Informatics

Hybrid Approaches to Sports League Scheduling using Constraint Programming and Simulated Annealing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Bernhard Neumann, BSc
Matrikelnummer 01634034

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Proj.Ass. Dipl.-Ing. Nikolaus Frohner, BSc

Wien, 26. Jänner 2023

[illegible]

Bernhard Neumann   Günther Raidl

Informatics

Hybrid Approaches to Sports League Scheduling using Constraint Programming and Simulated Annealing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Bernhard Neumann, BSc
Registration Number 01634034

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Proj.Ass. Dipl.-Ing. Nikolaus Frohner, BSc

Vienna, 26th January, 2023

[illegible]

Bernhard Neumann   Günther Raidl

Erklärung zur Verfassung der Arbeit

Bernhard Neumann, 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, 26. Jänner 2023

Bernhard Neumann

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich in meinem Studium unterstützt und insbesondere während der Erstellung dieser Diplomarbeit motiviert haben. Ich bedanke mich bei meinem Betreuer Günther Raidl, dass er mir die Möglichkeit gegeben hat, das Thema Sports League Scheduling genauer zu erforschen. Besonders bedanken möchte ich mich auch bei Nikolaus Frohner, der mir stets mit Rat und Tat zur Seite gestanden ist. Weiters möchte ich mich bei meinem Cousin Sebastian Neumann für das Korrekturlesen der Arbeit bedanken. Außerdem möchte ich mich bei meinen Eltern bedanken, die mich in meiner Ausbildung stets unterstützt haben. Zu guter Letzt danke ich meiner Freundin Julia für den emotionalen Rückhalt während des Studiums und insbesondere während der Erstellung dieser Diplomarbeit.

Kurzfassung

Ligenplanungsprobleme befassen sich mit der Erstellung von Spielplänen für Teams in jeweils einer bestimmten Sportliga mit ihren Regeln und einem geteilten Verständnis davon, was einem guten Plan entspricht. Sie gelten als schwierige kombinatorische Optimierungsprobleme, bedingt durch ihre hohe Anzahl von Constraints, welche es erschweren, schon allein gültige Lösungen zu finden, gepaart mit einer Optimierungskomponente, bei der Abweichungen von gewünschten positiven Eigenschaften minimiert werden sollen.

In dieser Arbeit untersuchen wir hybride Ansätze basierend auf dem Constraint Programming (CP) Paradigma und der Metaheuristik Simulated Annealing (SA) angewandt auf zwei herausfordernde Probleme aus dieser Klasse, nämlich das Traveling Tournament Problem (TTP) und das International Timetabling Competition Problem 2021 (ITC2021). Das TTP versucht die grundsätzliche Schwierigkeit in einer möglichst einfachen Problemstellung zu erfassen, während das ITC2021 auf Echtwelt-Probleminstanzen mit ihrer Vielzahl an Constraints und Optimierungskriterien abstellt.

Konkret implementieren wir den Traveling Tournament Simulated Annealing (TTSA) Ansatz für das TTP von Anagnostopoulos et al. in der Programmiersprache Julia und kombinieren diesen mit einem eigenen CP-Modell zum schnellen Finden von zufälligen Startlösungen. Wir können damit Experimente mit schneller Abkühlung für Benchmark-Instanzen aus der Literatur erfolgreich reproduzieren. Wir erweitern TTSA um die Constraints und Zielfunktion des ITC2021 Problems, nehmen einige algorithmische Verbesserungen vor und führen anschließend entsprechende Tuning-Experimente durch, um ein Portfolio von günstigen Parameter-Konfigurationen zu bestimmen.

Zum Finden gültiger Lösungen entwerfen wir zuerst ein Transformationsprogramm, welches die per XML spezifizierten ITC2021 Instanzen in die Sprache der CP Modellierungssoftware MiniZinc übersetzt. Dies gestattet das effiziente Vergleichen von vier unterschiedlichen Backend-Solvern mit verschiedensten Konfigurationen und Restarting-Mechanismen. Die hohe Anzahl an Constraints hat eine Verwendung von CP nahegelegt, jedoch vermochten wir damit nur für etwas mehr als die Hälfte der 45 bereitgestellten Testinstanzen gültige Lösungen zu finden.

In einer parallelen Hybridisierung verknüpfen wir nun CP mit SA. Dabei werden je Instanz mehrere Threads gestartet, wobei jeweils zuerst versucht wird, über den Chuffed CP-Solver mit eigener Randomisierung eine gültige Lösung innerhalb eines kürzeren

Zeitlimits zu finden. Diese wird anschließend dem SA zur Verbesserung übergeben, wobei eine Konfiguration aus dem Portfolio zufällig selektiert wird. Sollte über CP keine gültige Lösung gefunden werden, so wird dies dem SA überlassen, welcher auch mit ungültigen Lösungen umgehen und Constraint-Verletzungen reduzieren kann. Damit können wir zumindest für zwei Drittel der Instanzen gültige Lösungen finden, was eine Verbesserung gegenüber reinen CP-Optimierungsläufen darstellt. Außerdem können wir oft CP-Startlösungen verbessern.

Gesamt betrachtet können unsere Lösungen mit denen der anderen ITC2021 Kompetitoren nicht mithalten. Insbesondere erscheint eine Verwendung von Integer Programming effektiver als CP zu sein. Allerdings stellte sich der hybride Lösungsansatz für uns als vorteilhaft dar, da wir teilweise Lösungen nur in der CP-Phase, teilweise nur in der SA-Phase fanden. Wir hatten aber auch Instanzen, bei denen SA die durch CP gefundene Lösung deutlich verbesserte.

Abstract

Sports league scheduling deals with constructing solutions in form of tournaments for a set of teams following rules of a considered sports league and a corresponding shared view on what good solutions should look like. They constitute difficult combinatorial optimization problems due to their large number of constraints, which makes finding valid solutions alone a challenging task, paired with an optimization aspect, which aims at minimizing the deviation from desired positive tournament properties.

In this thesis, we investigate hybrid approaches based on the constraint programming (CP) paradigm and the metaheuristic simulated annealing (SA) applied on two concrete problems of this class, namely the Traveling Tournament Problem (TTP) and the International Timetabling Competition 2021 problem (ITC2021). The TTP aims at capturing the essence of the complexity of sports league scheduling while the ITC2021 is motivated by real-world instances with their plethora of hard and soft constraints.

More concretely for the TTP, we implement and evaluate the Traveling Tournament Simulated Annealing (TTSA) approach from Anagnostopoulos et al. in the Julia programming language and combine it with a CP model to quickly find random initial solutions. This allows us to successfully reproduce fast cooling experiments from the literature on benchmark instances. We adapt the TTSA further for the constraints and objective function of the ITC2021 problem, implement some algorithmic improvements, and perform final tuning experiments to obtain a portfolio of promising configurations.

To find initial solutions, we design and implement a tool which transforms the ITC2021 XML instances into the language of the CP modeling software MiniZinc, which allows us to efficiently compare four different backend solvers with various configurations and restarting mechanisms. Given the highly constrained nature of the problem, we assumed CP to work better, however, with this pure CP approach we could only find feasible solutions for slightly more than half of the 45 instances.

In a parallel hybridization, we finally combine CP with SA. For each instance, multiple threads are started in each of which we first search for a feasible solution via the Chuffed CP solver using randomization and a shorter time limit. The solution is subsequently provided to the SA to improve it, for which we sample a configuration from the aforementioned portfolio randomly. In case no feasible solution is found by CP, this then becomes the task of SA, which can handle infeasible solutions and reduce constraint violations. This allows

us to find feasible solutions for two thirds of the instances, which is an improvement compared to pure CP runs. Moreover, we are able to frequently improve initial CP solutions while sometimes full CP runs are better.

Still, the overall performance is below other ITC2021 competitors; we cannot reach the best known solutions by a substantial margin. In particular, it seems preferable to use integer programming instead of CP. In our setting, it turned out to be beneficial to have a hybrid approach since we dealt with instances where only CP found a solution and instances where only SA found a solution. We also had instances where the solution found by CP was improved substantially by the SA.

Contents

CHAPTER 1

Introduction

Scheduling sports leagues satisfactorily is a difficult and complex task in which a number of various important stakeholders are involved. In our context, it is about assigning games to slots in a previously agreed upon calendar fulfilling certain constraints while maximizing a defined quality measure. The teams, their supporters, the media, the public, and the league itself each have their own, often conflicting, interests in how a schedule should be structured. Teams and supporters each do no want to have too long away streaks, the media and the public want to have certain high class games not in conflict with other events of general interest, the league requires the schedule to have a specific form etc.

While the current approach by sports timetabling practitioners is to focus on a specific league and compare schedules of different algorithmic approaches with each other and with manually derived schedules, in the International Timetabling Competition 2021 (ITC2021) the participants' task is to create and describe a corresponding general solver able to find and optimize schedules for any league specified. To test their performance and organize an actual competition, a number of different leagues, concrete problem instances for the solvers, are provided.

The well-known Traveling Tournament Problem (TTP) introduced by Easton et al. [10] in 2001 is more of an academic sports timetabling problem where the goal is to find a double round robin (DRR) schedule for a league where the total distance traveled by all teams shall be minimized. It assumes that teams travel directly from venue to venue, starting and ending at their home venue. The real-world motivation comes from the Major League Baseball in the United States, where travel distances between opposing teams are large. Moreover, two additional hard constraints have to be met: The *no-repeat* constraint ensures that two teams do not play against each other in two consecutive rounds. The *at-most* constraint guarantees that a team plays at most U games consecutively at home or away with U usually set to 3, and therefore a balance in the schedule regarding home and away streaks is achieved. It is an $\mathcal{NP}$-hard problem [45] and has proven to be very difficult in practice—benchmark TTP leagues with 12 teams have not yet been solved to optimality—and therefore acts as a benchmark problem for both exact and metaheuristics approaches, from which solution approaches to the more realistic ITC2021 leagues could draw inspiration.

1.1 Aim of this Work

The aim of the work is to implement a heuristic solver for ITC2021 sports leagues. For the TTP, a number of metaheuristic approaches have been applied successfully. Therefore, we want to explore if those approaches, when suitably adapted, also work well for ITC2021 leagues in which there are other constraints and in a larger number than for the TTP, moreover, a different objective function where the traveled distance is irrelevant and instead the unsatisfied soft constraints are taken into account.

First, we want to find a constraint programming (CP) model which should provide us with initial feasible solutions to be subsequently improved using a metaheuristic search. Different backend solvers should be compared by testing and tuning corresponding search parameters for the relevant benchmark instances. As mentioned before, approaches based on metaheuristics were already successfully applied to the TTP, see [1, 53, 46, 8]. Hence, we want to design and implement a standalone (meta)heuristic algorithm also capable of generating an initial solution and compare it with the pure CP approach.

