the number of home breaks each individual team has

  - FA2: Limits the difference of home breaks between multiple teams at certain time slots. E.g. at time slots 2 and 3 the difference in the number of home breaks between Teams A and B can not be higher than 2. In practice, this is used to ensure fairness at various points in the competition because it is considered to be an advantage if a team plays at home more often than another team.

- Separation Constraints: Limits how far apart the home and away games between pairs of teams can be. The only constraint in this category is called SE1.

  - SE1: Given a pair of teams the constraint limits the amount of timeslots between the first match and the second match of those teams. E.g. if the pair consisting of teams 1 and 2 have their first match in the third timeslot the return game can not be any later than the eighth timeslot.

Fairness and Separation Constraints only appear as soft constraints, meaning that they don't have to be fulfilled for a solution to be feasible. All other constraints can appear both as soft and hard constraints. If a soft constraint is violated it adds a penalty to our objective function. The exact penalty depends on the kind of constraint and how close we are to fulfilling it. For example, if we look at a violated CA1 constraint the penalty is equal to the amount of home or away games more than the given maximum (so if the maximum is two home games but there are four home games the penalty would be two). The exact penalty terms for each constraint can be found in [VG23b].

The instances used in the ITC2021 all have between 16 and 20 teams. This closely resembles real-world tournaments in, for example, soccer [GS12] while also being very challenging to solve using state-of-the-art techniques. The concrete instances were generated by Van Bulck and Goossens [VG23b] using instance space analysis [SB15a] to generate a variety of difficult problems with diverse features. The results of the competition [VG23b] also suggest that the problems are indeed non-trivial as only 2 out of 45 instances were solved to proven optimality. Furthermore, the competition has shown that various different state-of-the-art techniques, some of which we will discuss in Section 2.2, have varying degrees of success depending on the exact instances which indicates that the instances are indeed diverse.

2.2 State of the Art

Creating optimized timetables for sporting events is a long-standing problem with early research coming from the 1970s [BW77]. The current state-of-the-art was shown in the very recent ITC2021 competition [VG23b] where an ILP model from Lamas-Fernandez et al. [LFMSP21] had the best performance. They used a so-called fix-and-relax approach where they fixed a large portion of the variables (with different strategies for deciding which variables to fix for adaptability). While this does not follow the exact steps of an ALNS, since nothing ever gets truly destroyed or repaired, the similarities are quite obvious. The fixated variables can be interpreted as variables that are not destroyed (and therefore remain unchanged) while the free variables are put into an ILP solver that finds an optimal assignment for the sub-problem (which is almost like destroying the old assignments for those variables and finding new ones). So in a way, Lamas-Fernandez et al. have implemented something that resembles an ALNS. However, they have not really considered many ways to be adaptive. They simply iterated over all five of their neighborhoods and all currently unfulfilled constraints to exhaustively search for possible improvements. Every time all neighborhoods failed to make any further improvement they increased the size of their neighborhoods by one until the ILP solver did not terminate in a reasonable amount of time anymore. While this outperformed all other teams it also had one of the longest runtimes in the competition and used very high computational resources. Concretely a single run on a single instance took up to 6 days of runtime while also using 60 multi-start runs using 4 CPUs (2.6GHz Intel Sandy Bridge) with 16GB of memory.

Another state-of-the-art technique from the ITC2021 competition uses Multi-neighborhood Simulated Annealing [RPGS22]. Rosati et al. achieved second place in the competition by using six different neighborhoods in three stages of Simulated Annealing. Each stage on its own represents a full run of the classic Simulated Annealing Metaheuristic. In the first stage, they focus purely on hard constraints. In the second phase, all constraints are considered but moves that violate hard constraints are penalized. Finally, in the third phase, moves that violate hard constraints are completely forbidden. Using this heuristic each run takes roughly between 1,5 and 13 hours on a single virtual core (of an AMD Ryzen Threadripper PRO 3975WX processor with 64 virtual cores) which is much faster than the previously discussed method by Lamas-Fernandez et al. [LFMSP21]. However, it is noteworthy that in order to achieve such a high placement in the competition each instance was run a minimum of 48 times. Nevertheless, even the average reported results of each instance are quite competitive, especially considering the comparably low runtime.

As previously mentioned there also exists an approach to optimize the timetables using an ALNS. Phillips et al. [POW21] have also taken part in the ITC2021 and have to the best of our knowledge implemented the most novel ALNS technique for the time-constraint DRRST. They used four different neighborhoods and treated the selection of neighborhoods as a multi-armed bandit problem. The size of the neighborhood they destroyed is determined based on the runtime of the previous iteration. Concretely if the ILP manages to repair the schedule in less than 5 minutes the size of the neighborhood is increased by one and if it takes more than 30 minutes it is decreased by one. They also used a lot of computing resources for their approach using almost ten days of computing time on a c2-standard-30 computing instance from Google Cloud. Their results were not quite as good as other strategies when they were starting from scratch (achieving 7^{th} place in the competition), but they manged to find several best-known solutions by using the previously best-known solutions as their starting schedule.

In a very recent paper Van Bulck and Goossens have developed a first-break-then-schedule (FBST) approach [VG23a] where the problem is split into two different problems that are solved one after the other. In the first stage, for each time slot they fixed which teams play at home and which play away, therefore solving the break constraints without fixing the exact matches and creating a so-called home-away pattern (HAP). In the second stage, an opponent schedule was generated on top of the HAP, meaning they fixed the exact matchups according to the previously created assignments for match locations. Both stages are solved using different ILP formulations. Because it is very likely that the generation of the HAP can lead to infeasibility, they used benders decomposition [Ben05, BG23] which essentially means they forbade certain infeasible variable assignments for the HAP set using Bender's infeasibility cuts or in case the second stage found a feasible solution strengthened it using Benders' optimality cuts. Using this technique they were able to find 10 new best-known solutions for several instances of the ITC2021 instances [VG23b]. However, similar to many approaches from the ITC2021 competition the computational resources needed for this approach are comparably high. They allowed up to 24 hours of runtime per instance on 10 cores for generating the HAP set and then used 50 different random seeds each running an average of 1 hour and 45 minutes on a single core.

It is noteworthy that almost all approaches that we looked at in this section (excluding the one by Rosati et al. [RPGS22]) used strategies to look at small sub-problems instead of the whole schedule at once. Therefore, we are convinced that using the ALNS approach, which focuses on solving many small problems instead of one big problem will produce very good results in a shorter time than most other approaches. Additionally, we use new techniques to select the neighborhood type and size based on learned qualities, making a metaheuristic that is more adaptable than existing solutions. Furthermore, we came up with a novel approach to generating feasible solutions that found feasible solutions faster than the current approaches from the literature. We also introduce six new neighborhoods and analyze their effectiveness and the effectiveness of the four neighborhoods that were used in previous methods.

CHAPTER 3

An ALNS Approach for Generating DRRST Schedules

In this chapter, we present a new ALNS approach that builds upon approaches from the ITC2021 [VG23b] using both established and innovative new neighborhoods and techniques for adaptivity.

On a high level, we build upon the ALNS approach that was originally designed for vehicle routing problems [RP06] which in turn extends the Large Neighborhood Search (LNS) developed by Shaw [Sha98]. This means we start off by creating some likely infeasible schedule iteratively destroy parts of it and optimally reconstruct the destroyed parts using ILP. To determine how much of the schedule and which exact parts we should destroy we treat neighborhood type selection as a multi-armed-bandit problem [Rob52] while learning over time the size we need to use for each neighborhood to achieve similar runtimes in the reconstruction step of each neighborhood type. The time target for the reconstruction time then changes over time depending on how frequent improvements are found.

Similar to other approaches [LFMSP21, RPGS22] tackling this problem, we also split the problem into first creating a feasible schedule (ignoring all soft constraints) and afterward we start to improve the schedule without allowing any infeasible solutions as intermediate results.

In Section 3.1 we will describe the neighborhoods we analyzed and used as part of our ALNS. We will then discuss a new addition to a common approach to creating a good initial solution that not only satisfies the structural constraints and BR2 constraints but also many of the other hard constraints in Section 3.2. In Section 3.3 we will describe the used ILP that was heavily influenced by Lamas-Fernandez et al. [LFMSP21] who won the ITC2021 as well as our novel additions to a multi-stage approach. Afterward, in Section 3.4 we will describe our methods for neighborhood selection in regards to neighborhood type, neighborhood size, and exact team and slot selection. In Section 3.5 we will describe some additional heuristic improvements. Finally, in Section 3.6 we will show the complete algorithm including pseudo-code.

3.1 Neighborhood Types

In total, we analyze ten different neighborhood types. Four of the neighborhoods (Slots, Teams, Team Pairs, and Combi) are established in the literature, four are simple extensions of the existing neighborhoods (Slots Phased, Teams Phased, Slots Home Away, Teams Home Away) and two are to the best of our knowledge completely novel (Grouping Teams and Grouping Slots). We analyze the effectiveness of the ten neighborhoods in Section 4.2 and determine which subset of neighborhood types is most effective on which size of schedule.

Slots: We select n days (slots) of the tournament and delete the current matches on those days. This neighborhood was used by many previous approaches to this problem [LFMSP21, POW21, VG23a] and is perhaps the most straightforward of all the neighborhoods.

Team Pairs: We select n teams and delete all matches where both participants are part of the n teams. This neighborhood is established in the literature and was also used in state-of-the-art approaches [LFMSP21, POW21].

Teams: We select n teams and delete all matches where one of the participants is part of the n teams. While this neighborhood is very similar to the Team Pairs neighborhood it puts more focus on the n selected teams since all the matches of them are deleted. Phillips et al. [POW21] have demonstrated that the two neighborhoods are working well together and this neighborhood has also been used in other approaches [VG23a].

Combi: This neighborhood combines the Teams and Slots neighborhoods. We select two teams and n slots and delete all matches that either happen on the selected days or those in which one of the two teams is participating. This neighborhood was introduced by Lamas-Fernandez et al. [LFMSP21] and has also been used in the more recent approach by Van Bulck and Goossens [VG23a].

Slots Phased: An adaptation of the Slots neighborhood for phased schedules where n slots are all selected from either the first or second half of the tournament.

Teams Phased: An adaptation of the Teams neighborhood for phased schedules where n teams are selected and all matches in either the first or the second half of the tournament are deleted if they contain one of the teams.

Slots Home Away: In this neighborhood we select n Slots and allow all location swaps for all matches that are part of this slot (which also swaps the location of the rematch that might not be part of the n selected Slots). This neighborhood was designed because Lamas-Fernandez et al. [LFMSP21] reported that globally allowing location swaps creates a very challenging ILP therefore we tried to simplify the problem.

Teams Home Away: This neighborhood is similar to the Slots Home Away neighborhood but instead of allowing swaps on specific days we choose n teams and all matches where one of the n teams plays can change location.

Grouping Teams: In this neighborhood we group all teams that are part of the tournament into groups of size n (if the total amount of teams is not divisible by n then there will be one group of smaller size). We then allow teams that are all part of the same group to switch matches with each other, essentially creating multiple small Team Pairs neighborhoods. Creating multiple groups that are solved simultaneously by the ILP solver allows us to make adaptations to the whole schedule at once without running into the problem of very long solving times. Note that this is also not equal to simply running the n groups sequentially, since assignments inside the groups can depend on assignments of other groups.

Grouping Slots: This neighborhood is similar to the Grouping Teams neighborhood but instead of grouping teams together, we create slot groups of size n (again there might be one smaller group). We then allow matches scheduled on one of the days of the group to be moved to any other day of the group. This again has the same effect as the Grouping Teams neighborhood where we look at the whole schedule at once but limit the number of possible adaptations to reduce runtime. This is similar to running multiple iterations of the Slots neighborhood but also considers dependencies across groups.

3.2 Initial Solution

There are multiple ways [RUdW23] to generate initial schedules that follow the structural constraints of the tournament in polynomial time. Which of those to use is not a trivial decision since it highly depends on the exact constraints of the tournament. One particularly popular approach originally developed by de Werra [dW81] uses a *canonical factorization* to minimize the total amount of breaks in a single round-robin tournament. The reason that this approach is particularly popular is that in a lot of cases, BR2 constraints are amongst the hardest to solve.

By using the canonical factorization and mirroring the days and matches of the tournament for the second half (meaning that if a match (i, j) occurs on the last day of the first half of the tournament match (j, i) will occur on the first day of the second half and vice versa) our resulting initial schedule is guaranteed to contain the least amount of breaks possible.

Proof: Let us call the number of teams n. We know that the first half of the tournament has exactly $n - 2$ breaks [RUdW23]. Mirroring the days of the tournament does not change the number of breaks since a break is defined as a relationship between two consecutive days, where the order of those days has no influence on the existence of a break. Changing the location of all days in the second half of the tournament flips the home-away status off all teams on all days and therefore transforms away breaks into home breaks and vice versa, but this won't change the total amount of breaks. This means that the second half of the tournament also has $n-2$ breaks which we know is the least amount possible. Finally, between the last day of the first half of the tournament and the first day of the second half of the tournament, there won't be any breaks since the matchups are the same with flipped home-away status therefore each team that played at home on the last day of the first half plays away on the first day of the second half. Therefore, the overall tournament has $2n-4$ breaks.

However, using this method also has some negative side effects compared to simply copying the first half of the tournament and flipping the locations of all games to create the second half of the tournament creating 6n-4 breaks [RUdW23]. In particular, using the mirroring approach means that every team will play back-to-back against the opponent they face in the last round of the first half of the schedule. For many tournaments, this is undesirable which is why many of the instances use separation constraints (SE1). We decided to use the mirroring of days only for tournaments that do not use SE1 constraints.

To further improve starting schedules we apply a heuristic that reduces the number of hard GA1 violations without increasing the number of breaks. The pseudo-code is provided in Algoritm 3.1 To understand how exactly this works we first have to clarify that in the *canonical factorization* by de Werra [dW81] they used a sorted list of teams from 1 to n and created the schedule based on the position of each team in the list. Clearly, the ordering of the list does not change the total amount of breaks as switching the positions of two teams in that list is equivalent to switching two teams in the final schedule. What we then do is iterate over the hard GA1 constraints (Line 2) and see if the current ordering of the list would fulfill them by finding all team pairs that currently contribute to the left-hand side (LHS) of the GA1 ILP constraint (Line 3) and the ones that could potentially contribute but currently don't (Line 4). If the constraint is violated we would switch the positions of teams to increase (Line 17-26) or decrease (Line 7-16) (depending on if the constraint is violated because of having more games than the maximum scheduled or less than the minimum) the number of games that affect the LHS. After a constraint is fulfilled we fix the positions of all teams in the list and continue with the next constraint. This guarantees that we do not violate the constraint when trying to satisfy others. However, this also makes it possible that we can't fulfill a GA1 constraint because of previously fixed teams. While this approach is very trivial and results in most of the list being fixed before all GA1 constraints are satisfied it heuristically reduces the amount of GA1 violations without increasing the overall amount of breaks. This approach could be improved by adding methods like backtracking, however, since the analysis in Section 4.3 shows that the effect of reducing GA1 constraints is only minimal we did not explore further improvements.

Additionally since creating initial schedules is very quick we create 1000 schedules and choose the one with the best weighted objective value. To weight the objective value we use the analysis by Rosati et al. [RPGS22] where they analyzed good weights for the individual hard constraints in the context of simulated annealing. The weights of the soft constraints remained unchanged.

Algorithm 3.1: Canonical Pattern with GA1 reduction

```
Input: Random order list of teams T, list of GA1 constraints G
Output: Initial DRRST schedule with min breaks and reduced GA1 hard
        violations S
fixed_teams ← list(); # list of teams with fixed position in T starts empty
for g in G do
    c_LHS ← get_LHS_contributions(g, T); # current LHS contributions
    p_LHS ← get_potential_LHS_contributions(g, T); #
      potential LHS contributions that currently do not contribute to LHS
    count ← 0;
    while len(c_LHS) > g[max] or len(c_LHS) < g[min] do
        if len(c_LHS) > g[max] then
            for team_pair in c_LHS do
                if not both_teams_fixed(team_pair, fixed_teams) then
                    success ←
                      move_team_in_pair_to_non_contributing_position(team_pair,
                      p_LHS, T);
                    if success then
                        break for;
                    end
                end
            end
        end
        if len(c_LHS) < g[min] then
            for team_pair in c_LHS do
                if not both_teams_fixed(team_pair, fixed_teams) then
                    success ←
                      move_team_in_pair_to_contributing_position(team_pair,
                      p_LHS, T);
                    if success then
                        break for;
                    end
                end
            end
        end
        c_LHS ← get_LHS_contributions(g, T);
        p_LHS ← get_potential_LHS_contributions(g, T);
        count ← couint + 1
        if count > 100 then
            break for;
        end
    end
    if count ≤ 100 then
        fix_team_in_GA1_constraint(g, fixed_teams);
    end
end
S ← getScheduleFromOrder(T); #
  use canonical factorization algorithm by de Werra [dW81]
return S
```

3.3 Linear Programming Formulation

For the repair step of our ALNS, we use ILP, specifically Gurobi 10.0.1. For the basic ILP, we use the formulation by Lamas-Fernandez et al. [LFMSP21] since at the point of writing the thesis this is the most successful approach that made use of an ILP. We also make some adaptions in order to solve the neighborhood's Grouping Teams and Grouping Slots. Further, we develop a new multistage approach where the creation of the first feasible solution is split into a separate stage for each constraint type. We will describe in detail which constraints are active at what stages.

3.3.1 Basic ILP

The basic ILP essentially deals with three different problems. First, it ensures that the schedule is a valid time-constraint (possibly phased) double-round robin schedule. Second, it encodes the nine different constraint types into ILP constraints which are either hard and therefore have to be fulfilled or soft. If they are encoded as soft constraints we use deviation variables that show how far we are from fulfilling the constraint. Finally, the ILP's objective function uses the deviation variables to calculate the objective value of the whole schedule according to the rules of the ITC2021 [VG23b].

We will now list the objective function, variables, and constraints using the same formulation as Lamas-Fernandez et al. [LFMSP21] that have again used established formulations [DGM+07] for the DRRST problem. T represents the set of all Teams, S represents the set of all Slots, S' the time slots in the first half of the tournament and C represents the set of all constraints.

Variables and Constants:

- First we use binary variables x_{ijs} to denote if in slot s there is a game between team i and team j happening at the home venue of team i. If that is the case the variable is set to 1 otherwise it is set to 0.

$$x_{ijs} \qquad \forall i, j \in T \mid i \neq j, \forall s \in S \quad (3.1)$$

- Next we use binary variables b^H_{is}, b^A_{is}, b^{HA}_{is} to denote if in slot s team i has a home break, away break, or either home or away break. If that is the case the respective variable is set to 1.

$$b^H_{is} \qquad \forall j \in T, \forall s \in S \quad (3.2)$$

$$b^A_{is} \qquad \forall j \in T, \forall s \in S \quad (3.3)$$

$$b^{HA}_{is} \qquad \forall j \in T, \forall s \in S \quad (3.4)$$

- For the separation constraints we have to know if the game (i, j) or (j, i) happens first. For this, we use the binary variables y_{ij}. y_{ij} is 1 if match (i,j) is scheduled at an earlier slot than match (j, i).

$$y_{ij} \qquad \forall i, j \in T \mid i \neq j \quad (3.5)$$

- Finally for each constraint c there is a constant t_c and an integer variable d_c that denote the threshold and deviation respectively. The threshold represents a certain maximum or minimum that is part of the constraint e.g. for CA1 constraints this could be the maximum number of home games a team is allowed to play during a set of slots. The deviation variable represents how far away we are from that threshold. Meaning that if we use the same example and the team is allowed to play three games at home but the schedule contains five such games d_c has to be two. Note also that a constraint might require multiple constants and variables to be represented but in that case, we can always take the sum of the variables to get an overall constraint variable.

$$t_c \qquad c \in C \tag{3.6}$$

$$d_c \qquad c \in C \tag{3.7}$$

Constraints:

The constraints can be split into structural constraints (including linking of variables) and encodings of the nine different constraint types. We start by listing the structural constraints:

- Every team has to play at every time slot:

$$\sum_{j \in T \setminus \{i\}} x_{ijs} + x_{jis} = 1 \qquad \forall i \in T, \forall s \in S \tag{3.8}$$

- Every match is part of the schedule:

$$\sum_{s \in S} x_{ijs} = 1 \qquad \forall i, j \in T \mid i \neq j \tag{3.9}$$

- For phased tournaments exactly one of the games of each matchup is in the first half of the tournament:

$$\sum_{s \in S'} x_{ijs} + x_{jis} = 1 \qquad \forall i, j \in T \mid i \neq j \tag{3.10}$$

- Linking x_{ijs} variables with b^H_{is} variables:

$$\sum_{j \in T \setminus \{i\}} x_{ijs} + x_{ij(s+1)} \leq 1 + b^H_{is} \qquad \forall i \in T, \forall s \in S \tag{3.11}$$

- Linking x_{ijs} variables with b^A_{is} variables:

$$\sum_{j \in T \setminus \{i\}} x_{jis} + x_{ji(s+1)} \leq 1 + b^A_{is} \qquad \forall i \in T, \forall s \in S \tag{3.12}$$

- Linking b^A_{is} and b^H_{is} variables with b^{HA}_{is}:

$$b^A_{is} + b^H_{is} = b^{HA}_{is} \qquad \forall i \in T, \forall s \in S \quad (3.13)$$

- Linking x_{ijs} variables with y_{ij} variables:

$$\sum_{s \in S} s(x_{jis} - x_{ijs}) \leq M y_{ij} \qquad \forall i, j \in T \mid i \neq j, M \geq |S| \quad (3.14)$$

$$\sum_{s \in S} s(x_{ijs} - x_{jis}) \leq M(1 - y_{ij}) \qquad \forall i, j \in T \mid i \neq j, M \geq |S| \quad (3.15)$$

Next, we need one or more constraints for each of the nine constraint types that exist for the problem. The description of the constraint types can be found in Section 2.1 as well as the paper of the ITC2021 [VG23b]. We will refer to slots and teams that are part of a specific constraint c as S_c and T_c respectively. Further, if a constraint focuses on a specific team i we will refer to that team as i_c, and if the constraint specifies a set of games we call them G_c.

- The set of CA1 constraints consists of two subsets. Those concerning the maximum number of home games ($CA1^H$) and those concerning the maximum number of away games ($CA1^A$). The first is encoded as constraints 3.16 the latter is encoded as 3.17.

$$\sum_{j \in T \setminus \{i\}} \sum_{s \in S_c} x_{ijs} \leq t_c + d_c \qquad i = i_c, \forall c \in CA1^H \quad (3.16)$$

$$\sum_{j \in T \setminus \{i\}} \sum_{s \in S_c} x_{jis} \leq t_c + d_c \qquad i = i_c, \forall c \in CA1^A \quad (3.17)$$

- Similar to CA1 constraints CA2 constraints are also split into multiple subsets ($CA2^H$, $CA2^A$, $CA2^{HA}$), again $CA2^H$ and $CA2^A$ constraints are referring to home games and away games while $CA2^{HA}$ constraints are referring to both.

$$\sum_{j \in T_c} \sum_{s \in S_c} x_{ijs} \leq t_c + d_c \qquad i = i_c, \forall c \in CA2^H \quad (3.18)$$

$$\sum_{j \in T_c} \sum_{s \in S_c} x_{jis} \leq t_c + d_c \qquad i = i_c, \forall c \in CA2^A \quad (3.19)$$

$$\sum_{j \in T_c} \sum_{s \in S_c} x_{ijs} + x_{jis} \leq t_c + d_c \qquad i = i_c, \forall c \in CA2^{HA} \quad (3.20)$$

- CA3 constraints are split just like CA2 constraints. We use an additional set $K = \{0, \ldots, |S| - I_c\}$ where I_c is referring to the size of the interval that is part of the CA3 constraint.

$$\sum_{j \in T_c} \sum_{s=k+1}^{k+I_c} x_{ijs} \leq t_c + d_c \qquad i = i_c, \forall c \in CA3^H, \forall k \in K \quad (3.21)$$

$$\sum_{j \in T_c} \sum_{s=k+1}^{k+I_c} x_{jis} \leq t_c + d_c \qquad i = i_c, \forall c \in CA3^A, \forall k \in K \quad (3.22)$$

$$\sum_{j \in T_c} \sum_{s=k+1}^{k+I_c} x_{ijs} + x_{jis} \leq t_c + d_c \qquad i = i_c, \forall c \in CA3^{HA}, \forall k \in K \quad (3.23)$$

- CA4 constraints consist of two different groups of constraints: Those that are applied globally ($CA4_G$) and those that specify certain time slots and the constraint has to hold on every single time slot in the set ($CA4_E$). Both of those groups can then deal with either home games, away games, or both similar to CA2 and CA3 constraints. Also since there are two different sets of teams in each CA4 constraint we will refer to them as T_{c1} and T_{c2}.

$$\sum_{i \in T_{c1}} \sum_{i \in T_{c2}} \sum_{s \in S_c} x_{ijs} \leq t_c + d_c \qquad \forall c \in CA4_G^H \quad (3.24)$$

$$\sum_{i \in T_{c1}} \sum_{i \in T_{c2}} \sum_{s \in S_c} x_{jis} \leq t_c + d_c \qquad \forall c \in CA4_G^A \quad (3.25)$$

$$\sum_{i \in T_{c1}} \sum_{i \in T_{c2}} \sum_{s \in S_c} x_{ijs} + x_{jis} \leq t_c + d_c \qquad \forall c \in CA4_G^{HA} \quad (3.26)$$

$$\sum_{i \in T_{c1}} \sum_{i \in T_{c2}} x_{ijs} \leq t_c + d_c \qquad \forall s \in S_c, \forall c \in CA4_E^H \quad (3.27)$$

$$\sum_{i \in T_{c1}} \sum_{i \in T_{c2}} x_{jis} \leq t_c + d_c \qquad \forall s \in S_c, \forall c \in CA4_E^A \quad (3.28)$$

$$\sum_{i \in T_{c1}} \sum_{i \in T_{c2}} x_{ijs} + x_{jis} \leq t_c + d_c \qquad \forall s \in S_c, \forall c \in CA4_E^{HA} \quad (3.29)$$

- GA1 constraints have a lower bound t_{cL} and an upper bound t_{cU}. The lower and upper bounds are encoded using separate constraints.

$$\sum_{(i,j) \in G_c} \sum_{s \in S_c} x_{ijs} \leq t_{cU} + d_c \qquad \forall c \in GA1 \quad (3.30)$$

$$\sum_{(i,j) \in G_c} \sum_{s \in S_c} x_{ijs} \geq t_{cL} + d_c \qquad \forall c \in GA1 \quad (3.31)$$

- BR1 constraints are split into three sets like many of the capacity constraints. The difference is that BR1 constraints differentiate between home breaks ($BR1^H$), away breaks($BR1^A$), or both($BR1^{HA}$).

$$\sum_{s \in S_c} b_{is}^{H} \leq t_c + d_c \qquad i = i_c, \forall c \in BR1^H \quad (3.32)$$

$$\sum_{s \in S_c} b_{is}^{A} \leq t_c + d_c \qquad i = i_c, \forall c \in BR1^A \quad (3.33)$$

$$\sum_{s \in S_c} b_{is}^{HA} \leq t_c + d_c \qquad i = i_c, \forall c \in BR1^{HA} \quad (3.34)$$

- BR2 constraints only consist of constraints that focus on both home and away breaks.

$$\sum_{i \in T_c} \sum_{s \in S_c} b_{is}^{HA} \leq t_c + d_c \qquad \forall c \in BR2^{HA} \quad (3.35)$$

- FA2 constraints deal with two individual teams which we will call i_{c1} and i_{c2}.

$$\sum_{j \in T_c} \sum_{s=1}^{s'} (x_{ijs} + x_{i'js}) \leq t_c + d_c \qquad i = i_{c1}, i' = i_{c2}, \forall s' \in S_c, \forall c \in FA2 \quad (3.36)$$

- SE1 constraints can be encoded using a single constraint.

$$\sum_{s \in S} s(x_{ijs} + x_{jis}) \geq t_c + 1 - d_c - My_{ij} \qquad \forall i, j \in T_c, \forall c \in SE1, M \geq |S| \quad (3.37)$$

Objective Function:

Each constraint $c \in C$ (with C being the set consisting of all nine constraint types) has an assigned penalty w_c that is applied for each unit of deviation (saved in the corresponding d_c variables) the goal is to minimize the overall penalty. The corresponding function is given in 3.38

$$\min \sum_{c \in C} w_c d_c \quad (3.38)$$

Gurobi Parameters:

To use Gurobi to its fullest potential it is important to choose meaningful parameters. This can significantly increase the performance of the models. The most important parameters for which we did not just use the default are as follows:

- **TimeLimit:** We chose two times the current time target which is explained in detail in Section 3.4.

- **Cutoff:** Is set as the objective value of the previous iteration minus one. This essentially tells Gurobi to stop searching for solutions once it has proven that it is definitely not better than our current objective value, which significantly reduces the average time spent on each neighborhood since we are not always trying to prove optimality.

- **Threads:** For all experiments except the final evaluation in Section 4.5.1 we used a single thread for the final evaluation we used two threads. While more threads would increase performance it would also limit us in how many experiments we can conduct since our resources are limited. Also, the performance does not increase linearly with the amount of threads used so we would expect diminishing returns if we used a very high amount of threads.

- **MIPFocus:** We experimented with different values but found that MIPFocus=1 which tells the solver to find feasible solutions as quickly as possible rather than focusing on proving optimality got the best results. The same results were also reported by Phillips et al. [POW21].

3.3.2 Encode Neighborhoods Into ILP

Most neighborhoods are straightforward to implement using ILP. As described in many previous approaches [LFMSP21, POW21, VG23a] we fix the values of all variables that are not part of the neighborhood, meaning we assign the value one to the game variable x_{ijs} if a game is currently scheduled on day s between teams i and j at the home venue of team i (using constraints of the form $x_{ijs} = 1$) and leave the variables that are part of the neighborhood free, essentially deleting the prior knowledge of this part of the schedule to find a possibly better alternative.