We also want to answer the question whether it is more beneficial to start our improvement heuristic with a feasible solution and spend a lot of time generating the initial feasible solution or if it suffices to start from a random, possibly infeasible one, and then spend a lot of time in the improvement phase which is able to deal with infeasible solutions. In this spirit, we want to finally study a hybrid approach combining the constraint programming solver with the metaheuristic in a two-phase approach.

1.2 Outline

In Chapter 2, we discuss state-of-the-art approaches to the ITC2021 and simulated annealing for the TTP. In Chapter 3, we present the methodology related to our solution approaches, concretely basic definitions related to combinatorial optimization, the foundations of constraint programming to solve constrained optimization problems, and simulated annealing as neighborhood-based improvement metaheuristic. Then we turn to the problem definition of the ITC2021 in Chapter 4 and also introduce the solution representation based on splitting a schedule in opponents and home-away patterns [37]. Our solution approaches based on CP and SA are presented in the subsequent Chapter 5 before we move to our thorough computational study in Chapter 6. We conclude and provide our thoughts on future work in Chapter 7.

1.3 Key Results

We summarize our key results and findings in the following list:

- We provide an effective and efficient solver for the TTP based on a CP model with globals and restarting and a reimplementation of TTSA [1] in Julia, confirming results from the literature.

- While Chuffed with heavy restarting seems to be the most promising CP solver, we could not find an effective standalone CP approach using the opponents and home-away pattern formulation to find feasible solutions for the ITC2021 with high probability within 24 hours.

- We were able to improve the CP approach in terms of feasibility and sometimes in terms of solution quality by parallelizing and hybridizing it with SA while tuning the latter properly over a wider range of instances with different properties remains a challenging task.

- The comparison of pure CP (with Chuffed and randomization) and a standalone SA approach shows that CP finds solutions to eight of the Early instances whereas SA finds solutions to only seven of those. Both approaches used multiple configurations and runs (CP: three configs with 60 runs each, SA: 96 configs with three runs each) to solve the instances. Each run had two hours to solve its instance. This means the question if one should put more effort into finding feasible initial solutions with CP or into the SA part, cannot be answered definitely as it highly depends on the instance. Since the performance of CP was rather disappointing, we believe it is better to put more effort into the improvement phase.

CHAPTER 2

State of the Art

In this chapter, we present the state of the art regarding sports timetabling in form of solution approaches to the challenging ITC2021 and the groundbreaking works on simulated annealing for the TTP and a recent heuristic approach based on randomized beam search. The ITC2021 is introduced by Van Bulck et al. [50], the TTP by Easton et al. [10]. For an overview on sports league scheduling in general, refer, for instance, to De Werra [7], Kendall et al. [24], and Durán [9].

2.1 ITC2021 Solution Approaches

Berthold et al. [3] used Mixed-Integer Linear Programming (MILP) with several different solvers to find solutions, also making use of a restarting scheme. In a final step, they used an adapted traveling tournament simulated annealing (TTSA) algorithm as described by Anagnostopoulos et al. [1] to improve their best found solutions.

In their work, van Doornmalen et al. [51] made use of the circle method proposed by Anderson [2] to construct the initial schedule (phase 1), which is very likely to violate hard constraints. Then they implemented a combination of MILP and a local search in simulated annealing fashion that explores the neighborhood mentioned by Januario et al. [22]. In phase 2, they use this combination to generate a feasible schedule out of the initial schedule adhering to all hard constraints. In phase 3, they try to improve the found solution using the same approach but only allow for feasible solutions.

Sumin and Rodin [43] used four different MILP models to narrow the search space and thereby ease the finding of solutions. Their baseline model was a complete MILP formulation of the problem but could only be used for smaller instances with fewer constraints; their pattern model used additional variables which define if a team plays at home in a specific round; their patterns mirrored model generates mirrored schedules, thereby reducing the search space significantly. In their two-phased model they decompose the model into the two halves of the schedule, first solving the first part, then scheduling the second phase with respect to the solution of the first phase.

Philips et al. [36] first try to solve a MILP model to create an initial solution. If this does not yield a solution in feasible time, they use the Canonical Factorization of De Werra [7] which guarantees to satisfy one of the break constraints but will most likely violate other hard constraints. Afterwards, they iteratively apply an Adaptive Large Neighborhood Search to improve the initial solution including solutions violating hard constraints as well.

Fonseca and Toffolo [12] used a fix-and-optimize algorithm, which is a matheuristic that iteratively employs a mathematical programming solver to optimize a sub problem while the rest of the problem remains fixed. They apply their procedure twice, once with an initial schedule obtained by the polygon method to find a feasible solution and once with the found feasible solution to improve that solution.

In their work, Lamas-Fernandez et al. [26] developed a MILP model to solve the instances. However, since the instances are too hard to solve as a whole, they used a fix-and-relax matheuristic approach to solve subproblems.

In their master's thesis, Subba and Stordal [42] also used a MILP approach to search for solutions as well as a cluster pattern approach to fix the home-away status of a subset of teams to reduce the search space.

Rosati et al. [39] implemented a three-stage simulated annealing approach, which makes use of a new neighborhood (in addition to the five neighborhoods used by Anagnostopoulos et al. for the TTP, see Section 2.2). The new neighborhood was created specifically for the phased version of the problem (see Section 4.2 for the definition of phased instances).

2.2 Traveling Tournament Problem

Anagnostopoulos et al. [1] proposed a simulated annealing approach to solve the TTP called TTSA. They used a random initial schedule and also considered infeasible solutions to escape local minima and address the rather small feasible neighborhood even with already quite complicated moves. Moreover, they implemented reheating to enlarge the considered solution space during one run. Strategic oscillation is used to balance the time spent in feasible and infeasible regions. Their approach works well for instances up to 14, beyond that point the investigation of the neighborhoods needs a lot of time. However, they also studied a fast cooling approach with which they also found good solutions for larger instances rather quickly.

In a follow-up work, Van Hentenryck and Vergados [53] implemented a population based variant of their simulated annealing approach, in which they used the TTSA of [1] as a black box. The algorithm works in waves. First, TTSA is run a number of times to create the population; after a certain time, the best solutions so far are compared and the runs within a subset of the n best solutions may continue their execution while the rest of the population is restarted with the overall best solution found so far. This approach has found most of the currently best known solutions for the larger instances of the TTP.

Frohner et al. [13] proposed and implemented a beam search approach for the TTP. For guidance of the beam search, different lower bounds variants were compared based on the independent lower bound [10, 47, 48]. Moreover, a randomized beam search variant for which a Gaussian noise is added to the heuristic value was also implemented and used to diversify multiple runs. This approach achieved improvement for two CIRC instances. In the parallelized variant of this randomized beam search approach [14], many new best feasible solutions to the TTP were found.

CHAPTER 3

Methodology

In this chapter, the main concepts relevant for our solution approaches are explained. First, we define combinatorial optimization problems, then we explain the ideas of constraint programming, an exact approach on finding a solution, followed by a description of heuristic optimization, a non-exact approach that sacrifices optimality for speed, including heuristics for constructions as well as for improvement.

For the definition of combinatorial optimization problems (COP) we follow Papadimitriou and Steiglitz [33] (restricted to finite solution sets). The goal of an optimization problem is to find a best solution out of a set of candidate solutions. This separates them from decision problems and satisfaction problems. Decision problems only answer a specific true-false question. For example, a question asked by a decision problem could be "Is there a feasible schedule for that tournament?". Solutions to these problems answer either yes or no. Moreover, satisfaction problems deal with finding a feasible solution that satisfies all constraints, e.g., a solution would give a feasible schedule to a tournament. On the other hand, an optimization problem might be "What is a schedule with the smallest objective value?", whose solution would give a concrete example of a best schedule regarding the objective value. The objective value of a solution is defined by the objective function of its related problem instance. An optimization problem can either be a minimization problem where one wants to minimize the objective value or a maximization problem where one wants to maximize the objective value. However, a maximization problem can be turned into a minimization problem by multiplying the objective values by -1. Therefore, in the following we will only elaborate on minimization problems. The set of solutions to an optimization problem can either contain finitely or infinitely many solutions. In the first case, we call these problems combinatorial optimization problems. In the latter case, we talk about either discrete optimization in general with a countable but possibly infinite number of solutions, or continuous optimization with (partially) continuous, hence uncountably many, solutions. In this thesis, we will only consider combinatorial optimization problems, which we define as follows:

Definition 1 *A combinatorial optimization problem is a set of problem instances. A problem instance is a pair (S, f) where S is the finite set of all feasible solutions to the instance, S is also called the search space of the problem instance, and f is an objective function $f\colon S \to \mathbb{R}$ that shall be minimized (or maximized). The goal is to find a globally optimal solution $x^* \in S$ such that $f(x^*) \leq f(x)\ \forall x \in S$ for minimization (or $f(x^*) \geq f(x)\ \forall x \in S$ in case of maximization).*

In Figure 3.1, we can see a graph of an objective function with only one parameter. The parameter is represented by the x-axis; the value of the objective function for that parameter is shown on the y-axis. The graph has a global minimum (B) and three local minima (A, B, C). The term global minimum has already been defined; it is the optimal solution to that problem instance. A local minimum is the lowest point on the objective function graph in an immediate neighborhood (which needs to be defined); going from that point in any direction leads to an increase of the objective value. Every global minimum is also a local minimum.

Solving the problems described in this thesis to optimality becomes very time-consuming when the problem size increases; if the problem size grows linearly, it is widely believed that the time needed grows exponentially. To tackle such problems, there are generally two approaches: exact and heuristic methods. Exact methods are guaranteed to find the global optimum at some point in time. Thanks to various techniques, for example, branch-and-bound or constraint propagation that allow to reduce the search space, exact methods do not need to check every possible solution. However, solving a problem to optimality and proving said optimality might still take a lot of time. Therefore, heuristic optimization methods are often used to find feasible, high-quality solution in much less time than an exact approach. These techniques do not guarantee to find a best solution (even if they do, they generally cannot prove that the found solution is optimal) and in general there is no bound on the gap to the best solution quality. For example, the nearest-neighbor heuristic for the Traveling Salesman Problem (TSP), where the goal is

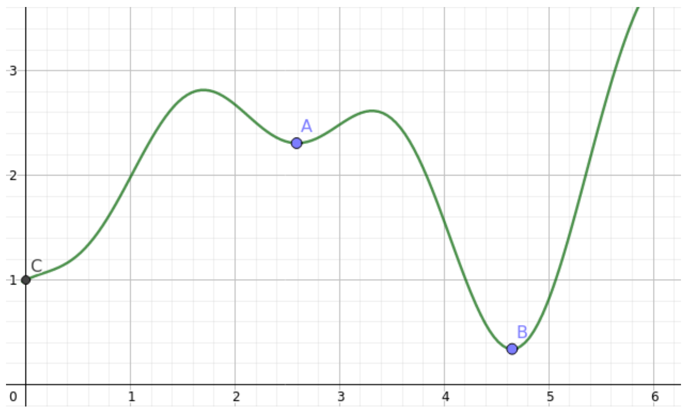

Figure 3.1: Objective function with multiple local minima and one global minimum

to find the shortest tour through a set of cities (nodes), starting and ending at the same city, can be arbitrarily worse than the optimum. However, they are often much faster since they focus on restricted parts of the search space that look promising and may work well on instance classes considered relevant. Some heuristic methods can guarantee that their found solution is not worse than a certain approximation ratio; they are then called approximation algorithms. For example, the Christofides heuristic [5] for the TSP guarantees that its found solution does not have an objective value higher than 1.5 times the optimal value.

Some of the described problems are $\mathcal{NP}$-hard or are conjectured to be $\mathcal{NP}$-hard. $\mathcal{NP}$-hard problems are problems that are at least as hard as all problems in $\mathcal{NP}$, however, they do not have to lie in $\mathcal{NP}$. If a problem is $\mathcal{NP}$-hard and lies in $\mathcal{NP}$, it is $\mathcal{NP}$-complete.

Definition 2 *A problem lies in $\mathcal{NP}$ if and only if the problem is accepted by a non-deterministic Turing machine which operates in polynomial time. Karp [23, p. 91]*

This means that if there are x alternatives for the algorithm to investigate, the algorithm makes x copies of itself and examines all of the alternatives at once. $\mathcal{NP}$ is the abbreviation of Non-deterministic Polynomial runtime. A solution to a problem in $\mathcal{NP}$ can be verified in polynomial time. For example, a non-deterministic Turing machine can find an optimal solution to a TSP(L) instance (TSP's decision variant, where we seek a solution of length below a given L) in polynomial time when it traverses all possible tours in parallel, starting from an initial city.

The evaluation of the tours can also be done in polynomial time since it only needs to sum up the distances between n cities. This is another important aspect of $\mathcal{NP}$ that a witness, in this case a tour of length $\leq L$, can be checked in polynomial time. We can calculate the objective value of a tour for a TSP instance in polynomial time as well as the objective value of a schedule for a TTP instance, both not in $\mathcal{NP}$ but $\mathcal{NP}$-hard problems. So far, no polynomial algorithm for an $\mathcal{NP}$-hard problem has been found; known algorithms for this problem class have exponential runtime in the size of the problem instance; the runtime is upper bounded by $2^{poly(n)}$, where $poly(n)$ is a polynomial in n. On the other hand, there are problems that lie in the complexity class of $\mathcal{P}$, which stands for polynomial runtime. Whether $\mathcal{P} = \mathcal{NP}$ is still an open question.

Definition 3 *A problem lies in $\mathcal{P}$ if and only if the problem is accepted by a deterministic Turing machine which operates in polynomial time. Karp [23, p. 88]*

In the following two sections, we present the basics of two approaches to tackle $\mathcal{NP}$-hard problems: constraint programming and heuristic optimization.

3.1 Constraint Programming

Constraint Programming (CP) is a paradigm for solving combinatorial search problems, which is applied to many domains such as scheduling, planning, vehicle routing, etc. To solve these problems, one has to translate them into a Constraint Satisfaction Problem (CSP), representing the problem in terms of decision variables and constraints (see Definition 4). For this section, we follow the Handbook of Constraint Programming by Rossi et al. [40]. We focus on the basic principles of constraint programming: we define a CSP and a solution to a CSP ([40, 2.2.1]), describe the concepts of constraint propagation [40, 2.2.3, chapter 3] and backtracking ([40, 2.2.4, chapter 4]) and introduce ordering heuristics ([40, 4.6.1]. Further techniques such as global constraints ([40, chapter 6]) or symmetry breaking ([40, chapter 10]) are not discussed but can be found in the Handbook of Constraint Programming. Another topic that is not covered but should be mentioned here is parallelization of constraint programming; for further reading see Régin and Malapert [38].

Definition 4 *A CSP $\mathcal{P}$ is a set of instances $\mathcal{P} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_k\}$ where each $\mathcal{I}$ is a triple $\mathcal{I} = \langle X, D, C\rangle$ where X is an n-tuple of variables $X = \langle x_1, x_2, \ldots, x_n\rangle$, D is a corresponding n-tuple of domains $D = \langle D_1, D_2, \ldots, D_n\rangle$ such that $x_i \in D_i$ and C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t\rangle$. A constraint C_j is a pair $\langle R_{S_j}, S_j\rangle$ where R_{S_j} is a relation on the variables in S_j. In other words, R_{S_j} is a subset of the Cartesian product of the domains of the variables in S_j. [40, Chapter 2.2.1]*

Definition 5 *A solution to a CSP instance $\mathcal{I}_{\|}$ is an n-tuple $A = \langle a_1, a_2, \ldots, a_n\rangle$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds on the projection of A onto the scope S_j. [40, Chapter 2.2.1]*

The set of all solutions to a CSP instance $\mathcal{I}_{\|}$ is represented by $sol(\mathcal{I}_{\|})$. If it is empty, the CSP instance is unsatisfiable, otherwise it is satisfiable, which means there is a solution to the problem instance.

A Constrained Optimization Problem (or Constraint Optimization Problem) is a special form of CSP where in addition to fulfilling all constraints the goal is to minimize an objective function. The parts of the objective function are often called soft constraints because if a soft constraint is not fulfilled, a penalization term is added to the objective function value.

While searching for solutions, CP solvers combine constraint propagation and backtracking to narrow the search space and escape infeasible paths.

3.1.1 Constraint Propagation

Constraint Propagation is used to narrow down the search space and is applied before the search starts and after each assignment of a value to a decision variable. Some constraints directly restrict variables. For example, the constraint $x_1 < 5$ restricts the domain D_i of x_1, every value ≥ 5 can be removed from D_i. Moreover, an assignment of a variable might lead to a restriction of other variables that have not been assigned a value yet. Therefore, the domains of the now restricted variables are reduced by the values which are not possible anymore. The restrictions thereby propagate to future variable assignments, thus reducing the search space. This process is done by consistency algorithms; in the following node consistency and arc consistency are described. For more details and other higher order consistencies see Chapter 3 of the Handbook of Constraint Programming.

The constraints of a CSP can be represented in the form of a constraint graph. A node i represents the decision variable x_i with domain D_i. Constraints that contain only one decision variable, called unary constraints, are usually not shown in a constraint graph, however, one could show them as a loop on the node. Constraints that use two variables, called binary constraints, are shown as arcs between the two variables. For constraints with higher order a hypergraph would be needed.

Node consistency checks the consistency of unary constraints. A node in the constraint graph is consistent if $D_i \subseteq R_i$ where R_i is the shorthand notation for $R_{\langle x_i \rangle}$. If it is not consistent, it can be made by computing $D_i \leftarrow D_i \cap R_i$. Thus, values that would violate the constraint are removed from D_i. The network becomes node-consistent by a single pass through the nodes.

The presented arc consistency algorithm works with binary constraints. Suppose there is a binary constraint, i.e., there is a relation R_{ij} (shorthand for $R_{\langle x_i, x_j \rangle}$) between variables x_i and x_j, then the arc $\langle i, j \rangle$ is consistent if and only if $D_i \subset \pi_i(R_{ij} \bowtie D_j)$, which means it is consistent if every value of D_i has a partner in D_j, so the constraint holds. If it is not consistent, i.e., there is a value in D_i that does not allow for a feasible assignment of a value of D_j to the variable x_j, it can be made by computing $D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$, which means values in D_i that do not have a partner in D_j are removed from D_i. Arc consistency is directional, so it has to be checked for both arcs of two nodes ($\langle i, j \rangle$ and $\langle j, i \rangle$). The deletion of a value in the domain of a variable x_k can lead to any arc x_k is part of to become arc inconsistent; therefore, arcs need to be revisited if one of their variables had its domain changed in the previous step, until all of the arcs are arc consistent.

There are different arc consistency algorithms; one of them is AC3, which was proposed by Mackworth [29] and is outlined in Algorithm 3.1 and 3.2. The revision of an arc happens in the function Revise3, which takes a variable X_i and a constraint c as input and deletes every value from the domain D_i of X_i which has no partner in D_j (X_j is the other variable of the constraint c). It returns true if a value was removed from the domain. AC3 revises the arcs until all arcs are consistent. To save on function calls, it manages a queue of pairs (x_i, c) for which we cannot be sure that $D(x_i)$ is arc consistent. The algorithm takes one of those pairs (x_i, c) and uses Revise to make the variable x_i arc consistent regarding c. If the domain was changed and thereby the domain became empty, we return false as we failed to make the graph arc consistent. If only the domain changed, it can be that a value of another variable lost its partner regarding another constraint, therefore, this variable needs to be revised again and hence is put into the queue. The algorithm stops when the queue becomes empty and then returns true as the graph is now arc consistent.

Algorithm 3.1: Revise3

```
input  : Variable $X_i$; constraint: $c$
output : Boolean
1 CHANGE ← false;
2 foreach $v_i \in D_i$ do
3     if $\nexists v_j \in D_j$ that allows $X_i$ to be assigned $v_i$ wrt $c$ then
4         remove $v_i$ from $D_i$;
5         CHANGE ← true;
6     end
7 end
8 return CHANGE;
```

Algorithm 3.2: AC3

```
input  : Variable set $X$; constraint $c$
output : Boolean
1  $Q \leftarrow (x_i, c) | c \in C, x_i \in X(c)$;
2  while $Q \neq \emptyset$ do
3      select and remove $(x_i, c)$ from $Q$;
4      if Revise($x_i$, $c$) then
5          if $D_i = \emptyset$ then
6              return false;
7          else
8              $Q \leftarrow Q \cup (x_j, c') | c' \in C \wedge c' \neq c \wedge x_i, x_j \in X(c') \wedge j \neq i$ ;
9          end
10     end
11 end
12 return true;
```

The algorithm runs in $\mathcal{O}(ed^3)$ where e is the number of constraints and d is the size of the largest domain. The algorithm is not the most efficient one as Revise does not store any information about the partners of certain values, therefore, it has to recalculate it for every call. However, this algorithm shows the underlying concept of arc consistency very well. There are further enhancements that improve AC3, namely AC4, AC6, and AC2001, which lower the time complexity of the algorithm.

If some constraints use more than two variables, the constraint can be represented as a hyperedge and arc consistency can be generalized to hyperarc consistency.