Encoding the new neighborhoods Grouping Teams and Grouping Slots involves fixing a lot of values to zero rather than one since we allow the whole schedule to change simultaneously while restricting the possible assignments for each game.

We will now go into more detail on how to encode each neighborhood.

Slots/Slots Phased: We leave the variables x_{ijs} free for all slots $s \in S$ where S are the slots that are selected as part of the neighborhood. For all $x_{ijs'}$ with $s' \notin S$ we add constraints that fix the game variables to one as described above.

Teams/Teams Phased: All variables x_{ijs} where either team i or j $\in T$ with T being the teams that are part of the neighborhood are left free (if they are part of the selected half of the tournament for the phased variant). All other variables x_{ijs} are fixed as described above.

Team Pairs: Similar to the Teams neighborhood with the exception that both team i and team j have to be part of T.

Combi: We combine the Teams neighborhood with the Slots neighborhood and leave all variables free that would be free in at least one of the two neighborhoods and fix all others.

Slots Home Away/Teams Home Away: We fix the same variables to one as in the Slots/Teams neighborhood. For the Slots/Teams that are part of the neighborhood we look at the current schedule and for each match (i, j) currently scheduled on day s we leave the variables x_{ijs} and x_{jis} free but fix all other matches that do currently not occur at that time slot to zero.

Grouping Teams: In this neighborhood we fix variables x_{ijs} to zero unless teams i and j are in the same group or either x_{ijs} or x_{jis} is one in the currently best-known schedule. All other variables are free.

Grouping Slots: For each group consisting of n slots that create a set S, we fix all variables x_{ijs} to zero if team i and j do not play against each other at the home venue of i at any of the slots in S. This means any game that is scheduled during one of the slots in S is free to be scheduled on any other slot in S but no slot outside of S.

3.3.3 Multi-stage Approach

Our ALNS consists of two main stages. First, we try to create a feasible solution and afterwards we enter the improvement phase.

Our novel idea is to split the first stage into 7 separate stages, one for each constraint type that appears as a hard constraint. The idea behind this is that some constraint types become much harder to fulfill if the schedule has become very rigid from fulfilling other constraints. Concretely we noticed in our experiments that GA1 constraints would often take a very long time to be fulfilled if a lot of other constraints already restrict the schedule. From a logical perspective, this makes sense since a GA1 constraint often requires a specific game to be scheduled in a specific time slot which can be hard if a lot of other constraints are affected by that time slot or game already. We also support our claims by experiments in Section 4.3, where we compare this setup to several variations including approaches that deal with all hard constraints at the same time.

To solve this issue we arrange the constraints according to the analysis done by Rosati et al. [RPGS22] where they analyzed what weights to use for each constraint type in simulated annealing. Concretely they assigned a weight of 1 to 10 to each hard constraint which led to the algorithm prioritizing the constraint with high weights. While we could have done the same and changed the weights of each constraint, we decided to go a step further and solve the different constraint types sequentially. The resulting order (excluding SE1 and FA2 because they do only appear as soft constraints) from first solved to last solved is:

$$"GA1", "CA2", "CA4", "CA1", "BR2", "CA3", "BR1"$$

What is interesting to observe about this order is that it roughly sorts the constraints based on how easy they are to maintain once they are fulfilled for the first time. To go into more detail about how we solve the feasibility stage we will explain step by step how the ILP changes over time until we get a feasible solution. The corresponding pseudo-code can be found in Section 3.6:

1. Start with a randomized initial schedule like described in Section 3.2.
2. Take the first element from the list and add all the hard constraints of that type as soft constraints.
3. Apply neighborhoods until the objective function becomes 0 (while potentially changing parameters over time).
4. Change the soft constraints to hard constraints.
5. Delete the first element from our list of unsolved constraint types.
6. Go back to step 2 until the list is empty and therefore the solution becomes feasible.

As a further improvement to this strategy, we multiply the weights of all constraints by a factor of 10000 and add another set of soft constraints that work in the following way: Take the set of all hard capacity and break constraints, and replace the constant threshold t_c by zero. We exclude GA1 constraints from this process because they specify a minimum value as well and it is unclear if a value close to the minimum is good for a schedule. This process has the effect of creating more room for change in the schedule. So for example, if the BR2 constraints are already fulfilled we will still try to lower the overall amount of breaks and this might make it possible to later make a change in the schedule that increases the number of breaks that would not have been possible if the number of breaks was close to the maximum. Additionally adding the soft constraint makes it so that while the initial constraint types are processed we do have the incentive to keep as many hard constraints satisfied as possible even if they would only be added to the model with the concrete threshold values at a later point. A thorough evaluation of this approach can be found in Section 4.3.

For the improvement stage, we keep all hard constraints and add all soft constraints. Note that we could also decide to simply set the penalty of hard constraints to a higher value so that we could still enter infeasible regions like Lamas-Fernandez et al. [LFMSP21], however, this comes with a significant decrease in ILP performance which we suspect is due to the much larger feasible space. We then apply neighborhoods, looking for improvements, until the time runs out. As the schedule improves and gets closer to an optimal objective value it becomes harder to find further improvements. That is why we on the one hand have to learn over time which neighborhoods have the highest chance of success in finding improvements and on the other hand we have to look into bigger neighborhoods when the smaller ones fail to find better schedules. This is why we have to carefully adapt parameters over time. What those parameters are and how we change them over time is explained in Section 3.4.

3.4 Adaptivity

As the objective values get better it becomes harder and harder to find improvements to a solution, which in turn means that we must invest more time for each improvement. Furthermore, with each transformation of the schedule, it becomes unclear if the same neighborhood types keep being effective or if others have a better chance to find enhancements. To solve those problems we use a multi-armed bandit formulation described in Section 3.4.1 to determine which neighborhood type to use at what point and subsequently we use heuristics to determine which (and how many) teams or slots to destroy. What influences said selection is described in Section 3.4.2.

3.4.1 Multi-Armed Bandit Problem

The basic idea of the multi-armed bandit problem is that a bandit with k arms can perform an action with each arm (e.g. pull the lever of a slot machine). Each action gives the bandit a reward that might change over time. The bandit now tries to find out which of its k arms will give him the biggest reward over time. The first formulation of the multi-armed bandit problem stems from Robbins [Rob52] but since then many different variants have been proposed [VM05, SWS$^+$22]. It is not a new idea to use a multi-armed bandit formulation for selecting the neighborhood type in an ALNS. In fact, Phillips et al. [POW21] have used a multi-armed bandit formulation for their ALNS approach in the ITC2021 competition using the Upper Confidence Bound (UCB) formulation [SB18] of the problem.

Because it is unclear to us if the same neighborhoods that are effective for improving the solution quickly when it is still far from optimal are also effective for finding improvements when the schedule is already very good we experiment with two different variants of the multi-armed bandit problem.

The first approach is to use a non-stationary ϵ-greedy variant [Wat89] with optimistic initial rewards of the multi-armed bandit formulation. The reason we decided to experiment with this variant is that initial experiments showed that some neighborhoods lead to very good improvements while the schedule is still far from the optimum but in later stages, other neighborhoods outperform them, we also maintain a fixed exploration rate ϵ (of 10%) since we suspected that even if a neighborhood might be worse on average it can be beneficial to keep exploring with it to escape potential local optima. Concretely, our estimation function for the reward of an action a after the action was already performed t times is calculated as follows:

$$Q_t(a) = \text{initial\_weight} * (1-\alpha)^t + \sum_{i=1}^{t-1} \alpha * (1-\alpha)^{t-i} * r_i \quad (3.39)$$

The initial weight is chosen to be higher than the objective value of the schedule which results in all neighborhood types being explored in the beginning. The variable r_i represents the reward at the i^{th} time the neighborhood type was used.

The second approach is to use the UCB formulation from Sutton and Barto (2.8) [SB18] that is based on the UCB1 algorithm by Auer et al [ACF02]. While the UCB algorithm represents the state-of-the-art of multi-armed bandit formulations it is not ideal if the expected rewards of actions change too much over time [SB18]. This is because the way the UCB algorithm works is to split the expected reward function for each action into two parts. The first part calculates the average reward of the previous times the action was taken while the second part is where the name of the algorithm has its origin because it adds the uncertainty of the calculated average on top of it. The formula from Sutton and Barto (2.8) [SB18] that we use can be found as Equation 3.40.

$$A_t = \mathbf{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \quad (3.40)$$

Here $Q_t(a)$ represents the expected reward when taking action a which is the average of all past instances where this action was taken, t is the total amount of actions taken up to this point, and $N_t(a)$ is the amount of time action a was taken up to this point.

Additionally, as time progresses, rewards tend to get smaller which might have a negative effect on neighborhood selection because the hard improvements later on are not rewarded as much as the easier, usually bigger, improvements at the beginning. While this still gives each neighborhood the same chance it sometimes happens that a neighborhood that is usually not making a lot of improvements gets one lucky big improvement at the start and is then over-selected for a long time. To balance this out we experimented with multiplying rewards by an exponential function (exponential in the number of attempted neighborhoods) with a very low base (1.025) to balance out the exponential decrease of improvements that were shown by Phillips et al. [POW21] and Fonseca and Toffolo [FT22]. However, to prevent too high rewards we limited the multiplier to be at most 100 which is reached after 187 iterations. Therefore, we mainly balance out the very early rewards. We also experimented with other functions as a multiplier (logarithmic, linear, and exponential with various bases) but the one described above led to the best results. We did however only conduct the experiments on a small subset of the Early instances so it is possible that different functions lead to even better results when looking at the wider instance space. The overall effects of this method seem to be rather small and the main benefit is that we do not over-select a neighborhood for quite as long just because it was successful in improving the schedule at a time when every other neighborhood could have been just as successful.

Our experiments have shown that the UCB method led to significantly better results. While we did not anticipate this for this specific problem because of the perceived change of effectiveness of the different neighborhood types it is also not completely unexpected since the UCB method generally is seen as the state-of-the-art of the bandit methods [SB18]. A formal comparison between ϵ-greedy and UCB multi-armed bandit methods as well as random neighborhood type selection can be found in Section 4.4.

3.4.2 Selecting Teams and Days

After selecting the neighborhood type the next step is to determine the exact neighborhood. To determine which and how many teams and slots to use, we made adaptations and improvements to the winning approach by Lamas-Fernandez et al. [LFMSP21].

To determine what teams and slots will become part of the neighborhood we iterate over all violated constraints and determine which teams and slots contribute to the left-hand side (LHS) of the ILP equations that describe the constraints and add up the total amount of violation contributions for each team and slot. This means that we find out which teams and slots are most responsible for the violations of constraints. This gives us a map structure M that maps each team and slot to the number of constraints that are violated partially because of the matches of that team or slot. A high number in M, therefore, indicates that there is a high potential for improvements if the team or slot is destroyed.

Next, we determine the size of the destroyed neighborhood. To do that we work with a time target that increases and decreases over time based on the number of found improvements. Specifically, every time we find an improvement to the current schedule the time target is multiplied by a factor t_d, and if we don't find any improvements for k iterations the time target is multiplied by a factor t_i. The values for t_d, t_i, and k are determined during parameter tuning. For each neighborhood, we then maintain a list of reconstruction times, and if the average reconstruction time of the last n iterations is more than 5% bigger than the time target we decrease the size of the neighborhood. Vice versa if the average reconstruction time is more than 5% smaller than the time target we increase the neighborhood size. Initial testing using this approach showed that the fluctuation in neighborhood sizes was too big and we wasted a lot of time on either too small or too big neighborhoods. To prevent this we keep separate lists of reconstruction times for each possible size of each neighborhood type. We then only allow an increase in neighborhood size if both the overall average of reconstruction times using this neighborhood type is 5% smaller than the time target and the current average reconstruction time using the current size of the neighborhood is also smaller than the time target. Similarly, if the average reconstruction time is 5% bigger, the average reconstruction time using the current size also has to be bigger to trigger a decrease of the neighborhood size. Finally, we do not allow two consecutive increases or decreases in neighborhood size but instead, require at least two iterations with the same neighborhood size in order to have a little more stability. Using those improvements we observe stable neighborhood sizes that almost always switch between the two sizes whose average reconstruction time surrounds the time target. This approach has the effect that the neighborhoods are evaluated fairly using the multi-armed bandit formulation without having the reconstruction time as a direct factor since all neighborhoods use the same time on average. To the best of our knowledge, this is a novel approach for selecting the size of neighborhoods and brings significant benefits compared to the more common method of directly increasing or decreasing neighborhood size. The main benefits are that, as previously mentioned, each neighborhood takes the same time on average making it easier to compare them fairly as well as a more fine-grained setting for neighborhood sizes since it now becomes possible to choose an arbitrary time target that lies somewhere between two neighborhood sizes, which implicitly makes the algorithm choose both neighborhood sizes a certain portion of the time. Each slight increase or decrease in time target then changes said portion so that one of the two sizes gets used slightly more and the other a little less. This is beneficial since it is not always clear if the smaller neighborhood size is still worth exploring or if it is better to invest more resources and explore the bigger neighborhood.

Now that we have both the map structure with global information about promising teams and slots to destroy and the size and type of the destroyed neighborhood, we have to determine the actual teams and slots that we will select. To do that we select a random violated constraint and look at its LHS contributions. Depending on the neighborhood type we either look at only teams, only slots, or both. If the amount of contributing teams or slots k is smaller than the determined neighborhood size n we will destroy all k of them and then probabilistically select $n - k$ from the remaining teams or slots. The probabilistic selection works by looking at M and then doing a weighted random selection with the weights being the squares of the violation contributions + a small constant. This has the effect that teams and slots that are part of many violations are selected more frequently but at the same time, we do not completely neglect other parts of the schedule. Similarly, when k is bigger than n we use the probabilistic weighted selection to determine which n of the k teams to include in the neighborhood. To the best of our knowledge, this is the first attempt at using global information in the form of M to select neighborhoods for a LNS and is a potential improvement to the approach by Lamas-Fernandez et al. [LFMSP21] that performed random selections based on the LHS of a single violated constraint.

It should also be noted that the neighborhood selection above does not apply to the Grouping Teams and Grouping Days Neighborhood, since those two neighborhoods will always be applied to the whole schedule using random groups. The size of the groups is determined in the same way as for all the other neighborhoods.

In Section 4.4 we analyze the effects of using LHS contributions and our map structure for team and slot selection.

3.5 Heuristic Improvements

While the general approach has been described in the previous sections there are some improvements to make the ALNS more effective.

Firstly, since we are putting more weight on some teams than others we found that we would sometimes look at the exact same neighborhoods that were already determined to be ineffective without much of the schedule having changed since the last usage of that neighborhood. To prevent that we introduce a Taboo List which is part of the Taboo Search metaheuristic developed by Glover [Glo89]. What this does is to remember the last n selected neighborhoods and prevent them from being used again until enough of the schedule has changed so that there is a good chance that the neighborhood can lead to further improvements. The parameter n is determined during parameter tuning.