3.1.2 Backtracking

If constraint propagation does not already provide a feasible solution, CP solvers may use depth first search to find one. Values are assigned to variables heuristically one after another, resulting in partial solutions, for which constraint propagation can be again applied to reduce the domains. However, the solver might often get stuck at nodes with at least one variable with an empty domain, therefore without a feasible completion. In the simplest form of backtracking, the solver will then jump back to the last partial solution where there is a variable assigned which has an alternative choice (i.e. a variable with a domain with more than one value). However, there are techniques to remember which variable assignment caused the failure and then jump back to that assignment. For more details, see Chapter 4 of the Handbook of Constraint Programming [40].

3.1.3 Example

A well known example for a CSP is the n-queens problem where the goal is to place n queens on an $n \times n$ chessboard in such a way that no queen can attack another one. A possible way of modeling that problem is to have a variable for each of the n columns. The value of the variable i defines the row in which the queen is positioned in the ith column. There are three different constraints that need to be satisfied: Firstly, there must be exactly one queen per column; this is always satisfied as there is one variable for each column. Secondly, there needs to be exactly one queen per row, which means we need to ensure that all values of the variables are different. Lastly, there may only be one queen per diagonal, therefore, the values of the variables may not differ by the same amount their respective column's index differs. The problem of finding a solution lies in $\mathcal{P}$ as shown by Hoffman et al. [21]. However, finding a solution to the related problem of n-queens completion where the goal is to find a feasible completion of a partial assignment of an n-queens board is $\mathcal{NP}$-complete as presented by Gent et al. [17].

In Figure 3.2, we can see on the left a queen positioned on a 4×4 chessboard. On the right side of that figure, we can see that the restrictions of the positioning of the first queen propagate to the possible values of the other variables. We use red to mark squares that cannot be used because of a queen in the same column, blue to mark squares that are prohibited because of a queen in the same row and green to mark squares that have a queen on the same diagonal already positioned. In Figure 3.3 on the left side, we can see that positioning the second queen in the third row leaves no possible values for the third queen. Therefore, we have to backtrack and position that queen in the fourth row (on the right side of 3.3). However, this leads to the point where there is no position left for the fourth queen. Therefore, we have to backtrack all the way back to the first queen and position it in the second row which can be seen on the left side of Figure 3.4. Now, the second queen has only one possible position as well as the third and the fourth queen after each step as seen on the right of 3.4 and on the left of Figure 3.5. In the end, all queens are positioned in that way, so no queen can attack another one, which can be seen on the right in Figure 3.5.

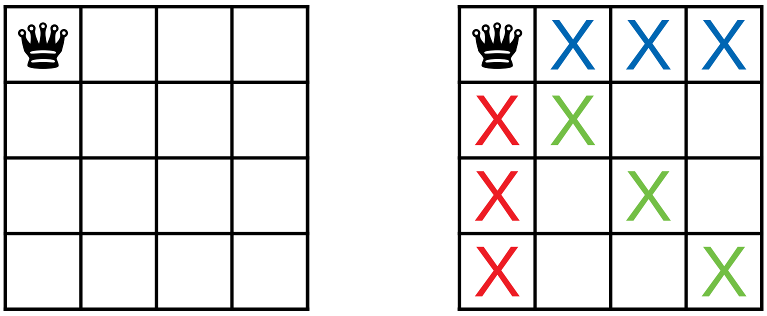

Figure 3.2: Left: First queen positioned in the first row. Right: Marked squares are no possible positions for other queens.

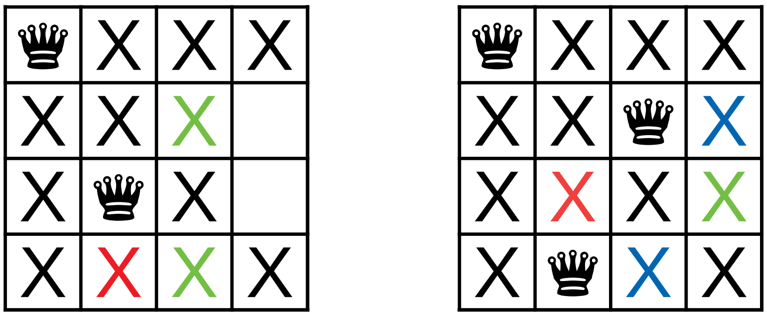

Figure 3.3: Left: Second queen positioned in the third row, which leaves no option for the third queen. Right: Second queen positioned in the fourth row leaves the second row as only option for the third queen, which then leaves no option for the fourth queen.

This problem could also be formulated as a COP. The objective function would then be the number of pairs of queens that can attack each other. The best possible value is therefore 0 when no queen can attack another. The worst possible value is $2(n-1)$ and occurs when all queens can attack each other, as it can be seen in Figure 3.6.

3.1.4 Ordering Heuristics

The ordering of which variables to assign a value first plays an important role in the search process, moreover, which value is assigned is also significant. In the example above, we always chose the variable (=column) with the lowest index to assign the next queen to. Another possible ordering would be to choose the variable with the smallest number of possible values in its domain, thereby, possible failures should get recognized earlier. Furthermore, one could choose a random variable, the variable with the smallest value in its domain, the variable with the highest difference in its lowest two values of the domain or the variable with the largest domain weighted by the number of constraints and how often it caused failure. The opposite of these variable orderings would also be possible. For the decision which value to assign to a variable there are also various heuristics. One possibility, which we also used in the example above, is to always assign

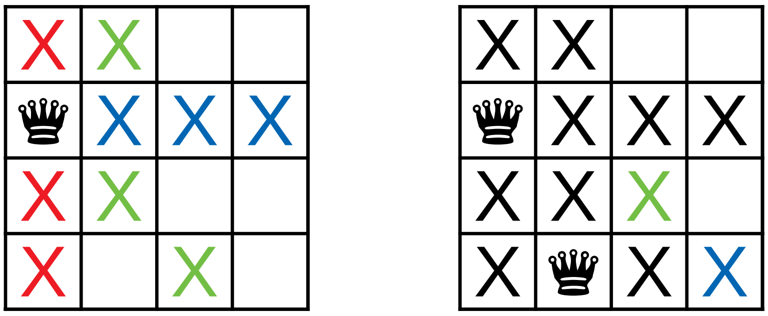

Figure 3.4: Left: First queen positioned in the second row leaves only one position open for second queen. Right: Second queen positioned in the fourth row leaves the first row as only option for third queen.

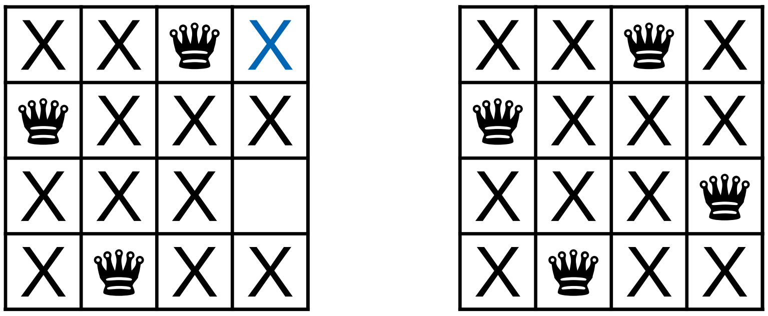

Figure 3.5: Left: Third queen positioned in the first row leaves one position open for last queen. Right: Last queen positioned in the third row, all queens are positioned.

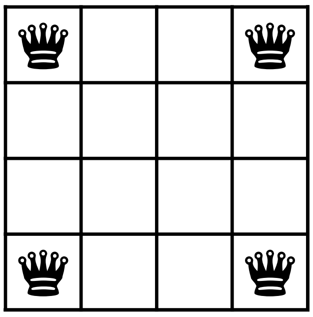

Figure 3.6: Worst possible solution for the n-queens problem as a COP for $n = 4$.

the lowest possible value to the variable. Moreover, one could assign the median value of the domain, the value that is closest to the average of the domain values or a random value of the domain. For further information, see Subsection 4.6 of the Handbook of Constraint Programming [40].

3.1.5 Randomization & Restarting

Often, a whole subtree of the search tree does not lead to a feasible solution. In this case, backtracking inside this subtree only uses up time in particular when we made a bad decision at the beginning of the search. Therefore, the technique of restarting from the initial state is used, in combination with randomization to ensure different resulting schedules.

Of course, during the search it is unknown whether the whole subtree is infeasible, therefore, there are various heuristic ways to determine when to restart. For example, after a number of failures, i.e., when we encounter a node with a variable with an empty domain, or after a defined amount of time in which no feasible solution was found or after an amount of time after which a new layer in a subtree has not found a feasible assignment for its respective decision variable.

Moreover, there are various strategies for restarting. If we consider only restarting after a certain amount of failures, we can still differentiate between the restarting procedures. Constant restarting leads to restarting after a certain number of failures. Others like linear or geometric restarting increase the number of failures that lead to a restart after every restart; this is so the search allows more and more failures because if the number of allowed failures before restarting is too low, the search might never find a solution. There are also strategies where the number of allowed failures increases and decreases along a predefined sequence.

Restarting also requires a form of randomization, otherwise, we would always go through the same nodes of the search tree. Either the variable ordering or the value ordering can be randomized, or both. Variable ordering randomization can be either completely random or guided by a heuristic, e.g., randomly choosing a variable that is within a small factor of the best variable (according to the heuristic). Moreover, one could also only randomize the tie breaking of a variable order heuristic. Important to note is that the randomization method needs to have enough different decisions near the top of the search tree. For value ordering heuristics one could make all values equally likely or also use a heuristic as guidance. For further information, see Section 4.7 of the Handbook of Constraint Programming [40].

An example of randomization and restarting could be the n-queens problem where we restart after a number of failures we encounter and use a randomized value ordering. So after we run out of options to put the third queen on the chessboard as seen in Figure 3.3 on the left, we would restart and choose a random row for the first queen. If we put it in the bottom left, we would still run into a failure. On the other hand, if the random ordering chooses the second or third row, we would come to a solution without further restarts.

3.2 Heuristic Optimization

Heuristic optimization aims to find high-quality solutions to difficult problems in reasonable time. It can be used when exact methods like integer programming [54] or constraint programming [40] fail or are too slow, at the cost of generally not having an optimality guarantee. To create a solution to a given problem instance from scratch, a *construction heuristic* is used whereas to improve an existing solution an *improvement heuristic* is applied. As we mostly formulate the construction of solutions in the context of CP, in this section we focus on improvement (meta)heuristics. Heuristics and metaheuristics are used to guide the search process. Heuristics are usually tailored to one specific problem, metaheuristics on the other hand are algorithmic templates and therefore can be applied to different optimization problems.