The second problem we encountered was that we would get stuck in local optima. To escape such a local optimum we allowed worse solutions after a certain amount of iterations (determined by parameter tuning). In order to create a promising worse solution we select one of the unfulfilled soft constraints and increase the penalty using the following formula:

$$w_c^{new} = \max(100/d_c, w_c + 10) \tag{3.41}$$

This new weight guarantees that the penalty is increased by at least 10 points per current deviation, but potentially the weight is increased to up to 100 if the current deviation is only one. The constants in the equation above were carefully manually selected such that the objective value does not increase so much that it would take a long time to potentially find a better solution but also the penalty of the changed constraint increases enough so that it is quickly solved by the ILP (unless solving the constraint would involve violating a lot of other constraints, in which case the constraint is usually a bad choice). After changing the weights and saving the currently best-known solution we continue the optimization with the changed weight for k iterations. After the k iterations we will reset the weight of the constraint and continue for another n iterations. If at any point we find a schedule that is better than the best-known schedule (evaluated using the original weight w_c) we will immediately reset the weight of the constraint and continue our optimization. If after the n iterations we still have not found a better schedule than the best-known schedule we will reset to the best-known schedule, try to optimize it for a small number of iterations (we got the best results using only 5 iterations here), and then choose another constraint. During parameter tuning, we found that once a schedule reaches the point where penalties are changed it is very time-consuming to find improvements by attempting to improve the best-known schedules. While increasing the size of the neighborhood does help with that, it also leads to much longer reconstruction times. Therefore it is more effective to go into a worse schedule like described above and then try to improve that to go beyond the best-known schedule. It is noteworthy that for quite a few of the iterations, the schedule with the changed penalties is still the same as the best-known schedule since it takes a while for the ILP to find a way to solve the constraint with the changed penalty. Therefore, we spend a lot more time still trying to improve on the best-known schedule than apparent at first glance. Good values of k and n are also determined during parameter tuning.

An interesting fact about this approach is that it has a lot of similarities with what a human might try when manually optimizing the schedule. While we have not conducted a survey with experts who have manually tuned such schedules before, this is how we would personally approach the task:

- Start with a schedule that is already fairly good.
- Select a constraint that looks promising. (Resembles our random constraint selection in heuristic.)
- Try to somehow fulfill the constraint changing up a bit of the schedule. (In the heuristic this happens through the penalty change.)
- If the schedule looks too messed up after the fix change it back. (We do not allow such a schedule because we implicitly limit how bad it can get by deciding on the new penalty.)
- Try to fix everything that got worse because of the changes made. (The $k + n$ iterations we spend on the worse schedule)
- If the result does not look better after a while go back to the schedule before any constraint was selected. (Our reset to the best-known solution after $k + n$ iterations)
- If at any point the schedule is better than anything that was seen before use this schedule for all further changes.

We believe that a real survey with experts could potentially show us new ways to either improve the above-mentioned approach or give us ideas for potentially even better ways to tackle the problem. While this is not part of the scope of this thesis it is something to look into for future research.

One promising idea with this approach is to parallelize it, changing the penalty of a different violated constraint in each branch. We could then run each branch for a certain amount of iterations before choosing the one with the biggest improvement. This is particularly promising as we were not able to identify any patterns that could help us select a promising constraint that has a high chance of leading to an improvement. However, this might be possible using some methods from Reinforcement Learning. Evaluating such a parallel approach, and identifying good constraints to destroy goes beyond the scope of this thesis. Nevertheless, we hope to explore both things in the future to potentially further improve the results of our approach. Further, instead of using the constants in formula 3.41 it is likely that a better approach would be to choose the values based on the state of the schedule. specifically its current objective value and previous attempts at penalty changes. However, the constants above work well, and improving the method will be part of future research.

3.6 The Complete ALNS

Now that we have looked at all the individual parts of our ALNS, it is also important to go over the algorithm as a whole to understand how the individual parts interact with each other. Algorithm 3.2 describes how we generate feasible solutions from an initial schedule and then Algorithm 3.3 describes how we optimize feasible schedules. The algorithms have quite a few similarities and share certain sections like the maintenance of neighborhood selection (Lines 17-19 and 10-12), the tabu list (Lines 20-24 and 13-17), updates of the time target (Lines 29-41 and 22-41), changes of neighborhood sizes (Line 42 and 57) as well as the initialization of some structures used for the various parts of the algorithms. Additionally, in Algorithm 3.2 we see how we select the current constraint type focus in Lines 9-16, which affects the ILP used in Line 26. Meanwhile, in Algorithm 3.3 the ILP used in Line 19 stays the same but the penalties of some constraints may change over time as described in Section 3.5 (Lines 42-56) and we have to maintain a best-known schedule since the penalty changes implicitly allow worse solutions.

Algorithm 3.2: Algorithm for creating feasible schedule

Input: Initial schedule created with Algorithm 3.1 S, list of constraints C, list of neighborhood types N, hyperparameters from automatic tuning h, sorted list of constraint types in order of consideration *ctl*

Output: Feasible schedule that fulfills all hard constraints S

```
fct ← list(); # fulfilled constraint types
N_rew ← init_rewards(N); # rewards per neighborhood
N_rec ← init_reconstruction_times(N); # reconstruction times per neighborhood type and size
N_s ← init_neighborhood_sizes(N); # neighborhood sizes
tt ← h[min_time_target] # current time target
lc ← 0; # last change
tabu ← list(); # tabu list
while not is_feasible(S) do
    for ct in ctl do
        if constraint_type_fulfilled(ct, S) then
            ctl.remove(ct) fct.append(ct)
        else
            break for;
        end
    end
    foc_ct ← ctl[0] # currently focused constraint type
    rc ← choose_random_unfulfilled_constraint_of_current_focus(S, foc_ct, C)
    M ← get_map_of_LHS_hard_constraint_deviations_teams_and_slots(S)
    N_u ← select_neighborhood_using_UCB_bandit_selection(N_rew, N_s, rc, M); #
      neighborhood chosen based on reward, current Neighborhood size, a random unfulfilled
      constraint and general state of schedule using UCB multi-armed bandit approach
    if tabu.contains(N_u) then
        continue while;
    else
        tabu.append_uppdate_iterations_and_delete_old(N_u, h[tabu_length])
    end
    S_des ← destroy_schedule(S, N_u);
    S, reward, time ← repair_schedule(S_des, fct, foc_ct, C); # repair schedule using Gurobi
      with ILP described in Section 3.3 and extra soft constraints; reward describes the
      improvement in objective value
    N_rew ← update_rewards(N_rew, N_u, reward);
    N_rec ← update_reconstruction_times(N_rec, N_u, time);
    if reward = 0 then
        lc ← lc + 1;
        if lc > h[iter. without change before increase] and
        tt < h[max_time_target] then
            tt ← tt * h[time_change_bigger];
            lc ← 0;
        end
    else
        lc ← 0;
        if tt > h[min_time_target] then
            tt ← tt * h[time_change_smaller];
        end
    end
    N_s ← update_neighborhood_sizes(N_rec, tt);
end
return S
```

Algorithm 3.3: Algorithm for optimizing feasible schedules

```
Input: Feasible schedule created with Algorithm 3.2 S_best, list of constraints C, list of
       neighborhood types N, hyperparameters from automatic tuning h
Output: Optimized schedule that fulfills all hard constraints S
N_rew ← init_rewards(N); # rewards per neighborhood
N_rec ← init_reconstruction_times(N); # rec. times per neighborhood type and size
N_s ← init_neighborhood_sizes(N); # neighborhood sizes
tt ← h[min_time_target] # current time target
lc ← 0; # last change
tabu ← list(); # tabu list
li ← 0; # last improvement
pc ← False; # indicates if a penalty is currently changed
while not reached_time_limit() do
    rc ← choose_random_unfulfilled_constraint(S, C)
    M ← get_map_of_LHS_hard_constraint_deviations_teams_and_slots(S)
    N_u ← select_neighborhood_using_UCB_bandit_selection(N_rew, N_s, rc, M);
    if tabu.contains(N_u) then
        continue while;
    else
        tabu.append_uppdate_iterations_and_delete_old(N_u, h[tabu_length])
    end
    S_des ← destroy_schedule(S, N_u);
    S, reward, time ← repair_schedule(S_des, C); # using Gurobi with ILP described in
      Section 3.3
    N_rew ← update_rewards(N_rew, N_u, reward);
    N_rec ← update_reconstruction_times(N_rec, N_u, time);
    if reward = 0 then
        li ← li + 1;
        lc ← lc + 1;
        if lc > h[iter. without change before increase] and
        tt < h[max_time_target] then
            tt ← tt * h[time_change_bigger];
            lc ← 0;
        end
    else
        if objective_value(S) < objective_value(S_best) then
            S_best ← S; # save best-known schedule
            C ←reset_penalty_changes(C); # if already reset nothing happens
            pc ← False;
        end
        li ← 0;
        lc ← 0;
        if tt > h[min_time_target] then
            tt ← tt * h[time_change_smaller];
        end
    end
    if li > h[iter. before penalty changes] and not pc then
        li ← 0;
        C ←change_penalty_of_random_soft_constraint(C);
        pc ← True;
    end
    if li > h[iter. before reset of penalty changes] and pc then
        C ←reset_penalty_changes(C);
        pc ← False;
    end
    if li > h[max iter. before resetting to best-know] and S != S_best then
        li ← h[iter. before penalty changes] - 5;
        C ←reset_penalty_changes(C);
        pc ← False;
        S ← S_best;
    end
    N_s ← update_neighborhood_sizes(N_rec, tt);
end
return S_best
```

CHAPTER 4

Computational Results

In this chapter, we will evaluate the ALNS described in the previous section. We will start by describing the instances and our general setup in Section 4.1. We will then discuss how we automatically tuned the parameters of our ALNS using state-of-the-art parameter tuning software in Section 4.2. Next, we will analyze the performance of our multi-stage approach by comparing it to other approaches in Section 4.3 and look into the impact of our strategies for adaptivity in Section 4.4. Finally, we will evaluate our ALNS using 45 instances and compare the results to other state-of-the-art methods in Section 4.5.

4.1 Setup

4.1.1 Instances

The instances we use come from the ITC2021. The paper [VG23b] by the organizers of the competition describes exactly how the instances featured in the competition were generated. To create a diverse set of instances that offers different challenges they used instance-space analysis [SL12, SBWL14, SB15b] which is a framework that tries to visualize similarities and differences between instances of a problem by distributing them in a 2D space, grouping similar instances together. They checked for each problem that it can not be easily solved using modern algorithms, verified that all problems do have feasible solutions, and tried to model problems that are as similar to real-world instances as possible.

Their analysis resulted in 45 instances that were split into three sets of 15 each. The sets are called Early, Middle, and Late and each of the sets tries to cover the full instance space using three instances of size 16, six of size 18, and six of size 20, with the size referring to the number of teams in the tournament. Approximately half of the instances (22 out of 45) are phased (meaning each half of the tournament is a single round-robin tournament). Difficulty vise there should be no significant difference between the three sets, they were merely split for the sake of the competition and released at different times. The different release dates resulted in teams having six and a half months to optimize the Early instances, three months for the Middle instances, but only two weeks for the set of Late instances. We list the instances including some metadata about them in Table 4.1. Note that the exact count of soft and hard constraints split by constraint type and for each instance can be found in the paper about the organization of the competition by Van Bulck and Goosens [VG23b].

4.1.2 Testing Environment

All experiments except for the final evaluation were done using exclusively the Early instances. The Early instances represent our training set and were used to find good hyperparameters and to evaluate various strategies for creating feasible solutions. We decided to use only the Early instances in order to avoid overfitting on competition instances. The advantage of using this set is that as described above the Early set covers the instance space fairly well. Further, using this set for tuning is the fairest comparison to the teams that participated in the competition as they would have used this set as well for all initial design choices because it was the only set available for the majority of the competition.

The parameter tuning and evaluation of the multi-stage approach were performed using a VM with 8 processor cores (and 16 threads) of an Xeon Silver processor and 16GB of RAM and up to 3.2 GHz. The final evaluation described in Section 4.5 was done on a 13th Gen Intel i7 13700KF with 16 cores and 24 logical processors that can overclock to up to 5.4 GHz which has significantly better single-core performance and 32GB of RAM (which are not fully utilized). We limited Gurobi to use a maximum of two threads per instance in order to experiment with multiple instances in parallel. Further, to guarantee that the parallel tasks do not interfere with each other we never run more than 8 instances in parallel also guaranteeing that no processor cores have to be shared. Finally, all experiments use Gurobi version 10.0.1.

Table 4.1: Intances of the ITC2021 [VG23b] listed with the amount of hard and soft constraints as well as types of constraints used in each instance and whether the instance is phased or not.

Instance	Phased	Size	Hard	Soft	Types
Early 1	TRUE	16	83	113	BR1, BR2, CA1, CA2, CA4, FA2, GA1, SE1
Early 2	TRUE	16	53	114	BR1, BR2, CA1, CA3, FA2, GA1
Early 3	TRUE	16	148	186	BR1, BR2, CA1, CA2, CA3, FA2, GA1
Early 4	TRUE	18	164	268	BR1, BR2, CA1, CA2, CA4, GA1, SE1
Early 5	TRUE	18	207	587	BR1, BR2, CA1, CA2, CA3, CA4, GA1, SE1
Early 6	TRUE	18	192	797	BR2, CA1, CA2, CA3, CA4, FA2, GA1, SE1
Early 7	FALSE	18	175	1159	BR1, BR2, CA1, CA2, CA3, CA4, GA1, SE1
Early 8	FALSE	18	70	582	BR1, CA1, CA2, CA3, CA4, FA2, GA1
Early 9	FALSE	18	90	102	BR1, BR2, CA1, CA2, CA3, FA2, GA1
Early 10	TRUE	20	246	1015	BR1, BR2, CA1, CA2, CA3, CA4, SE1
Early 11	FALSE	20	246	1108	BR1, BR2, CA1, CA2, CA3, CA4, GA1, SE1
Early 12	TRUE	20	179	35	BR1, BR2, CA1, CA2, CA3, CA4, GA1
Early 13	FALSE	20	100	432	BR1, BR2, CA1, CA2, CA3, GA1
Early 14	FALSE	20	56	56	BR1, BR2, CA1, FA2, GA1
Early 15	FALSE	20	187	1224	BR1, BR2, CA1, CA2, CA3, CA4, FA2, GA1
Middle 1	TRUE	16	144	993	BR1, BR2, CA1, CA2, CA4, SE1
Middle 2	TRUE	16	246	1231	BR2, CA1, CA2, CA3, CA4, GA1, SE1
Middle 3	FALSE	16	237	1212	BR1, BR2, CA1, CA2, CA3, CA4, FA2, GA1, SE1
Middle 4	TRUE	18	97	168	BR1, CA1, CA2, CA3, CA4, GA1
Middle 5	TRUE	18	151	197	BR1, BR2, CA1, CA2, CA3, FA2, GA1
Middle 6	TRUE	18	162	154	BR1, BR2, CA1, CA2, CA3, CA4, GA1, SE1
Middle 7	FALSE	18	141	476	BR1, BR2, CA1, CA2, CA3, CA4, GA1, SE1
Middle 8	FALSE	18	62	224	BR1, CA1, CA2, CA3, CA4, GA1
Middle 9	FALSE	18	94	201	BR1, BR2, CA1, CA2, CA3, CA4, FA2, GA1
Middle 10	TRUE	20	198	714	BR1, BR2, CA1, CA2, CA4, GA1
Middle 11	TRUE	20	176	1048	BR1, CA1, CA2, CA3, CA4, FA2, GA1
Middle 12	TRUE	20	63	241	BR1, BR2, CA1, CA2, CA3, FA2, GA1, SE1
Middle 13	FALSE	20	219	350	BR1, CA1, CA2, CA3, CA4, GA1, SE1
Middle 14	FALSE	20	63	817	BR1, BR2, CA1, CA2, CA3, CA4, FA2, GA1
Middle 15	FALSE	20	95	133	BR1, BR2, CA1, CA2, CA3, GA1, SE1
Late 1	FALSE	16	235	542	BR1, CA1, CA2, CA3, CA4, FA2, GA1
Late 2	FALSE	16	246	1077	BR1, BR2, CA1, CA2, CA3, CA4, GA1
Late 3	FALSE	16	127	439	BR1, BR2, CA1, CA2, CA3, CA4, FA2, GA1, SE1
Late 4	TRUE	18	96	34	BR1, CA1, CA4, GA1, SE1
Late 5	TRUE	18	176	747	BR2, CA1, CA2, CA3, CA4, FA2, GA1
Late 6	TRUE	18	163	159	BR1, BR2, CA1, CA2, CA4, GA1, SE1
Late 7	FALSE	18	126	738	BR1, BR2, CA1, CA2, CA3, GA1, SE1
Late 8	TRUE	18	110	195	BR1, BR2, CA1, CA2, CA3, GA1, SE1
Late 9	FALSE	18	102	402	BR1, BR2, CA1, CA2, CA3, FA2, GA1
Late 10	TRUE	20	233	694	BR1, BR2, CA1, CA2, CA3, CA4, GA1, SE1
Late 11	TRUE	20	52	366	BR1, BR2, CA1, CA2, CA3, FA2, GA1
Late 12	FALSE	20	244	1009	BR1, BR2, CA1, CA2, CA3, CA4, SE1
Late 13	FALSE	20	169	134	BR2, CA1, CA2, CA3, CA4, FA2, GA1, SE1
Late 14	FALSE	20	116	993	BR1, CA1, CA2, CA3, CA4, FA2, GA1
Late 15	FALSE	20	51	41	BR1, BR2, CA1, CA3, FA2, GA1