3.2.1 Improvement Heuristics

In the field of optimization there are various improvement metaheuristics such as evolutionary algorithms, swarm intelligence based ones like ant colony optimization, or neighborhood-search based ones like tabu search. In this section, we will describe two classic metaheuristics, *local search* and *simulated annealing*. For a state-of-the-art overview, see the Handbook of Metaheuristics by Gendreau and Potvin [16].

Local search is a fundamental improvement metaheuristic. Let the search space consist of all possible solutions S, infeasible or feasible. The local search starts with an initial solution $x_s \in S$ and traverses through the search space by iteratively going to an improving neighbor of the current solution. A neighbor $x' \in S$ of a solution x is another solution that can be obtained by performing a move $m(x)$ on x. For example, by exchanging two nodes in a solution to a TSP instance another solution is obtained. All the neighbors that can be obtained by such a move applied to x make up the neighborhood $N(x)$ of x. The search terminates when a local optimum regarding this neighborhood is reached, which means it cannot be further improved by a neighbor. However, there might be some other local optimum that is better than that local optimum. Avoiding to get stuck in local optima is the main concern of more advanced metaheuristics.

In such neighborhood-based searches, we may also accept infeasible solutions as the next current solution and penalize the constraint violations by adding a certain term to the objective value. This is especially useful for highly constrained problems where finding a feasible solution and moving from one feasible to another feasible solution is difficult or takes multiple steps.

Simulated Annealing was first introduced by Kirkpatrick et al. [25]. It is a metaheuristic used to explore the neighborhood by means of a guided random walk, also accepting worse solutions with certain probability as opposed to local search. It finds its inspiration in the process of a liquid (e.g., a metal) solidifying over time while its temperature decreases. By slowly decreasing the temperature of the substance, the atoms have more time to arrange themselves in the right place and build stable crystals corresponding to a configuration with minimal energy. The varying temperature in the optimization process of simulated annealing is directly related to the probability that a solution might temporarily get worse in order to surpass local optima.

In Algorithm 3.3, we can see the pseudo code for simulated annealing. The algorithm receives an initial solution x as an input parameter. First, we initialize a counter t that is needed for the cooling process and the initial temperature T is set. We also set the best known solution to x_b, which is x at that stage. The outer loop goes on until a predefined stopping criterion is met; this could be a certain temperature reached, no improvements in the last τ iterations or after a certain number of iterations. The inner loop goes on until a predefined equilibrium condition is met, which is typically based on a counter of the iterations of the inner loop. After the loop the temperature gets lowered according to a cooling schedule $g(T, t)$, which can be, for example, a geometric cooling, where every time the current temperature T is multiplied by a cooling factor $\beta < 1$ or it can be based on the iteration counter t.

Inside the inner loop we generate a new solution x' by randomly selecting one of the neighbors $x' \in N(x)$ of the current solution x. If x' is better, then it is accepted as the new current solution x. "To be better" can mean that the objective value of x' might be better (smaller for minimization problems, larger for maximization problems) or that x' has fewer violations than x making it "more" feasible. If x is also better than the best known solution x_b, then x_b is set to x. If x' is not better, then it might still be accepted as the new current solution to enable the algorithm to leave local optima and more freely wander around in the solution space. This is achieved by generating a random number P between 0 and 1. If P satisfies the so-called Metropolis criterion $P < e^{-|f(x')-f(x)|/T}$ where f is the objective function evaluating the quality of a solution, then the proposed solution x' is accepted. It depends on the quality difference between the new and the current solution and the current temperature. The lower the temperature and the worse the quality of the solution, the less likely it is that the new solution x' is accepted as the new current solution.

Reheating

The use of a non-monotonic cooling schedule was suggested by various authors, for example, by Osman [32], respectively already in one of his previous works together with Christofides [31]. After some time into the process of simulated annealing, it becomes more and more unlikely for worse feasible solutions or for infeasible solutions to be accepted as the new current solution as the probability of accepting non-improving solutions decreases with the temperature. Thereby, the optimization can get caught in a probabilistic local optimum with an acceptance rate of neighbors of virtually zero. Therefore, a natural idea is to reheat the system at some point in order to surpass such kinds of local optima.

Algorithm 3.3: Classical simulated annealing

```
input   : initial solution $x$
output  : best solution $x_b$
1  $t \leftarrow 0$;
2  $T \leftarrow T_{init}$;
3  $x \leftarrow$ *initial solution*;
4  $x_b \leftarrow x$;
5  while *stopping criteria not satisfied* do
6      while *equilibrium condition not satisfied* do
7          choose a $x'$ randomly from $N(x)$;
8          if $x'$ *is better than* $x$ then
9              $x \leftarrow x'$;
10             if $x < x_b$ then
11                 $x_b \leftarrow x$;
12             end
13         else
14             $P \leftarrow$ random number $\in [0, 1)$;
15             if $P < e^{-|f(x')-f(x)|/T}$ then
16                 $x \leftarrow x'$;
17             end
18         end
19         $t \leftarrow t + 1$;
20     end
21     $T \leftarrow g(T, t)$;
22 end
23 return $x_b$;
```

There are different possibilities on how and when to reheat the system. Since we may not want the high randomness of the initial temperature, often the temperature T_b where the best solution was found, is stored and the temperature is reset in relation to T_b; for example, the temperature might be set to $2 \cdot T_b$ or $T_b/2$. Furthermore, the time when to reheat can be determined dynamically based on a number of non-improving iterations, or statically, after a certain number of iterations, regardless of improving or not.

Strategic Oscillation

Strategic oscillation was suggested by Glover and Laguna [19] working with tabu search and also in previous works, for example, when Glover [18] was working with integer programming. It was initially used to oscillate between feasible and infeasible solutions. The way it is used by Anagnostopoulos et al. [1] is that the emphasis on feasibility increases and decreases over time, thereby, the time spent exploring the feasible regions and the time spent exploring infeasible regions of the search space should even out. This

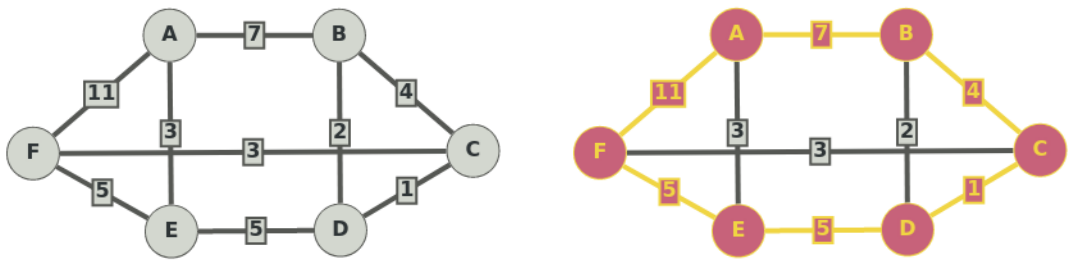

Figure 3.7: Left: A weighted graph G. Right: Graph G with a Hamiltonian cycle of length 33 ($ABCDEF$).

can be done, for example, by increasing and decreasing a weight which is used in the objective function to factor in violations regarding the feasibility. The weight could be decreased when accepting a feasible solution as the new current solution to decrease the emphasis on the violations and make it more likely to accept infeasible solutions in upcoming iterations. On the other hand, it would then be increased when accepting an infeasible solution to push the searched region back to feasibility.

Example

We introduce the Shortest Hamiltonian Cycle Problem (SHCP) in order to use it for the following SA example. It is a minimization problem where the goal is to find a Hamiltonian cycle in a weighted undirected graph with the smallest possible length where the length is the sum of the weights of all used edges. A Hamiltonian cycle C is a sequence of adjacent vertices, with also an edge between the last and the first, visiting every vertex of the graph exactly once. The objective function is calculated as follows:

$$obj(C) = \sum_{(i,j)\in C} w_{ij}. \tag{3.1}$$

In the above equation (3.1) the weights of the edges w_{ij} of an Hamiltonian cycle C are summed up.

We represent a feasible solution to this problem as a permutation of the vertices of the graph where between each pair of consecutive vertices and from the last to the first in the permutation there needs to be an edge in the graph. This problem is very similar to the TSP, however, the standard TSP is defined on complete graphs, therefore, all permutations of the vertices are feasible solutions to a TSP.

In Figure 3.7 (left), we can see a graph with edge weights. A feasible solution for this SHCP instance can be seen in 3.7 (right), which equals the permutation ($ABCDEF$) and has a length of 33. The best possible solution is shown in Figure 3.8 (left), which equals the permutation ($ABDCFE$). For example, an infeasible solution would be ($ACBDFE$).

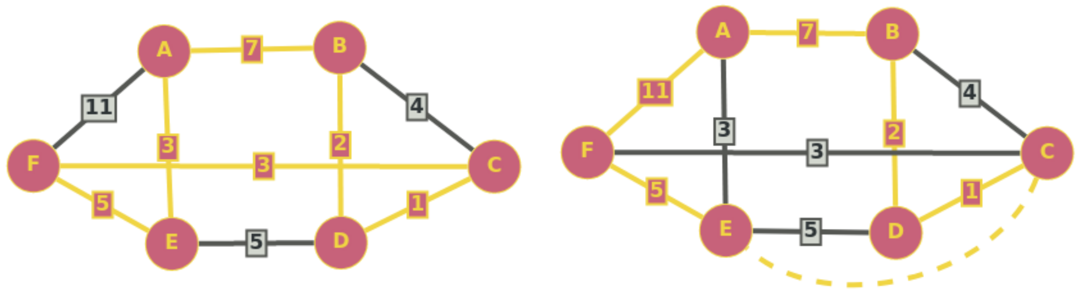

Figure 3.8: Left: Graph G with a Hamiltonian Cycle of length 21 ($ABDCFE$). Right: Neighbor of ($ABCDEF$) with a non-existing edge used and a Hamiltonian Cycle with cost 67.

A possible neighborhood structure for the SHCP could be to swap two consecutive vertices in the solution. For example, to go from ($ABCDEF$) to ($ABDCEF$). This might result in infeasible solutions as it is not guaranteed that there is an edge between B and D as well as between C and E (see Figure 3.8 (right)). However, for SA this is not a problem because SA was designed to be capable of handling infeasible regions. In a complete graph, this move will always result in a feasible solution.

We establish the cost function that combines the constraint violations in the form of the number of non-existing edges used with the actual objective function:

$$C(S) = M \cdot violations(S) + obj(S). \tag{3.2}$$

The M-value is designed so that every solution that uses a non-existing edge will be worse than any solution that only uses existing ones. A conservative choice for M, which we use in the following, is the sum over all edge weights in the whole graph. A tighter bound M-value would be to order the edges and take the sum over the n largest ones.

The M-value is multiplied by the number of violations of the solution, which is the number of used non-existing edges. The cost function will decide if a new solution is better or worse than the current one.

We also define the cooling schedule to be geometric by setting $g(T, t)$ to:

$$g(T, t) = 0.9 \cdot T. \tag{3.3}$$

We start the example run with the solution ($ABCDEF$) as shown in 3.7 as the initial solution x and currently best found x_b. We set the initial temperature to 100 and t to 0. For the stopping criteria we choose $T < 1$. Since this is not fulfilled, we enter the outer loop. For the equilibrium condition we define that $t+1$ needs to be divisible by 10, which means we will have ten iterations of the inner loop until we decrease the temperature. We then choose a random neighbor x' of x. For example, let x' be ($ABDCEF$), which uses a non-existing edge and therefore has a cost value of 67. This value is obviously worse than the 33 cost value of the current (and initial) solution x. However, there is still a chance that x' gets accepted as the new current solution x if a random number between 0 and 1 is less than $e^{-|f(x')-f(x)|/T}$, which means the random number needs to be less than 0.712. We assume that is the case and accept x' as the new current solution x'. Then t gets increased, however, the equilibrium criterion is not reached yet, therefore, we enter the inner loop again. We choose a random neighbor x' of x, for example, $(ABDCFE)$. In this case, the new solution x' has a cost value of 21 which is better than the 67 of the current solution, therefore, x' is accepted as the new solution. Moreover, x is now better than the best known solution x_b; x_b is therefore reset. Thereby, we found the best possible solution. The outer loop will now carry on until the stopping criterion is met. The best solution x_b is then returned in the end.

CHAPTER 4

Problem Description

In this chapter, we describe double round robin schedules and define the problem posed by the Fifth International Timetabling Competition (ITC2021). In particular, we describe its different constraints, its objective function, its computational complexity, and derived problem variants.

4.1 Double Round Robin Tournament

A Double Round Robin (DRR) schedule $T \in \mathcal{T}$ where $\mathcal{T}$ is the set of all DRRs with n teams defines which games are played in each round for a tournament of n teams $\{1, \ldots, n\}$ where n is an even number. In each round, also called slot, each team plays against another team, either at home or away. Obviously, if team t_1 plays at home against t_2 in round r, then t_2 plays against t_1 away in round r. Each team plays against each other team twice, once at home and once away. In Table 4.1, we can see a DRR schedule for six teams.

The schedule can be split into two parts as described by Rasmussen and Trick [37]: the timetable and the home-away pattern. A timetable is an assignment where each team has an opponent assigned to play against in each slot but with no home-away pattern defined. The home-away pattern is the assignment of a game between two teams to a venue of one of these two teams. The timetable of the DRR schedule shown in Table 4.1 can be seen in Table 4.2; its home-away pattern in Table 4.3.

4.2 ITC2021 Problem Definition

The goal of the problem defined by the Fifth International Timetabling Competition (ITC2021) [50] is to find a DRR schedule for a tournament defined by an instance file meeting all of the defined hard constraints while minimizing the penalization caused by

Table 4.1: DRR schedule for six teams. A minus sign in front of a team number indicates an away game.

team/round	1	2	3	4	5	6	7	8	9	10
1	-2	-5	6	3	-4	5	-3	2	4	-6
2	1	4	-3	-5	6	-5	5	-1	3	-4
3	-4	6	2	-1	-5	4	1	-6	-2	4
4	3	-2	5	-6	1	-3	6	-5	-1	2
5	-6	1	-4	2	3	-1	-2	4	6	-3
6	5	-3	-1	4	-2	2	-4	3	-5	1

Table 4.2: Timetable defined by the schedule of 4.1.

team/round	1	2	3	4	5	6	7	8	9	10
1	2	5	6	3	4	5	3	2	4	6
2	1	4	3	5	6	5	5	1	3	4
3	4	6	2	1	5	4	1	6	2	4
4	3	2	5	6	1	3	6	5	1	2
5	6	1	4	2	3	1	2	4	6	3
6	5	3	1	4	2	2	4	3	5	1

Table 4.3: Home-away pattern of the schedule of 4.1.

team/round	1	2	3	4	5	6	7	8	9	10
1	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$
2	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$
3	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$
4	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$
5	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{H}$	$\mathcal{A}$
6	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$	$\mathcal{A}$	$\mathcal{H}$

the violation of soft constraints—it is therefore a constraint optimization problem. An instance is defined by an XML file in the RobinX data format although only a subset of the constraints proposed by Van Bulck et al. [49] is used. In the XML file, the number of teams and the constraints are described. Moreover, an instance may also be defined to be *phased*, which means that the DRR schedule splits up into two Single Round Robin (SRR) schedules in which each pair of teams plays against each other once in the first SRR schedule and once in the second. The home-away status of games in the first phase changes accordingly in the second phase. The example shown in Table 4.1 is a phased schedule. There are infinitely many instances since one can define new instances via an XML file. However, the competition came with 54 instances that were to be solved during the competition; they are further described in Section 6.2.

4.3 Constraints

All of the constraints can either appear as a hard or a soft constraint. Hard constraints need to be fulfilled for a schedule to be feasible. The sum of soft constraint violations constitutes the objective function. A violated soft constraint contributes its penalty factor multiplied by the deviation of the allowed range to the objective value.

Capacity constraints impose restrictions on *where* teams are allowed to play in certain time slots. There are four different capacity constraints. The first one CA1 defines the maximum number of home or away games (depending on the mode of the constraint) by a certain team in given slots; CA2 defines the maximum number of home/away/games in general against all given opponents in the given slots. Thereby, CA2 generalizes CA1. The third capacity constraint CA3 defines the maximum number of consecutive home/away/games in general against all given opponents for a given team. The last of the capacity constraints CA4 defines the maximum number of home/away/games in general of all teams defined by the constraints against a second group of teams either in each defined slot or over all of the slots (depending on the mode of the constraint).

The game constraint GA1 enforces or forbids certain games in certain time slots by defining the maximum and minimum number of games from a defined game pool that have to be played in the given time slots. A maximum number of 0 forbids games; a minimum other than 0 enforces games.

The break constraints manage the amount and the timing of breaks in the schedule. A break occurs when a team plays a game with the same home-away status as its previous game. Therefore, a home (away) break is when a team plays two home (away) games consecutively. For example, in $\mathcal{AHHAAA}$ there are three breaks: one home break and two away breaks. The first break constraint BR1 defines the maximum number of home/away/breaks in general of a team during the defined time slots. The other break constraint BR2 defines a maximum number of breaks that all teams defined by the constraint combined have in the schedule.

The fairness constraint FA2 defines the maximum value by which the number of home games played by each given team differs from the home games played by all other given teams in the given time slots. The other possible modes of FA2 (away games or all games) are not considered by ITC2021.

The separation constraint SE1 defines the minimum number of games between mutual games of all pairs of the given teams. A mutual game of two teams is when they play against each other.

In Listing 4.1 all types of constraints are shown, the examples are taken from the test instances of the ITC2021. The CA1 constraint states that team 0 plays at most one home game in slot 9, 6 and 7. The CA2 constraint says that team 4 plays at most two games against a team from $\{0, 3, 2\}$ in slots $\{2, 6, 1, 0\}$. The third constraint defines that team 2 plays at most two away games in each span of four slots against teams from $\{0, 3, 5, 4, 1\}$. The CA4 constraint states that the teams $\{4, 3, 1, 2\}$ play at most four home games against each other in the slots $\{2, 3, 1, 0\}$. The game constraint GA1 defines that at most three games of the given games (called meetings in the constraint) take place in slots $\{2, 3, 8\}$. The BR1 constraint says that team 3 has at most two breaks in slots $\{3, 4, 5, 6, 8, 9\}$. The BR2 constraint defines that the sum of all breaks of the teams $\{2, 0, 1, 4, 3, 5\}$ in the given slots is no more than 18. The fairness constraint FA2 states that the difference of games played at games between two teams from $\{2, 0, 1, 4, 3, 5\}$ is no more than 2 in the given slots. Last, the separation constraint SE1 says that there has to be at least ten games between mutual games of each pair of the teams $\{2, 0, 1, 4, 3, 5\}$.

Listing 4.1: Constraint types of ITC2021

```
<CA1 max="1" min="0" mode="H" penalty="5" slots="9;6;7" teams="0"
     type="HARD"/>
<CA2 max="2" min="0" mode1="HA" mode2="GLOBAL" penalty="1"
     slots="2;6;1;0" teams1="4" teams2="0;3;2" type="HARD"/>
<CA3 intp="4" max="2" min="0" mode1="A" mode2="SLOTS" penalty="5"
     teams1="2" teams2="0;3;5;4;1" type="SOFT"/>
<CA4 max="4" min="0" mode1="H" mode2="GLOBAL" penalty="1"
     slots="2;3;1;0" teams1="4;3;1;2" teams2="4;3;1;2" type="HARD"/>

<GA1 max="3" meetings="0,3;2,3;4,3;" min="1" penalty="1"
     slots="2;3;8" type="HARD"/>

<BR1 intp="2" mode1="LEQ" mode2="HA" penalty="5" slots="3;4;5;6;8;9"
     teams="3" type="SOFT"/>
<BR2 intp="18" homeMode="HA" mode2="LEQ" penalty="1"
     slots="1;2;3;4;5;6;7;8;9;0" teams="2;0;1;4;3;5" type="HARD"/>

<FA2 intp="2" mode="H" penalty="10" slots="1;2;3;4;5;6;7;8;9;0"
     teams="2;0;1;4;3;5" type="SOFT"/>

<SE1 mode1="SLOTS" min="10" penalty="10" teams="2;0;1;4;3;5"
     type="SOFT"/>
```

4.4 Objective Function

Given a schedule T that fulfills all the hard constraints $c' \in \mathcal{C}_{\text{hard}}$, each element c of the set of soft constraints $\mathcal{C}_{\text{soft}}$ triggers a penalty $p_c(T)$, where p_c is the penalty function $p_c \colon T \to \mathbb{R}_0^+$ for the constraint c and its value is defined by multiplying the penalization factor λ_c with the deviation to the allowed range $d_c(T)$, thereby $p_c(T) = \lambda_c \cdot d_c(T)$. For example, regarding the first constraint in Listing 4.1, if team 0 plays a home game in each of the 3 slots ($\{9, 6, 7\}$), the constraint triggers a penalty of $5 \cdot 2 = 10$ because there are played two games more than allowed; if there is no violation, d_c will be 0, therefore, the penalty equals 0. The objective function then sums up all these penalties triggered by the soft constraint violations of a schedule T

$$f(T) = \sum_{c \in C_{soft}} p_c(T). \tag{4.1}$$

4.5 $\mathcal{NP}$-Hardness

Conjecture 1 *The ITC2021 problem is $\mathcal{NP}$-hard.*

The break minimization problem takes a given timetable for n teams as its input, where n is an even number. The goal is to find a home-away pattern with a minimum number of breaks. A break is when a team plays back-to-back home or away games. Elf et al. [11] conjectured that the break minimization problem is $\mathcal{NP}$-hard. They showed that the break minimization problem can be reduced to the max-cut problem, which is known to be $\mathcal{NP}$-complete, however, they did not show a reduction from an $\mathcal{NP}$-hard problem to break minimization. Since every break minimization problem instance can be reduced to an ITC2021 problem instance in polynomial time, the $\mathcal{NP}$-hardness of break minimization would imply $\mathcal{NP}$-hardness of the ITC2021 problem. The reduction could be as follows:

- The given schedule needs to be modeled with hard GA1 constraints, to define for each team against which other team it has to play according to the given schedule by setting for each slot exactly one game with min/max=1. The ITC2021 deals only with double round robin tournaments, however, we can just ignore the second part of the schedule, it will be filled up by a feasible schedule without any further constraints considered.
- For the break minimization part, we need a soft BR2 constraint over all slots of the first half of the schedule and all teams that allows for 0 breaks, thereby, the used breaks are summed up by the objective function.