4.2 Parameter Tuning

Since our ALNS depends on many different parameters that are not independent of each other, parameter tuning is essential to the performance of our heuristic. We use the hyperparameter optimization library SMAC3 [LEF+22] to determine good values for our parameters. We decided upon this library because it is effective at finding good hyperparameters in comparably few evaluations. This is crucial to our approach since a single run has to run for multiple hours before it becomes clear if the parameters are effective or not. Specifically, the parameters concerning the change of penalties described in Section 3.5 only become relevant once we get close to local optima. For this reason, we decided on a runtime of 3 hours for each set of parameters. However, when trying to optimize without any further adjustments the results seemed to be almost random. We identified that the issue was that since there is a lot of randomnesses involved in our approach some run that does not necessarily have great hyperparameters essentially gets lucky and the parameters are wrongfully identified as better than some more effective parameters that did not get as lucky. To solve this problem we decided to evaluate each set of parameters over three runs and use the average objective value for tuning.

In our initial tests, we also found that some parameters work much better on instances involving 16 teams than those involving 20 teams. Therefore, we decided to optimize for each schedule size separately. It is also noteworthy that we only used three instances (with different features and only selecting from the early instances of the competition) per schedule size for tuning. This prevents overfitting on the competition instances and gives us an objective evaluation. The results of the parameter tuning can be found in Tables 4.2- 4.4. Most of the parameters have been discussed in previous sections but we want to once again give a quick overview:

- **tabu length:** Indicates how many iterations (one iteration being one explored neighborhood) after a specific neighborhood is used we can not use it again. So if we use the Teams neighborhood with teams 1, 2, and 3 being destroyed we can not select the same 3 teams again for tabu length iterations.
- **max reconstruction time:** The time target can not get bigger than this parameter effectively limiting the maximum neighborhood size we explore for each neighborhood type.
- **min reconstruction time:** The time target can not get smaller than this parameter. This prevents neighborhoods from getting too small to the point where they are ineffective. Note that this also is our initial time target.
- **iter. before penalty changes:** Indicates the number of iterations before we change the penalty of a constraint, making the solution worse.
- **iter. before reset of penalty change:** Indicates the number of iterations before we restore the original penalty of a constraint after making the solution worse.

- **max iter. before resetting to best-known:** Indicates the maximum number of iterations we explore a worse schedule before resetting to the best-known solution.
- **iter. without change before increase:** After this amount of iterations without improvement, we will increase the time target.
- **time increase factor:** Whenever we increase the time target we calculate the new time target by multiplying the old time target with this factor.
- **time decrease factor:** Every time we find an improvement to the schedule we decrease the time target by multiplying with this factor.
- **exploration rate:** Indicates the exploration rate of the UCB multi-armed bandit method. Note that in literature [SB18, ACF02] you mostly find values between 1 and 10, however, the choice of the constant depends on the average reward size you expect. Since we have rewards in the magnitude between the 10s and the 100s (because of the exponential factor discussed in Section 3.4) we expect the exploration rate to also be higher than in the literature where rewards are often normalized or generally lower.
- **use ... neighborhood:** Indicates whether to use the neighborhood type or not.

As we can see in Tables 4.2- 4.4 the results of the parameter tuning do indeed indicate that different schedule sizes have different optimal parameters using our ALNS. For example, we see a trend that as schedule size increases the minimum time target becomes bigger, the increase factor rises, and the iterations before we perform increases and reset the schedule get larger. Other parameters, like exploration rate, tabu length, iterations before the reset of a penalty change, and the time decrease factor do not change significantly enough to say for sure that their values depend on the schedule size. Another interesting fact to observe is what neighborhoods are used for each neighborhood size. There are five neighborhoods that get used on every size of schedule namely Days, Days Phased, Teams, Teams Phased, and Combi. Our new neighborhoods' Grouping Teams and Grouping Days are only used in schedules of sizes 16 and 18 our suspicion as to why it is not used for neighborhoods of size 20 is, that they become fairly slow on this schedule size, and mostly only group very few teams or days together. Finally, the neighborhood Team Pairs (which was used by the winners [LFMSP21] of the ITC2021) as well as Teams Home Away and Days Home Away were deemed ineffective by our tuning. While it is not a huge surprise that the Home Away swap neighborhoods are not high-performing since many of the other neighborhoods implicitly also allow swaps, it is rather surprising to us that the established Team Pairs neighborhood has such poor performance. For the schedules of sizes 16 and 18, we attribute the lack of performance of this neighborhood to the Grouping Teams neighborhood that due to its nature essentially performs multiple Team Pairs neighborhoods at the same time across the schedule and could therefore make the Team Pairs neighborhood obsolete. However, the schedules of size 20 do neither use the Team Pairs nor the Grouping Teams neighborhood so it might be the case that the

Team Pairs neighborhood similar to the Grouping neighborhoods is not efficient enough for schedules of this size.

Table 4.2: Parameter Tuning results for schedules with 16 teams

parameter name	range	default value	post tuning value
tabu length	10 - 1000	500	175
max reconstruction time target	30 - 90	60	71
min reconstruction time target	3 - 30	10	8
iter. before penalty changes	35 - 200	105	70
iter. before reset of penalty change	10 - 50	30	30
max iter. before reset to best-known	51 - 125	75	58
iter. without change before increase	10 - 75	35	30
time increase factor	1.0 - 2.0	1.15	1.16
time decrease factor	0.5 - 1.0	0.8	0.82
exploration rate	1.0 - 300.0	100	26
use Teams neighborhood	T / F	T	T
use Team Pairs neighborhood	T / F	T	F
use Teams Home Away neighborhood	T / F	T	F
use Days neighborhood	T / F	T	T
use Days Home Away neighborhood	T / F	T	F
use Combi neighborhood	T / F	T	T
use Grouping Teams neighborhood	T / F	T	T
use Grouping Days neighborhood	T / F	T	T
use Teams Phased neighborhood	T / F	T	T
use Days Phased neighborhood	T / F	T	T

Table 4.3: Parameter Tuning results for schedules with 18 teams

parameter name	range	default value	post tuning value
tabu length	10 - 1000	500	248
max reconstruction time target	30 - 90	60	84
min reconstruction time target	3 - 30	10	12
iter. before penalty changes	35 - 200	105	107
iter. before reset of penalty change	10 - 50	30	37
max iter. before reset to best-known	51 - 125	75	88
iter. without change before increase	10 - 75	35	40
time increase factor	1.0 - 2.0	1.15	1.17
time decrease factor	0.5 - 1.0	0.8	0.88
exploration rate	1.0 - 300.0	100	32
use Teams neighborhood	T / F	T	T
use Team Pairs neighborhood	T / F	T	F
use Teams Home Away neighborhood	T / F	T	F
use Days neighborhood	T / F	T	T
use Days Home Away neighborhood	T / F	T	F
use Combi neighborhood	T / F	T	T
use Grouping Teams neighborhood	T / F	T	T
use Grouping Days neighborhood	T / F	T	T
use Teams Phased neighborhood	T / F	T	T
use Days Phased neighborhood	T / F	T	T

Table 4.4: Parameter Tuning results for schedules with 20 teams

parameter name	range	default value	post tuning value
tabu length	10 - 1000	500	221
max reconstruction time target	30 - 90	60	88
min reconstruction time target	3 - 30	10	16
iter. before penalty changes	35 - 200	105	117
iter. before reset of penalty change	10 - 50	30	30
max iter. before reset to best-known	51 - 125	75	89
iter. without change before increase	10 - 75	35	57
time increase factor	1.0 - 2.0	1.15	1.22
time decrease factor	0.5 - 1.0	0.8	0.92
exploration rate	1.0 - 300.0	100	34
use Teams neighborhood	T / F	T	T
use Team Pairs neighborhood	T / F	T	F
use Teams Home Away neighborhood	T / F	T	F
use Days neighborhood	T / F	T	T
use Days Home Away neighborhood	T / F	T	F
use Combi neighborhood	T / F	T	T
use Grouping Teams neighborhood	T / F	T	F
use Grouping Days neighborhood	T / F	T	F
use Teams Phased neighborhood	T / F	T	T
use Days Phased neighborhood	T / F	T	T

Finally, we also compare the performance of the ALNS pre- and post-tuning: Table 4.5 shows the difference in objective value between using the default parameters and the post-tuning parameters. Instances 1-3 have 16 teams, 4-9 have 18 teams and 10-15 have 20 teams in their schedule. We see that the results did improve significantly in almost all instances with the biggest difference being Early 14 where the default parameters had an objective value almost 4 times higher. There seems to be a trend that the bigger instances are affected more by the tuning of the parameters. This can have three reasons: Either the parameters we found for the small instances are not that good, the small instances are easier so even "bad" parameters produce good results or the default parameters are closer to the optimal parameters of small instances than big instances. Given that our very early experiments from which the default parameters stem were mostly performed on instance Early 1 and comparing the default values to the tuned ones we arrive at the conclusion that the third option is the most likely. Considering the explained difficulty in parameter tuning for this problem described above we are very happy with the results, but it is likely that with more time spent on automatic tuning even better parameters are achievable.

Table 4.5: Comparison of objective value between tuned and default parameters. The % difference indicates the difference in the average objective value of each instance. We use 10 runs with a time limit of 3 hours each. Infeasible results are excluded from the calculations.

Instance	tuned:	obj. val.	std. dev.	default:	obj. val.	std. dev.	% dif.
Early 1		543	86		544	35	100,2
Early 2		358	18		394	33	110,1
Early 3		1281	61		1260	53	98,4
Early 4		1319	123		1684	113	127,7
Early 5		INF	INF		INF	INF	-
Early 6		4499	229		4438	202	98,6
Early 7		7470	217		8172	841	109,4
Early 8		1549	83		1769	118	114,2
Early 9		723	77		800	52	110,7
Early 10		INF	INF		INF	INF	-
Early 11		7236	801		8401	691	116,1
Early 12		1024	68		1113	93	108,7
Early 13		402	29		502	67	124,9
Early 14		297	67		1174	105	395,3
Early 15		5099	104		6076	175	119,2

4.3 Comparison of Strategies for Creating Feasible Solutions

The instances presented in the ITC2021 [VG23b] are very challenging. In fact, multiple teams [LFMSP21, POW21] participating in the competition have reported that modern ILP solvers are unable to come up with feasible solutions in a reasonable amount of time when not splitting the problem into smaller subproblems. However, there are many different approaches to splitting the problem. In this section, we will evaluate existing approaches and compare them to the results of our new approach presented in Section 3.3.3. We also will look into some variants of our new approach and discuss the advantages of each.

As mentioned in Section 3.3.3 previous approaches tackling the DRRST looked at all hard constraints at the same time. This has the advantage of always having a global overview of the schedule and enables the solver to only accept strict improvements (meaning schedules that have strictly less hard constraint violations). But, what we identified is that this also can lead to situations where most hard constraints are fulfilled but the remaining few are very hard to solve because at that point the schedule has become much more rigid. This leads to a long time to feasibility where the most time is spent eliminating the last few hard constraint violations. Some teams have also identified this problem and come up with solutions. For example, Rosati et al. [RPGS22] have come up with the idea to analyze the "difficulty" of each constraint type and they changed the penalties accordingly. Lamas-Fernandez et al. [LFMSP21] have chosen a different approach where they try to find a feasible solution and once they find no more improvements they increase the coefficients of the d_c variables in the ILP and restart from the beginning. Our new multi-stage approach takes the idea of prioritizing certain constraints over others a step further by fulfilling one constraint type at a time. Tables 4.6 and 4.7 compare the time and objective value of our new multi-stage approach to both a weighted and unweighted version of the single-stage approach using the Early instances of the competition. The weights for the weighted approach stem from the paper by Rosati et al. [RPGS22]. Note that we also used GA1 reduction and the additional soft constraints that were described in Section 3.3.3 in all experiments except when we mention otherwise. The results indicate that the multi-stage approach outperforms the unweighted approach on almost all instances in regard to time to feasibility. When comparing it to the weighted approach the difference is less significant however it is still more than 10% faster on 6 out of the 15 instances while the single-stage approach only significantly outperforms the multi-stage approach on instance Early 1. If we compare objective values we see that there are no significant differences on the instances that reached feasibility on all 10 trials, however, if the solutions do not become feasible the single-stage approach generally has fewer hard constraint violations. This makes sense because if the multi-stage approach is still working on e.g. CA3 constraints when the experiment reaches its time limit it will not have considered BR1 constraints at all leading to a lot of extra violations even if they would be easy to fix. The single-stage approach on the other hand will look at all hard constraints solving the easy ones right away therefore resulting in a better objective value.