  Since the objective function of the break-minimization-ITC2021 sums up all breaks that occur in the solution (without applying any further constraints), the objective value is equal to the objective value of the break minimization problem. The second part of the schedule can be ignored since the completion of the schedule to a DRR does not add to the objective value because only the first part of the schedule is considered for the BR2 constraint.

4.6 Derived Problems

We define the following derived problem variants where certain constraints are left out to focus on some aspects of the problem. We call the problem without any soft constraints ITC2021-NSC, the problem without break constraints ITC2021-NBC and if both are left out ITC2021-NSC-NBC.

CHAPTER 5

Solution Approaches

In this chapter, we present our solution approaches to the ITC2021. In Section 5.1, we describe the modeling language MiniZinc and the translation of ITC2021 instances into MiniZinc models. In Section 5.2, we describe our metaheuristic approach, an SA algorithm based on the TTSA by [1], and the neighborhoods used for it. In Section 5.3, we describe our approach to combine the exact method CP with the metaheuristic SA, in the form of a parallel multi-start hybrid.

5.1 Constraint Programming

For the exact part of our approach, we implement a compiler that translates the instance XML files into MiniZinc models.

MiniZinc was introduced by Nethercote et al. [30], who tried to tackle the problem that there are many different CP solvers where most of them have their own modeling language. This circumstance prevented practitioners from quickly testing their models with different solvers, therefore, they came up with MiniZinc as a possible standard modeling language and FlatZinc as the link between MiniZinc and a CP solver. FlatZinc is a low-level solver input language for which CP solvers need to provide an interface. It is also the target language for MiniZinc.

The following example shows what the example CSP from 3.1.3 looks like when modeled with MiniZinc (taken from the MiniZinc Handbook of Stuckey et al. [41, 2.5]). The first line defines a parameter which is similar to constant variables in most programming languages. They can only be assigned once, however, it is also possible to declare the value of a parameter in a separate data file.

Decision variables are another type of variable that can be defined inside a MiniZinc model. Decision variables are not like variables in programming languages but rather like variables in mathematics. They are not assigned a value by the modeler; instead, the value of the variable is unknown until the model is executed and a solver determines if the decision variable can be assigned a feasible value.

Constraints specify boolean expressions that must hold by solution candidates to be valid. A solution candidate consists of assignments to all decision variables. The solve statement defines what kind of solution should be searched. The solve mode *satisfy* simply searches for a solution that satisfies all of the given constraints, *minimize* and *maximize* search for an optimum solution regarding the expression they are followed. One can also specify annotations to define a variable ordering and a value ordering inside the domain of a variable, for example, `int_search(q, input_order, indomain_min)` where q needs to be an array of integers. Moreover, one could also specify a restart annotation to define what type of restarting should be used. All of these annotations can also be assigned to a parameter which then can be used in the solve statement.

The `include` statement makes the contents of another file (here `alldifferent.mzn`) available in the current file. Comments start with a percent sign (%) and the output statement allows to print out the values of variables and strings.

Listing 5.1: n-queens in MiniZinc taken from the MiniZinc Handbook [41]

```
int: n = 4;
array [1..n] of var 1..n: q; % queen in column i is in row q[i]

include "alldifferent.mzn";

constraint alldifferent(q);                       % distinct rows
constraint alldifferent([ q[i] + i | i in 1..n]); % distinct diagonals
constraint alldifferent([ q[i] - i | i in 1..n]); % upwards+downwards

% search
solve satisfy;
output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

In Listing 5.1, n defines the number of rows, columns and queens to be placed on the chessboard. The decision variable q is an array where each of the indices represents a column and the values at these indices represent the rows; the queen of column i is placed in row $q[i]$. The `alldifferent` statement defines that every value of the provided array must be different. The first `alldifferent` ensures that there is only one queen per row, the second one ensures that there is only one queen per ascending diagonal (going left to right), and the third one ensures that there is only one queen on each of the descending diagonals. The solve statement defines that there is no objective function to optimize and the output statement prints feasible queen placements.

In Figure 5.1, we can see the two solutions that exist for $n = 4$, however, obviously, the second solution is only the first one rotated by 90 degrees because there is only one distinct solution for that problem instance (e.g. for $n = 8$ there are 12 distinct solutions).

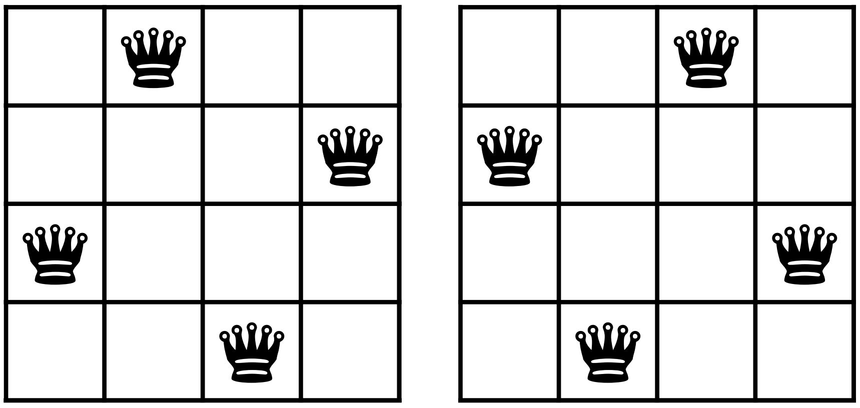
Figure 5.1: Two solutions for $n = 4$

5.1.1 Translation to MiniZinc

In this subsection, we present how the instance files of the ITC problem are processed and translated into MiniZinc models.

We first look at the core structure of our generated models, which is the same for all instances and can be seen in Listing 5.2. The number of teams is processed by the translator and defines the parameter N. The number of slots is also processed and defines the parameter R where R2 is half the value of R. Afterwards, we declared annotations that are used in the solve statement at the end of the model. They are assigned values by data files (or via the command line operator -D). Then four sets are initialized: TEAMS represents all teams playing in the tournament of the given instance, SLOTS represents all rounds in which games are played, SLOTS1 are the first R2 rounds and SLOTS2 the second R2 rounds.

Then the decision variables are defined: *opponents_per_team_and_round* is a two dimensional array that represents which team plays against which in each round, $venue\_per_team\_and\_round$ defines at which venue each team plays in each round (1 for home games, -1 for away games).

Below the decision variables, the double round robin constraints are introduced: the first one defines that a team may not play against itself; the second one defines that playing against each other is symmetric, so if a team t_1 plays against a team t_2 in round r, then team t_2 also plays against team t_1 in round r. The third constraint defines that the venues need to be the opposite for each pair of teams playing against each other. The fourth constraint states that each pair of teams needs to play once in the first half and once in the second half of the schedule against each other—this constraint is active if the instance to solve is a phased instance; if it is not, the constraint would go over all SLOTS and require the sum to be 2. The last one defines that each pair of teams has to play against each other once at home and once away (the sum of the venues is thereby 0).

The DRR constraints are followed by the ITC specific hard and soft constraints, which will be discussed one by one below this example as well as the objective function which consists of the penalization terms of the soft constraints. With the *solve* statement, we define how the search process should be run and that objective *obj* is to be minimized; as parameters for the search annotation *int search* the previously defined annotations *varchoice* and *constraintchoice* are used. We enable the restarting functionality of the search by the previously defined annotation *restart*. In the end, the description of how the schedule should be printed can be found.

Listing 5.2: Core structure of a model to solve an ITC2021 problem instance

```
int: N = 4;
int: R = 6;
int: R2 = 3;
ann: varchoice;
ann: constraintchoice;
ann: restart;
set of int: TEAMS = 1..N;
set of int: SLOTS = 1..R;
set of int: SLOTS1 = 1..R2;
set of int: SLOTS2 = (R2 + 1)..R;

array[TEAMS,SLOTS] of var TEAMS: opponents_per_team_and_round;
array[TEAMS,SLOTS] of var {-1, 1}: venue_per_team_and_round;


% team does not play against itself
constraint
  forall (i in TEAMS, r in SLOTS) (
    opponents_per_team_and_round[i, r] != i
  );

% all games (opponent+venue) per team must be unique
constraint
  forall (i in TEAMS, j in TEAMS where i != j) (
    sum(r in SLOTS where opponents_per_team_and_round[i,r] == j)
       (venue_per_team_and_round[i,r]) == 0
  );

% opponent connection constraint
constraint
  forall (i in TEAMS, r in SLOTS) (
    opponents_per_team_and_round[opponents_per_team_and_round[i,r],r] == i
  );
% venue connection constraint
constraint
  forall (i in TEAMS, r in SLOTS) (
    venue_per_team_and_round[opponents_per_team_and_round[i,r],r] ==
      -venue_per_team_and_round[i,r]
  );

% each team must play exactly once against each other team in
% each phase (SRR) of the schedule
constraint
```