Table 4.6: Comparison of time (s) to reach feasibility for multi-stage approach, all unweighted hard constraints at the same time (single stage unweighted) and all unweighted hard constraints at the same time (single-stage weighted) using a 30-minute time limit. The % difference indicates the difference to the multi-stage approach. Avg. of 10 runs.

Instance	multi-stage	single-stage unweighted	% dif.	single-stage weighted	% dif.
Early 1	152	166	9,2	107	-29,6
Early 2	164	247	50,6	151	-7,9
Early 3	9,6	15,3	59,4	14,2	47,9
Early 4	1800	1800	0	1800	0
Early 5	1800	1800	0	1800	0
Early 6	1550	1800	16,1	1760	13,5
Early 7	1762	1799	2,1	1710	-3
Early 8	5,6	7,8	39,3	6	7,1
Early 9	5,8	6,9	19	6,6	13,8
Early 10	1800	1800	0	1800	0
Early 11	1800	1800	0	1728	-4
Early 12	595	1800	202,5	866	45,5
Early 13	112	158	41,1	144	28,6
Early 14	10,6	10	-5,7	18,6	75,5
Early 15	297	490	65	308	3,7

Table 4.7: Comparison of weighted objective value after reaching feasibility or 30-minute time limit. Comparing the multi-stage approach to all unweighted hard constraints at the same time (single stage unweighted) and to all weighted hard constraints (single stage weighted). The % difference indicates the difference to the multi-stage approach. Weight of hard constraints = 10000. Avg. of 10 runs.

Instance	multi-stage	single-stage unweighted	% dif.	single-stage weighted	% dif.
Early 1	2492	2259	-9,3	2326	-6,7
Early 2	820	833	1,6	816	-0,5
Early 3	4245	4363	2,8	4495	5,9
Early 4	149011	132189	-11,3	147053	-1,3
Early 5	510387	339322	-33,5	293249	-42,5
Early 6	16922	46880	177	7050	-58,3
Early 7	154842	36537	-76,4	31761	-79,5
Early 8	4400	4806	9,2	4596	4,5
Early 9	4395	4531	3,1	4235	-3,6
Early 10	449302	384224	-14,5	196180	-56,3
Early 11	166122	117245	-29,4	101197	-39,1
Early 12	1938	20849	975,8	1908	-1,5
Early 13	1435	1425	-0,7	1431	-0,3
Early 14	4159	5379	29,3	5324	28
Early 15	6857	6948	1,3	6872	0,2

As described in Section 3.3.3 we also add additional soft constraints that aim to keep the schedule more flexible by incentivizing the solver to not just fulfill a constraint but also stay as far from the maximum as possible. Tables 4.8 and 4.9 show the average difference in runtime and objective value for the early instances of the competition with the added soft constraints vs. without the added soft constraints, clearly indicating that the soft constraints reduce both the time to feasibility and resulting objective value of the feasible solution for a majority of the Early instances from the competition. We suspect that the decrease in the runtime does indeed stem from the heightened flexibility of the schedule while the decrease in objective value likely comes from a combination of the former and the overall reduced amount of breaks which helps to fulfill more soft constraints once reaching feasibility.

Table 4.8: Comparison of time (s) to reach feasibility with vs. without using additional soft constraints (SC). 30-minute time limit or stop on reaching feasibility. 10 runs avg..

Instance	with SC	without SC	% dif.
Early 1	152	194	27,6
Early 2	164	513	212,8
Early 3	9,6	9,5	-1
Early 4	1800	1800	0
Early 5	1800	1800	0
Early 6	1550	1570	1,3
Early 7	1762	1800	2,2
Early 8	5,6	4,8	-14,3
Early 9	5,8	7,3	25,9
Early 10	1800	1800	0
Early 11	1800	1800	0
Early 12	595	990	66,4
Early 13	112	206	83,9
Early 14	10,6	12,8	20,8
Early 15	297	339	14,1

Table 4.9: Comparison of objective value with vs. without using additional soft constraints (SC). 30-minute time limit or stopping once reaching feasibility. Weight of hard constraints = 10000. Avg. from 10 runs.

Instance	with SC	without SC	% dif.
Early 1	2492	2234	-10,4
Early 2	820	1779	117
Early 3	4245	4585	8
Early 4	149011	499073	234,9
Early 5	510387	836494	63,9
Early 6	16922	30749	81,7
Early 7	154842	264626	70,9
Early 8	4400	4899	11,3
Early 9	4395	4801	9,2
Early 10	449302	570267	26,9
Early 11	166122	315964	90,2
Early 12	1938	17994	828,5
Early 13	1435	1462	1,9
Early 14	4159	5994	44,1
Early 15	6857	6857	0

We also looked into the effects of our strategy for reducing violated GA1 constraints. Tables 4.10 and 4.11 respectively show that overall the GA1 reduction leads to a slight improvement in runtime for 7 of the 15 instances and on 4 of the instances the runtime was slightly better without the reduction. When ignoring differences of less than 10% that could easily stem from the high variances in the tests this changes to 5 and 1 respectively with the biggest relative differences in instance Early 8 (160% more runtime without GA1 reduction). This indicates that using the GA1 reduction is beneficial to reducing the runtime, especially since the computational overhead is minimal. Regarding the objective value of the schedules after the 30-minute time limit (or after reaching feasibility) most differences can be attributed to having fewer hard constraint violations upon reaching the time limit which directly correlates with having a better runtime. In those instances that reached feasibility every time we only found significant differences in instances Early 1 and Early 14 however, with instance Early 1 favoring the approach without GA1 reduction and instance Early 14 having a better objective value using the GA1 reduction. However, looking at all other experiments described in this Section it appears that the objective values of Early 1 and Early 14 seem to be statistical outliers with Early 1 having the worst average objective value of all experiments and Early 14 having the best average objective value (by a big margin). We therefore do not feel confident in reporting any significant difference in objective value after reaching feasibility, which is expected since we don't see how reducing the initial amount of hard GA1 constraints could influence the objective value that is only affected by soft constraints after reaching feasibility.

Table 4.10: Comparison of time (s) to reach feasibility with vs. without using GA1 reduction. 30-minute time limit or stopping once reaching feasibility. Avg. from 10 runs.

instance	with GA1 reduction	without GA1 reduction	% dif.
Early 1	152	139	-8,6
Early 2	164	197	20,1
Early 3	9,6	15,2	58,3
Early 4	1800	1800	0
Early 5	1800	1800	0
Early 6	1550	1515	-2,3
Early 7	1762	1800	2,2
Early 8	5,6	14,6	160,7
Early 9	5,8	7,3	25,9
Early 10	1800	1800	0
Early 11	1800	1800	0
Early 12	595	675	13,4
Early 13	112	116	3,6
Early 14	10,6	10,1	-4,7
Early 15	297	247	-16,8

Table 4.11: Comparison of objective value with vs. without using GA1 reduction. 30-minute time limit or stopping once reaching feasibility. Weight of hard constraints = 10000. Avg. from 10 runs.

instance	with GA1 reduction	without GA1 reduction	% dif.
Early 1	2492	2190	-12,1
Early 2	820	781	-4,8
Early 3	4245	4216	-0,7
Early 4	149011	192097	28,9
Early 5	510387	535198	4,9
Early 6	16922	14910	-11,9
Early 7	154842	217825	40,7
Early 8	4400	4761	8,2
Early 9	4395	4302	-2,1
Early 10	449302	478175	6,4
Early 11	166122	199134	19,9
Early 12	1938	1935	-0,2
Early 13	1435	1555	8,4
Early 14	4159	5343	28,5
Early 15	6857	6834	-0,3

Finally, we also explored different orders of constraint types in our multi-stage approach. The first order, which was used in all previous experiments (except the single-stage ones) is ["BR2", "GA1", "CA2", "CA4", "CA1", "CA3", "BR1"]. This order results from ranking the constraint types according to the analysis by Rosati et al. [RPGS22] with the exception that we moved the BR2 constraints to the front because they were fulfilled as a result of our initially generated schedule. This order represents our default order. As a second experiment we use the order ["GA1", "CA2", "CA4", "CA1", "BR2", "CA3", "BR1"] where we use the same order but keep the BR2 constraints at the position that would result out of the previously mentioned analysis. Finally, we also experimented with the order ["BR2", "GA1", "CA2", "CA1", "CA4", "BR1", "CA3"] that ranks the constraint not according to an empirically evaluated difficulty but instead tries to rank the constraints by how much of the schedule they affect (except BR2 for the same reason as the first order). The idea behind this final ranking is that it becomes very hard to fulfill constraints that concern a very specific part of the schedule once a lot of other constraints affect that part and make it rigid, whereas if a constraint affects a bigger part of the schedule we suspected that there might be more opportunities to fulfill that constraint even if a lot of the schedule is already more or less fixed. We therefore label this new approach "most specific first". Tables 4.12 and 4.13 show that the order of constraints has a significant impact on the runtime and objective value of the resulting schedules. However, it is interesting to observe that it depends on the instance which order is better. For example, if we look at instances Early 2 and 3 we find that the default order is significantly faster in finding feasible solutions than the other two orders. When looking at instances Early 11 and Early 7 we see that the multi-stage approach using the default order (almost) always timed out without finding a feasible solution while the other two orders found feasible solutions in multiple of the 10 runs, with the order default BR2 fifth being the most successful finding feasible solutions 7 and 5 times out of 10 respectively. When looking at the comparisons of objective values we find that for the instances that are feasible on every run, there is no significant difference between the orders. However, looking at those instances that either stay infeasible on every run or on a portion of the runs we observe that there are significant differences, with the default with BR2 fifth having the best results for those instances most of the time. The results of those experiments suggest to us that different instances favor different orders of constraint types. However, since finding a feasible solution for as many instances as possible is the most important part we use the order default BR2 fifth together with the additional soft constraints and GA1 reduction, which we found to be beneficial, for our experiments in Section 4.5.

Table 4.12: Comparison of time (s) to reach feasibility for multi-stage approach comparing different orders of constraint types namely default, default with BR2 on fifth instead of first position (both resulting out of the research by Rosati et al. [RPGS22] as well as most specific first. Using a 30-minute time limit. The % difference indicates the difference to the default order. Avg. from 10 runs.

Instance	default	default BR2 fifth	% dif.	most specific first	% dif.
Early 1	152	108	-28,9	76	-50
Early 2	164	260	58,5	199	21,3
Early 3	9,6	17,7	84,4	25	160,4
Early 4	1800	1800	0	1800	0
Early 5	1800	1800	0	1800	0
Early 6	1550	1445	-6,8	1642	5,9
Early 7	1762	1408	-20,1	1626	-7,7
Early 8	5,6	8,5	51,8	11,2	100
Early 9	5,8	6,3	8,6	11	89,7
Early 10	1800	1800	0	1800	0
Early 11	1800	1503	-16,5	1712	-4,9
Early 12	595	666	11,9	693	16,5
Early 13	112	171	52,7	115	2,7
Early 14	10,6	10,5	-0,9	11,7	10,4
Early 15	297	218	-26,6	268	-9,8

Table 4.13: Comparison of weighted objective value after reaching feasibility or 30-minute time limit. Comparing different orders using the multi-stage approach namely default, default with BR2 on fifth instead of first position (both resulting out of the research by Rosati et al. [RPGS22] as well as most specific first. The % difference indicates the difference to the default order. Weight of hard constraints = 10000. Avg. from 10 runs.

Instance	default	default BR2 fifth	% dif.	most specific first	% dif.
Early 1	2492	2454	-1,5	2320	-6,9
Early 2	820	816	-0,5	848	3,4
Early 3	4245	4248	0,1	4051	-4,6
Early 4	149011	90888	-39	187917	26,1
Early 5	510387	452211	-11,4	450302	-11,8
Early 6	16922	16911	-0,1	17799	5,2
Early 7	154842	25688	-83,4	61799	-60,1
Early 8	4400	4702	6,9	4642	5,5
Early 9	4395	4076	-7,3	4294	-2,3
Early 10	449302	499952	11,3	433157	-3,6
Early 11	166122	28909	-82,6	75448	-54,6
Early 12	1938	3984	105,6	1920	-0,9
Early 13	1435	1501	4,6	1483	3,3
Early 14	4159	5423	30,4	5404	29,9
Early 15	6857	6853	-0,1	6843	-0,2

4.4 Impact of Adaptivity

Since one of our main contributions is the use of adaptive techniques for a more efficient generation of close-to-optimal schedules, it is important to directly compare the multi-armed bandit selection to a baseline model that does not use adaptivity but instead selects neighborhoods at random. In Table 4.14 we evaluate the effects of adaptivity on the Early instances using ten runs with a three-hour time limit. The table shows that the approach using adaptivity has better results in every instance except for Early 4, which only got feasible twice in the case without adaptivity and four times with adaptivity. This, of course, makes the average more volatile to outliers. In the case of this specific instance, we believe that being feasible twice as often is more representative of the performance of the adaptivity than the worse average objective value. For the other instances, the differences in objective values look less impressive at first sight than they actually are. While a 5-10% difference may initially not look like too much one must keep in mind that each successive improvement becomes harder than the previous one. Also, the effects of adaptivity increase over time because for the first approximately 30 minutes almost every neighborhood has a very good chance of finding improvements since there is still a lot left to improve upon. So with longer runs like we use in Section 4.5 the differences would likely become much bigger. Nevertheless, the average improvement for a three-hour run is already 9,5% which clearly shows that using the UCB bandit formulation is beneficial.

Table 4.14: Comparison of objective value between using the UCB multi-armed bandit method for neighborhood type selection vs. choosing the neighborhood type at random. The % difference indicates the difference in the average objective value of each instance. We use 10 runs with a time limit of 3 hours each. Infeasible results are excluded from the calculations.

Instance	UCB:	obj. val.	std. dev.	no bandit:	obj. val.	std. dev.	% dif.
Early 1		543	40		615	72	13,3
Early 2		350	32		374	41	6,9
Early 3		1267	74		1289	49	1,7
Early 4		1464	61		1353	150	-7,6
Early 5		INF	INF		INF	INF	-
Early 6		4428	149		4627	89	4,5
Early 7		8223	777		8283	892	0,7
Early 8		1533	91		1698	69	10,8
Early 9		732	80		803	39	9,7
Early 10		INF	INF		INF	INF	-
Early 11		6771	432		6803	481	0,5
Early 12		971	69		1135	55	16,9
Early 13		379	28		409	37	7,9
Early 14		320	49		493	51	54,1
Early 15		5009	177		5227	153	4,4

Next, we look into the effects of using the global map structure M we described in Section 4.4. Table 4.15 shows that the results of using this structure are mixed. While some instances like Early 13 clearly benefit from it others like Early 14 perform better without it. If you take an average over all instances the results are approximately the same. This indicates that there are days and teams that when selected have a higher chance of yielding improvements. However, our structure does not reliably identify them, thus we have very mixed results based on whether we identified the right ones or not. If this was not the case we would expect less of a spread of results. This raises the question for future research: What better methods of identifying the right days and teams for each neighborhood do exist, that consistently outperform random choices?

Table 4.15: Comparison of objective value between using a global map structure for team and day selection vs. selecting them at random. The % difference indicates the difference in the average objective value of each instance. We use 10 runs with a time limit of 3 hours each. Infeasible results are excluded from the calculations.

Instance	with M:	obj. val.	std. dev.	no M:	obj. val.	std. dev.	% dif.
Early 1		543	40		525	39	-3,3
Early 2		350	32		351	28	0,3
Early 3		1267	74		1278	63	0,9
Early 4		1464	61		1546	336	5,6
Early 5		INF	INF		INF	INF	-
Early 6		4428	149		4513	218	1,9
Early 7		8223	777		7530	759	-8,4
Early 8		1533	91		1588	90	3,6
Early 9		732	80		724	64	-1,1
Early 10		INF	INF		INF	INF	-
Early 11		6771	432		6887	426	1,7
Early 12		971	69		999	103	2,9
Early 13		379	28		445	28	17,4
Early 14		320	49		287	77	-10,3
Early 15		5009	177		5031	138	0,4

Finally, we look into a comparison of using the UCB bandit method vs. the ϵ-greedy bandit method for selecting neighborhood types. The α parameter for the ϵ-greedy formulation was determined using automatic parameter tuning in SMAC3 with the same amount of runs as the UCB formulation received. The resulting α parameters are all between 0.65 and 0.75 depending on the instance size (with a trend of higher α on the bigger instances). Table 4.16 shows a comparison between using the UCB and the ϵ-greedy formulation. The results clearly indicate that most of the time the UCB method performs better on average. Interestingly, there are also instances where the ϵ-greedy formulation performs better, which possibly indicates that in those instances a shift of best neighborhood type occurs over time as described in Section 3.4. This claim is also supported by the fact that the two instances where this phenomenon occurs (Early 4 and Early 7) have almost equally good performance using the UCB method as when we use no bandit method at all.

Table 4.16: Comparison of objective value between using the UCB and the ϵ-greedy formulation of the multi-armed bandit problem. The % difference indicates the difference in the average objective value of each instance. We use 10 runs with a time limit of 3 hours each. Infeasible results are excluded from the calculations.

Instance	UCB: obj. val.	std. dev.	ϵ-greedy: obj. val.	std. dev.	% dif.
Early 1	543	40	610	69	12,3
Early 2	350	32	361	35	3,1
Early 3	1267	74	1307	47	3,2
Early 4	1464	61	1308	169	-10,7
Early 5	INF	INF	INF	INF	-
Early 6	4428	149	4642	224	4,8
Early 7	8223	777	7614	404	-7,4
Early 8	1533	91	1603	64	4,6
Early 9	732	80	808	54	10,4
Early 10	INF	INF	INF	INF	-
Early 11	6771	432	6768	426	0
Early 12	971	69	1067	91	9,9
Early 13	379	28	434	32	14,5
Early 14	320	49	414	61	29,4
Early 15	5009	177	5138	117	2,6

4.5 Evaluation on ITC2021 Instances

In this chapter, we will look at the performance of our approach. Specifically, in Section 4.5.1 we compare our solutions with the best-known solutions of each instance of the ITC2021 [VG23b] and make some general remarks about the overall performance. In Section 4.5.2 we compare our best results to the best results of four other teams to give a better impression of where we are ranking compared to the state-of-the-art. Here we will also look at how we would have ranked in the competition if we had participated back in 2021. Finally, we will look at some strengths and weaknesses that tell us in which instances the ALNS is working best.

4.5.1 Results

General Results

For our final evaluation, we use 10 runs with a time limit of 6 hours for instances numbered 1 through 9 (16 and 18 teams) and 5 runs with a time limit of 9 hours for instances numbered 10 through 15 (20 teams). Those time limits are very much on the low end for ILP-based approaches, but our resource limit did not allow us to go beyond this. The reason for the larger time target for instances with 20 teams is that after 6 hours improvements were still very frequent while after 9 hours they started to slow down. In general, almost all of the results listed in Table 4.17 can likely be improved by simply extending the time limit, but our goal was to implement a resource-efficient approach that competes with the runtimes of the simulated annealing approach by Rosati et al. [RPGS22]. Table 4.17 shows that for many instances we are relatively far away from the best-known solution but for 12 of them we have a less than 20% gap to the optimum. For one instance we found a new best-known solution. However, we will see in Section 4.5.2 that most of the time the best-known solution is not very representative of how most approaches perform, since it usually involves either very high runtimes (sometimes multiple days) or an excessive amount of trials (sometimes more than 100) or a combination of both. We can see in the column of our theoretical placement in the ITC2021 (the placement we would get if we had participated) that we usually rank between third and sixth (out of 14 including us) in most instances. If we sum up the points we would get from those ranking according to the competition rules we would rank fourth overall. It is clear that it is hard to reach top results with only 6 to 9 hours of runtime and 5 to 10 trials. This makes the fact that we did find a new best-known solution for instance Middle 3 much more significant.

To get a better understanding of how longer runtimes might influence our results, we decided to run the algorithm for 24 hours on each of our own best-known solutions. Table 4.18 shows that the solutions do indeed improve significantly. With this single run on each instance, we have found two more best-known solutions (Middle 10 and Late 2). We also generated one additional feasible solution (Middle 1) and improved the objective values of 35 out of 39 feasible but not proven to be optimal instances. The extent of the improvement varies across the instances with some improvements only being very minor while on some other instances we improved the objective value by almost 25 %. It is notable, that some instances still showed improvements towards the end of the 24-hour runtime but many of the instances with 16 and 18 teams showed no further improvements in the last 12 hours of the optimization. Generally, it seems to be very instance-dependent how long it takes before no further solutions are found, so ideally if there are enough computational resources the algorithm should only terminate if no further improvements can be found after a certain amount of time.

Another promising aspect of this approach is that we got feasible solutions on 41 out of 45 competition instances and only 2 approaches [RPGS22, LFMSP21] managed to get more than that. This shows that our strategy for coming up with feasible solutions is working very well.

Table 4.17: Evaluation of our ALNS on instances of the ITC2021 [VG23b]. Infeasible results are excluded from the calculation of average objective value and standard deviation. The feasible column indicates how many of the runs reached feasibility. The best-known solution includes solutions from both the ITC2021 as well as all post-competition solutions.

Instance	best obj. val.	avg. obj. val.	std. dev.	feasible	ITC pos.	best-known:	% dif.
Early 1	386	527	58	1.0	3	362	6,6
Early 2	247	321	32	1.0	3	160	54,4
Early 3	1105	1222	66	1.0	5	1012	9,2
Early 4	889	1590	298	0.3	5	512	73,6
Early 5	inf	inf	inf	0.0	inf	3127	-
Early 6	4058	4377	98	0.6	4	3352	21,1
Early 7	6342	7392	98	1.0	3	4763	33,2
Early 8	1371	1524	78	1.0	5	1051	30,4
Early 9	452	598	91	1.0	7	56	707,1
Early 10	inf	inf	inf	0.0	inf	3400	-
Early 11	5644	6302	645	1.0	6	4436	27,2
Early 12	765	826	40	1.0	4	320	139,1
Early 13	332	373	37	1.0	5	121	174,4
Early 14	65	104	27	1.0	6	4	1525
Early 15	4284	4517	167	1.0	5	3110	37,7
Middle 1	inf	inf	inf	0.0	inf	5177	-
Middle 2	inf	inf	inf	0.0	inf	7381	-
Middle 3	9426*	10943	568	0.7	1	9542	-1,2
Middle 4	9	13	3	1.0	8	7	28,6
Middle 5	472	524	37	1.0	3	295	60
Middle 6	1615	1844	133	1.0	3	1125	43,6
Middle 7	2742	3076	129	1.0	5	1784	53,7
Middle 8	180	239	36	1.0	4	129	39,5
Middle 9	1085	1185	59	1.0	8	440	146,6
Middle 10	1367	1487	93	1.0	3	1250	9,4
Middle 11	2923	3051	95	1.0	5	2511	16,4
Middle 12	954	1086	98	1.0	3	599	59,3
Middle 13	744	821	75	1.0	7	253	194,1
Middle 14	1418	1556	80	1.0	3	1140	24,4
Middle 15	1266	1337	45	1.0	6	495	155,8
Late 1	2113	2339	130	1.0	4	1969	7,3
Late 2	5860	5890	30	0.2	6	5400	8,5
Late 3	2617	2882	156	1.0	4	2369	10,5
Late 4	0*	0	0	1.0	1	0	0
Late 5	inf	inf		0.0	inf	1939	-
Late 6	1216	1270	51	1.0	6	923	31,7
Late 7	2228	2627	283	1.0	4	1558	43
Late 8	1077	1138	44	1.0	4	934	15,3
Late 9	1059	1195	81	1.0	5	527	100,9
Late 10	2341	2341	0	0.2	3	1988	17,8
Late 11	236	286	37	1.0	3	207	14
Late 12	5004	5140	96	1.0	6	3689	35,6
Late 13	2779	2901	89	1.0	6	1820	52,7
Late 14	1490	1595	73	1.0	5	1202	24
Late 15	140	199	26	1.0	10	0	infinity

Table 4.18: Results of continuing our optimization on our best-known solutions for an additional 24h. Only a single run was performed.

Instance	before	after	% dif.
Early 1	386	372	-3,6
Early 2	247	247	0
Early 3	1105	1046	-5,3
Early 4	889	784	-11,8
Early 5	inf	inf	-
Early 6	4058	3855	-5
Early 7	6342	5880	-7,3
Early 8	1371	1277	-6,9
Early 9	452	367	-18,8
Early 10	inf	inf	-
Early 11	5644	5058	-10,4
Early 12	765	710	-7,2
Early 13	332	252	-24,1
Early 14	65	63	-3,1
Early 15	4284	4184	-2,3
Middle 1	inf	6062	infinity
Middle 2	inf	inf	-
Middle 3	9426*	9426*	0
Middle 4	9	9	0
Middle 5	472	469	-0,6
Middle 6	1615	1615	0
Middle 7	2742	2634	-3,9
Middle 8	180	175	-2,8
Middle 9	1085	1045	-3,7
Middle 10	1367	1228*	-10,2
Middle 11	2923	2813	-3,8
Middle 12	954	914	-4,2
Middle 13	744	571	-23,3
Middle 14	1418	1384	-2,4
Middle 15	1266	1202	-5,1
Late 1	2113	2034	-3,7
Late 2	5860	5384*	-8,1
Late 3	2617	2583	-1,3
Late 4	0	0	0
Late 5	inf	inf	-
Late 6	1216	1065	-12,4
Late 7	2228	1975	-11,4
Late 8	1077	1006	-6,6
Late 9	1059	985	-7
Late 10	2341	2090	-10,7
Late 11	236	226	-4,2
Late 12	5004	4429	-11,5
Late 13	2779	2296	-17,4
Late 14	1490	1251	-16
Late 15	140	120	-14,3

Analysis of Neighborhood Usage

In Table 4.19 it is listed how many times each neighborhood type was used on average during our analysis which strongly correlates with the average reward gained through the neighborhood. The neighborhoods Team Paris, Teams HA, and Days HA were excluded from the table since our parameter tuning determined it is better to not spend any time using them. The first thing we observe when looking at the table is that we excluded the Grouping Teams and Grouping Days neighborhoods from the large instances because of our results from automatic parameter tuning and we excluded the Teams Phased and Days Phased neighborhoods from non-phased instances. We can see that it is indeed highly instance-dependent which neighborhoods are the most successful. However, there are some patterns we can observe. First of all the Days Phased neighborhood is almost always the most successful on phased instances. Next, we see that the Teams Phased neighborhood is more successful than the ordinary Teams neighborhood on only 9 out of 22 phased instances meaning that it is not strictly better to look at the two halves of the tournament separately when working with Team based neighborhoods. This is unexpected because when the time target is the same for both neighborhoods the Teams Phased neighborhood usually destroys two to three times the amount of teams in a single iteration. However, the Teams Phased neighborhood does not allow home-away swaps along with the switches of matchups which likely contributes to the attribute of being able to handle more teams at a time without an increase in average gained rewards.

Next, if we take a look at the Grouping Teams and Grouping Days neighborhoods we see that in almost all instances it is more successful to group teams rather than days. The cause for this could again be the heightened ability to change home-away patterns.

Finally, if we look at the Days, Teams, and Combi neighborhoods we see that they have the greatest variance of success across instances. The Days neighborhood is usually very successful on non-phased instances when the Days Phased neighborhood is not available. The Teams neighborhood has a very hit-or-miss performance where it shines on some instances like Late 10 and Middle 13 but is one of the least selected on others. We were, however, not able to find what caused the performance of the Team neighborhood to spike on some instances. Finally, the Combi neighborhood had good performances across almost all instances, especially the non-phased ones where it did not have to compete with the Days Phased Neighborhood. But there are also some non-phased instances where the Combi neighborhood was completely outclassed by other neighborhoods like Late 12.

Overall, this analysis of neighborhood usages shows us that adaptive neighborhood selection is indeed very important for the DRRST problem as different instances show very different patterns that are hard to pick up from just looking at the constraints.

Table 4.19: Neighborhood usages across instances

Instance	Days	Teams	Combi	Grouping Teams	Grouping Days	Teams Phased	Days Phased
Early 1	67	65	62	59	62	145	458
Early 2	109	69	169	74	79	148	341
Early 3	80	52	108	244	59	121	312
Early 4	68	52	92	70	44	56	165
Early 5	92	86	71	118	57	67	191
Early 6	35	42	43	87	44	68	403
Early 7	443	110	239	102	51	0	0
Early 8	322	69	196	199	38	0	0
Early 9	828	132	241	270	73	0	0
Early 10	90	141	274	0	0	79	383
Early 11	192	108	1003	0	0	0	0
Early 12	221	127	278	0	0	117	575
Early 13	927	234	540	0	0	0	0
Early 14	301	183	856	0	0	0	0
Early 15	734	173	514	0	0	0	0
Middle 1	100	125	114	113	78	68	109
Middle 2	80	105	130	119	72	72	93
Middle 3	316	155	187	114	64	0	0
Middle 4	173	137	194	138	126	160	277
Middle 5	188	112	174	284	193	84	852
Middle 6	337	40	226	65	44	74	380
Middle 7	262	588	615	93	40	0	0
Middle 8	501	170	342	123	117	0	0
Middle 9	508	217	570	116	85	0	0
Middle 10	210	147	164	0	0	163	1125
Middle 11	394	118	293	0	0	89	616
Middle 12	98	108	389	0	0	91	1119
Middle 13	395	1001	593	0	0	0	0
Middle 14	1025	148	431	0	0	0	0
Middle 15	160	106	1466	0	0	0	0
Late 1	701	262	363	498	158	0	0
Late 2	101	77	115	112	78	0	0
Late 3	815	215	667	161	135	0	0
Late 4	1	2	3	1	1	2	9
Late 5	51	70	76	104	48	52	160
Late 6	77	134	349	288	117	94	295
Late 7	355	163	683	54	141	0	0
Late 8	82	67	178	257	76	65	549
Late 9	983	87	206	69	57	0	0
Late 10	67	1206	87	0	0	53	568
Late 11	357	149	213	0	0	200	685
Late 12	898	21	612	0	0	0	0
Late 13	19	153	1438	0	0	0	0
Late 14	755	166	717	0	0	0	0
Late 15	961	329	562	0	0	0	0

4.5.2 Comparison to Other Approaches

In this Section, we compare our ALNS to the only other ALNS approach by Phillips et al. [POW21], the second place in the competition who used simulated annealing (SA) [RPGS22] (they also improved their results post competition), the winners of the competition who used an ILP-based fix and relax approach [LFMSP21] as well as the first break heuristic by Van Bulck and Goossens [VG23a] which emerged after the competition. We also provide the best-known objective values as well as the best objective values from the ITC2021 [VG23b]. Table 4.20 compares the best objective values each method was able to generate for the instances of the ITC2021. We see that our ALNS outperforms the state-of-the-art ALNS method by Phillips et al. [POW21] on 33 out of the 45 instances. We are also able to provide feasible solutions for three more instances than them. The fix and relax heuristic as well as the break first heuristic both provide many of the best-known solutions. While the approach by Lamas-Fernandez et al. [LFMSP21] has great results on almost every instance the success of the break-first heuristic is highly instance-dependent and they are only able to produce feasible results on 34 of the 45 instances. Note that the results shown in Table 4.20 do not include our 24-hour runtime experiment shown in Table 4.18 since we only performed a single run which is not enough data to draw strong conclusions. However, including those results, we would have three unique best-known solutions instead of one.

Although it has been mentioned throughout the paper we also want to do a final comparison of the various runtimes of the approaches. Even though the runtime highly depends on the hardware used we think it is still worth mentioning in what approximate time frame the solutions were produced. For this, we compare the runtime as it was reported in the various papers. In the case of the other ALNS by Phillips et al. [POW21] they used two measures namely the actual runtime (on 4 "c2-standard-30" virtual machine instances) and the equivalent on a consumer CPU, we provide the latter in the Table 4.20. We also exclude the runtime of instances that reached optimality within the timeframe (Late 4) as it would skew the results (it took us between 30 seconds and 2 minutes to reach optimality on this instance). Not all teams provided complete data regarding runtime and amount of trials. The break-first heuristic is split into two parts. They use three experiments for the generation which range from 12h to 24h runtime and afterwards, they do 50 runs of variable neighborhood search with different random seeds that run for 1 hour and 45 minutes each. The fix and relax heuristic uses by far the most resources with 60 runs per instance that each last up to 6 days (144 hours). Simulated annealing uses a comparatively small runtime for each run but they do at least 48 and sometimes more than 100 runs on each instance. Finally, our approach uses a much shorter runtime than all other ILP-based approaches and while this means that we can't quite compete with their best-known solutions it provides a good alternative when the goal is to find a good schedule with a low amount of resources and already outperforms the previous state-of-the-art ALNS by Phillips et al. [POW21] both in runtime and solution quality. Future experiments will show if with longer runtimes our approach will reach similar objective values as shown by other teams.

Table 4.20: Comparison of state-of-the-art methods from literature to our new ALNS approach.

Instance	Our ALNS	Other ALNS	SA	Fix and Relax	Break First	Best ITC2021	Best-known
Early 1	386	666	423	362*	674	362	362
Early 2	247	379	318	222	320	160	160
Early 3	1105	1171	1068	1052	1084	1012	1012
Early 4	889	inf	556	536	inf	512	512
Early 5	inf	inf	4117	3127*	inf	3127	3127
Early 6	4058	4821	3927	3714	inf	3352	3352
Early 7	6342	7208	5205	4763*	6092	4763	4763
Early 8	1371	1191	1051*	1114	1620	1064	1051
Early 9	452	447	132	108	56*	108	56
Early 10	inf	inf	4986	3400*	inf	3400	3400
Early 11	5644	6713	4526	4436*	8769	4436	4436
Early 12	765	925	1010	510	320*	380	320
Early 13	332	382	173	121*	230	121	121
Early 14	65	106	63	47	42	4	4
Early 15	4284	4667	3556	3368	3110*	3368	3110
Middle 1	inf	inf	5657	5177*	inf	5177	5177
Middle 2	inf	inf	inf	7381*	inf	7381	7381
Middle 3	9426*	11235	9542	9800	inf	9701	9426
Middle 4	9	7*	16	7*	55	7	7
Middle 5	472	681	510	494	295*	413	295
Middle 6	1615	2026	1701	1275	1485	1125	1125
Middle 7	2742	3317	2203	2049	3786	1784	1784
Middle 8	180	277	136	129*	235	129	129
Middle 9	1085	1315	640	450	440*	450	440
Middle 10	1367	2370	1357	1250*	1770	1250	1250
Middle 11	2923	3143	2696	2608	inf	2511	2511
Middle 12	954	911*	950	923	599*	911	599
Middle 13	744	1044	362	282	1835	253	253
Middle 14	1418	1704	1172	1323	1140*	1172	1140
Middle 15	1266	1401	985	965	1205	495	495
Late 1	2113	2406	2021	1969*	2279	1969	1969
Late 2	5860	inf	5715	5400*	5429	5400	5400
Late 3	2617	2900	2457	2369*	2772	2369	2369
Late 4	0*	0*	0*	0*	220	0	0
Late 5	inf	inf	2341	2218	inf	1939	1939
Late 6	1216	1310	930	923*	inf	923	923
Late 7	2228	2805	1765	1652	1997	1558	1558
Late 8	1077	1252	997	934*	1239	934	934
Late 9	1059	1343	715	563	527*	563	527
Late 10	2341	inf	2571	2031	inf	1988	1988
Late 11	236	376	207*	226	421	207	207
Late 12	5004	5542	3944	3912	4010	3689	3689
Late 13	2779	3099	1868	2110	2995	1820	1820
Late 14	1490	1714	1202*	1363	1219	1206	1202
Late 15	140	80	60	40	0*	20	0
Total feasible	40	37	44	45	34		
Total unique best	1	1	3	15	9		
Runtime (h)	6-9	24-48	1.5-12.7	?-144	24 (+1.75)		
Amount of runs	5-10	?	48-101+	60	3 + 50		

Lastly, we also believe that it provides value to compare average results rather than the best-known objective values since this gives less importance to statistical outliers that come from running the approach many times. However, the only state-of-the-art approach that provides data on their average results is the SA approach by Rosati et al. [RPGS22] which means that we can't compare to any other ILP-based approaches. However, this approach is the most similar in runtime to ours which gives a somewhat fair comparison even though the heuristic methods are fundamentally different. Table 4.20 shows that our ALNS has a better average objective value on 14 of the instances while the SA approach has a better objective value on 25 of the instances. Those are really good results for us considering that we often stopped an experiment when there was still a good chance for further improvement, while the SA method stopped when reaching the minimal temperature at which point further improvements would have been unlikely (at least without reheating methods).

Table 4.21: Comparison of average objective value between our approach and SA by Rosatti et al. [RPGS22]
.

Instance	Our ALNS	SA	% dif
Early 1	527	541	-2,6
Early 2	321	385	-16,6
Early 3	1222	1177	3,8
Early 4	1590	1008	57,7
Early 5	inf	inf	-
Early 6	4377	4543	-3,7
Early 7	7392	6722	10
Early 8	1524	1152	32,3
Early 9	598	229	161,1
Early 10	inf	inf	-
Early 11	6302	5785	8,9
Early 12	826	1200	-31,2
Early 13	373	234	59,4
Early 14	104	82	26,8
Early 15	4517	3946	14,5
Middle 1	inf	6075	-
Middle 2	inf	inf	-
Middle 3	10943	11403	-4
Middle 4	13	33	-60,6
Middle 5	524	624	-16
Middle 6	1844	2186	-15,6
Middle 7	3076	2453	25,4
Middle 8	239	197	21,3
Middle 9	1185	772	53,5
Middle 10	1487	1688	-11,9
Middle 11	3051	2997	1,8
Middle 12	1086	1054	3
Middle 13	821	479	71,4
Middle 14	1556	1305	19,2
Middle 15	1337	1100	21,5
Late 1	2339	2373	-1,4
Late 2	5890	6086	-3,2
Late 3	2882	2718	6
Late 4	0	0	0
Late 5	inf	inf	-
Late 6	1270	1121	13,3
Late 7	2627	2227	18
Late 8	1138	1155	-1,5
Late 9	1195	881	35,6
Late 10	2341	3527	-33,6
Late 11	286	289	-1
Late 12	5140	4831	6,4
Late 13	2901	2286	26,9
Late 14	1595	1326	20,3
Late 15	199	83	139,8
Total better	14	25	

CHAPTER 5

Conclusion and Future Work

In this thesis, we looked at a new ILP-based ALNS approach to the DRRST problem that involves a multi-armed bandit formulation for neighborhood type selection, a new multi-stage approach for efficient generation of feasible solutions as well as some new heuristics that help to escape local optima. We also designed six new neighborhood types, that can be used for any ILP-based heuristic.

The evaluation of our ALNS approach shows that it is highly effective compared to previous ALNS approaches even when using much lower computational resources. However, with the current self-imposed time limits, we were only able to go beyond the best-known solutions on 3 out of 45 instances of the ITC2021. Nevertheless, it is shown that we achieved similar average results as the state-of-the-art in simulated annealing on most of the instances which is the only modern heuristic that uses similar runtime.

Throughout the various experiments, some insights were gained into properties that can be used to solve the DRRST more efficiently. The following statements can be considered our main contributions:

- A thorough analysis was made of both existing and newly developed neighborhood types and identified that the effectiveness of each neighborhood type is highly instance-dependent. This speaks for the importance of using adaptive methods since it is hard to predict the strength of the various neighborhood types based on the metadata alone. We also found that some neighborhood types are not very useful in general while others only work well for instances of smaller size.
- It is very important to handle the different constraint types with care when trying to generate feasible solutions as some constraints become much harder to fulfill once a lot of the schedule is fixed. We showed that handling those constraints first usually results in a quicker generation of feasible solutions.

- We identified that the DRRST problem has a very high amount of hard-to-escape local optima, which is also the reason why the currently most successful heuristic methods use a lot of separate runs to achieve the best possible objective values. In this thesis, we show a way to escape such local optima in ILP-based approaches.
- In regards to adaptivity we showed that the UCB multi-armed-bandit method outperformed the non-stationary ϵ-greedy formulation with optimistic initial rewards for the selection of neighborhood types.

Our experiments also showed a lot of potential for future research. First of all, it is interesting how an ALNS approach such as ours would perform with similar computational resources as other ILP-based heuristics. There is also a lot of potential to further improve the adaptive aspects of such a heuristic. Examples of such improvements include a more adaptive selection of teams and slots once the neighborhood type is fixed, including non-ILP-based methods for schedule improvements instead of following strict destroy and repair cycles, and improved more flexible heuristics to escape local optima. In general, we need more research in regards to how humans handle the problem when trying to manually schedule such sports tournaments. The knowledge gained through such research then has a high potential to improve the current automatic methods.

List of Tables

List of Algorithms

Bibliography

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[Ben05] Jacques F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Comput. Manag. Sci.*, 2(1):3–19, 2005.

[BG23] David Van Bulck and Dries R. Goossens. A traditional benders' approach to sports timetabling. *Eur. J. Oper. Res.*, 307(2):813–826, 2023.

[Bri08] Dirk Briskorn. *Sports leagues scheduling: models, combinatorial properties, and optimization algorithms*, volume 603. Springer Science & Business Media, 2008.

[BW77] Bryan C. Ball and Dennis B. Webster. Optimal scheduling for even-numbered team athletic conferences. *A I I E Transactions*, 9(2):161–169, 1977.

[DGM+07] Guillermo Durán, Mario Guajardo, Jaime Miranda, Denis Sauré, Sebastian Souyris, Andrés Weintraub, and Rodrigo Wolf. Scheduling the chilean soccer league by integer programming. *Interfaces*, 37(6):539–552, 2007.

[dW81] Dominique de Werra. Scheduling in sports. *Annals of Discrete Mathematics (11), volume 59 of North-Holland Mathematics Studies*, pages 381–395, 1981.

[FT22] George H. G. Fonseca and Túlio A. M. Toffolo. A fix-and-optimize heuristic for the ITC2021 sports timetabling problem. *J. Sched.*, 25(3):273–286, 2022.

[Glo89] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[GS12] Dries R. Goossens and Frits C. R. Spieksma. Soccer schedules in europe: an overview. *J. Sched.*, 15(5):641–651, 2012.

[LEF+22] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022.

[LFMSP21] Carlos Lamas-Fernandez, Antonio Martinez-Sykora, and Chris N Potts. Scheduling double round-robin sports tournaments. In *Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling-PATAT*, volume 2, 2021.

[POW21] Antony E Phillips, Michael O'Sullivan, and Cameron Walker. An adaptive large neighbourhood search matheuristic for the itc2021 sports timetabling competition. In *Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling-PATAT*, volume 2, 2021.

[Rob52] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[RP06] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transp. Sci.*, 40(4):455–472, 2006.

[RPGS22] Roberto Maria Rosati, Matteo Petris, Luca Di Gaspero, and Andrea Schaerf. Multi-neighborhood simulated annealing for the sports timetabling competition ITC2021. *J. Sched.*, 25(3):301–319, 2022.

[RUdW23] Celso C. Ribeiro, Sebastián Urrutia, and Dominique de Werra. A tutorial on graph models for scheduling round-robin sports tournaments. *International Transactions in Operational Research*, 30(6):3267–3295, 2023.

[SB15a] Kate Smith-Miles and Simon Bowly. Generating new test instances by evolving in instance space. *Comput. Oper. Res.*, 63:102–113, 2015.

[SB15b] Kate Smith-Miles and Simon Bowly. Generating new test instances by evolving in instance space. *Comput. Oper. Res.*, 63:102–113, 2015.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SBWL14] Kate Smith-Miles, Davaatseren Baatar, Brendan Wreford, and Rhyd Lewis. Towards objective measures of algorithm performance across instance space. *Comput. Oper. Res.*, 45:12–24, 2014.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[SL12] Kate Smith-Miles and Leo Lopes. Measuring instance difficulty for combinatorial optimization problems. *Comput. Oper. Res.*, 39(5):875–889, 2012.

[SWS+22] Nícollas Silva, Heitor Werneck, Thiago Silva, Adriano C. M. Pereira, and Leonardo Rocha. Multi-armed bandits in recommendation systems: A survey of the state-of-the-art and future directions. *Expert Syst. Appl.*, 197:116669, 2022.

[VBGSG20] David Van Bulck, Dries Goossens, Jörn Schönberger, and Mario Guajardo. Robinx: A three-field classification and unified data format for round-robin sports timetabling. *European Journal of Operational Research*, 280(2):568–580, 2020.

[VG23a] David Van Bulck and Dries Goossens. First-break-heuristically-schedule: Constructing highly-constrained sports timetables. *Operations Research Letters*, 51(3):326–331, 2023.

[VG23b] David Van Bulck and Dries Goossens. The international timetabling competition on sports timetabling (itc2021). *European Journal of Operational Research*, 308(3):1249–1267, 2023.

[VM05] Joannès Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, volume 3720 of *Lecture Notes in Computer Science*, pages 437–448. Springer, 2005.

[Wat89] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge United Kingdom, 1989.