```
  forall (i in TEAMS, j in TEAMS where i != j) (
    sum(r in SLOTS1 where opponents_per_team_and_round[i,r] == j)(1) == 1
    /\
    sum( r in SLOTS2 where opponents_per_team_and_round[i,r] == j)(1) == 1
  );
...
% CONSTRAINTS
...
% solving
solve
:: int_search(opponents_per_team_and_round, varchoice, constraintchoice)
:: restart
minimize obj;

output [ format(3, 2, opponents_per_team_and_round[i,j] - 1) ++
      if j == R then "\n" else " " endif |
      i in TEAMS, j in SLOTS
];
output ["\n"];
output [ format(3, 2, venue_per_team_and_round[i,j]) ++
      if j == R then "\n" else " " endif |
      i in TEAMS, j in SLOTS
];
output ["objective = \(obj);"];
```

Subsequently, each translation of the possible constraints of ITC instances as discussed in Section 4.3 is described in detail. Since MiniZinc uses one-based array indices, the team numbers are all increased by 1 as compared to the constraint. For the output, we subtract 1 from each team index as seen in the basic structure above. In each listing, we first show the constraint in XML and afterwards present the constraint modeled as hard and as soft constraint in MiniZinc.

The given CA1 constraint in Listing 5.3 states that team 0 plays at most one home game in the given slots (6,7,9). The hard constraint models that by ensuring the number of games played at home is less than or equal to 1. We use sum(x where cond)(1) to count occurrences of the condition cond. If the mode was A, the condition inside the sum would be where venue_per_team_and_round[1,slot] == −1 to count the number of away games. The soft constraint uses the same technique to calculate how many more games are played and then calculate the penalization term that is summed up in the objective function. The maximum function is used because otherwise we could get negative penalization terms. If, for example, two games are allowed and there is only one played, the sum would be 1 and we would subtract 2, resulting in a negative value which would reduce the objective value.

Listing 5.3: Example of a CA1 constraint

```
<CA1 max="1" min="0" mode="H" penalty="5" slots="9;6;7" teams="0">

% hard constraint %
constraint sum(slot in {10,7,8}
                where venue_per_team_and_round[1,slot] == 1)(1) <= 1;
var int: penalty_ca1_1_6_7_9_H =
    5 * (max(sum(slot in {10,7,8}
              where venue_per_team_and_round[1,slot] == 1)(1) - 1, 0));
```

The given CA2 constraint in Listing 5.4 states that team 4 plays at most two games against one of the given teams (0,2,3) in the given slots (0,1,2,6). To translate this constraint into Minizinc, we need to look at the *opponents per team and round* since it matters against whom the restricted team plays (in contrast to CA1). We count the occurrences of the given other teams in the given slots and ensure the sum to be less than or equal to 2. If the mode was not HA, the condition of the sum would be expanded by /\ venue_per_team_and_round[5,slot] == −1 for A and with a 1 for H since then it would also matter where those games are played. The soft constraint works similar to the CA1 soft constraint; it uses the same sum as the hard constraint and subtracts the target value to calculate the number of excess games and use that value to calculate the penalization term.

Listing 5.4: Example of a CA2 constraint

```
<CA2 max="2" min="0" mode1="HA" mode2="GLOBAL" penalty="1"
     slots="2;6;1;0" teams1="4" teams2="0;3;2"/>

% hard constraint %
constraint sum (slot in {3,7,2,1}, opponent in {1,4,3}
    where opponents_per_team_and_round[5,slot] == opponent ) (1) <= 2;

% soft constraint %
var int: penalty_ca2_1_0_2_3 =
    1 * (max(sum (slot in {3,7,2,1}, opponent in {1,4,3}
    where opponents_per_team_and_round[5,slot] == opponent ) (1) - 2 , 0));
```

The given CA3 constraint in Listing 5.5 defines that team 2 plays at most two away games in each span of four slots against the given teams (0,1,3,4,5). The hard constraint ensures this by going through the possible start slots of a span (the last possible start slot is SLOTS − *intp*, where *intp* is the length of the span), counting the occurrences of the games in question, and checking that the number of occurrences is less than or equal to 2. The soft constraint sums up all additional away games that are played and multiplies that value by the penalization factor.

Listing 5.5: Example of a CA3 constraint

```
<CA3 intp="4" max="2" min="0" mode1="A" mode2="SLOTS" penalty="5"
    teams1="2" teams2="0;3;5;4;1"/>

% hard constraint %
constraint
  forall (team1 in {3})(
    forall (j in 1..7)(
      sum (slot in j..(j+3), team2 in {1,4,6,5,2}
          where opponents_per_team_and_round[team1,slot] == team2
```

```
            /\ venue_per_team_and_round[team1,slot] == -1) (1) <= 2
      )
   );

% soft constraint %
var int: penalty_ca3_1_2 = 5 *
   sum(team1 in {3}, j in 1..7) (max(
      sum (
         slot in j..(j+3), team2 in {1,4,6,5,2} where
            opponents_per_team_and_round[team1,slot] == team2
            /\ venue_per_team_and_round[team1,slot] == -1
      ) (1) - 2, 0
   ));
```

The given CA4 in Listing 5.6 states that the given teams *teams*1 (1,2,3,4) play at most four home games against the other teams *teams*2 (1,2,3,4) in the given slots. In this case the two team variables are the same, but they can be different. If the mode is *GLOBAL*, then the sum of played games is calculated over all given slots; if it is *EVERY*, the sum is calculated for each slot.

For the hard constraint, we sum up the games in question and ensure that the number of games is less than or equal to 4. The soft constraint also counts the number of respective games and calculates the penalization term to use in the objective function. If the mode was *EVERY* instead of *GLOBAL*, the two constraints on the bottom of the listing would be the result. The hard constraint ensures that for each slot the sum of the games played is below or equal to 4. The soft constraint calculates how many more games are played and multiplies the result by the penalization factor.

Listing 5.6: Example of a CA4 constraint

```
<CA4 max="4" min="0" mode1="H" mode2="GLOBAL/EVERY" penalty="1"
      slots="2;3;1;0" teams1="4;3;1;2" teams2="4;3;1;2"/>

% GLOBAL
% hard constraint %
constraint
   sum(team1 in {5,4,2,3}, team2 in {5,4,2,3}, slot in {3,4,2,1}
      where opponents_per_team_and_round[team1,slot] == team2
      /\ venue_per_team_and_round[team1,slot] == 1) (1) <= 4;

% soft constraint %
var int: penalty_ca4_global_1_0_1_2_3 =
   1 *  max(sum(team1 in {5,4,2,3}, team2 in {5,4,2,3}, slot in {3,4,2,1}
         where opponents_per_team_and_round[team1, slot] == team2
         /\ venue_per_team_and_round[team1,slot] == 1) (1) -4, 0);

% EVERY
% hard constraint %
constraint
   forall(slot in {3,4,2,1})(
      sum(team1 in {5,4,2,3}, team2 in {5,4,2,3}
      where opponents_per_team_and_round[team1,slot] == team2
      /\ venue_per_team_and_round[team1,slot] == 1) (1) <= 4
   );

% soft constraint %
var int: penalty_ca4_every_1_0_1_2_3 =
    1 * sum(slot in {3,4,2,1})
      (max(sum(team1 in {5,4,2,3}, team2 in {5,4,2,3}
      where opponents_per_team_and_round[team1,slot] == team2
      /\ venue_per_team_and_round[team1,slot] == 1) (1) - 4, 0 ));
```

The given GA1 in Listing 5.7 defines that at least one and at most three games from the given games (0@3, 2@3, 4@3) are played in the given slots (2, 3, 8). We first define a variable that sums up all the games played of the games in question and the constraint then defines the allowed range. The sum variable is also used in case of a soft constraint. Here we have two penalization terms: one for the case that more than the allowed games are played and one for the case that fewer are played.

Listing 5.7: Example of a GA1 constraint

```
<GA1 max="3" meetings="0,3;2,3;4,3;" min="1" penalty="1"
      slots="2;3;8"/>

var int: sum_ga1_1 =
  sum(slot in {3,4,9} where (opponents_per_team_and_round[1, slot] == 4
    /\ venue_per_team_and_round[1,slot] == 1))(1) +
  sum(slot in {3,4,9} where (opponents_per_team_and_round[3, slot] == 4
    /\ venue_per_team_and_round[3,slot] == 1))(1) +
  sum(slot in {3,4,9} where (opponents_per_team_and_round[5, slot] == 4
    /\ venue_per_team_and_round[5,slot] == 1))(1);

% hard constraint %
constraint sum_ga1_1 >= 1 /\ sum_ga1_1 <= 3;

% soft constraint %
var int: penalty_ga1_max_2_2_3_8 = 1 * max(sum_ga1_1 - 3, 0);
var int: penalty_ga1_min_2_2_3_8 = 1 * max(1 - sum_ga1_1, 0);
```

The given BR1 constraint in Listing 5.8 states that team 3 has at most two breaks in the given slots (3, 4, 5, 6, 8, 9). First, the number of breaks is calculated by adding up those times when the team played the same home-away-status back-to-back. Then the hard constraint defines that the sum has to be less than or equal to 2; the soft constraint calculates the amount of breaks that are above the allowed value and multiplies that value with the penalization factor. If the mode was not HA, the condition of the sum would have another part /\ venue_per_team_and_round[4,slot] == 1 for home breaks and the same with -1 at the end for away breaks.

Listing 5.8: Example of a BR1 constraint

```
<BR1 intp="2" mode1="LEQ" mode2="HA" penalty="5" slots="3;4;5;6;8;9"
      teams="3"/>
```

```
var int: sum_br1_1 =
    sum (slot in {4,5,6,7,9,10} where slot != 1  /\
    venue_per_team_and_round[4,slot -1] == venue_per_team_and_round[4,slot])(1);

% hard constraint %
constraint sum_br1_1 <= 2;

% soft constraint %
var int: penalty_br1_2_3_4_5_6_8_9 = 5 * max(sum_br1_1 - 2, 0);
```

The given BR2 constraint in Listing 5.9 defines that the sum of all breaks of the given teams (0,1,2,3,4,5) in the given slots (0,1,2,3,4,5,6,7,8,9) is not more than 18. The sum of all breaks (of the given teams and slots) is summed up in the variable and then the variable gets restricted by the hard constraint to be less than or equal to 18. The soft constraint uses that variable to calculate the penalization term, by summing up how many more breaks were in the schedule and multiplying that value with the penalization factor.

Listing 5.9: Example of a BR2 constraint

```
<BR2 intp="18" homeMode="HA" mode2="LEQ" penalty="1"
     slots="1;2;3;4;5;6;7;8;9;0" teams="2;0;1;4;3;5"/>

var int: sum_br2_1 =
    sum(slot in {2,3,4,5,6,7,8,9,10,1}, team in {3,1,2,5,4,6}
    where slot != 1 /\
    venue_per_team_and_round[team,slot -1] ==
        venue_per_team_and_round[team,slot])(1);

% hard constraint %
constraint sum_br2_1 <= 18;

% soft constraint %
var int: penalty_br2_2_0_1_2_3_4_5_6_7_8_9 = 1 * max(sum_br2_1 - 18, 0);
```

The given FA2 constraint in Listing 5.10 (the slots in the example from 4.3 were reduced to make the constraints smaller) states that the difference of games played at home between each pair of teams of the given teams (0, 1, 2, 3, 4, 5) may not be more than two in the given slots (0, 1, 2, 3, 4). The hard constraint sums up the number of home games each team has played up to each of the given slots for each pair of teams and defines that the difference of those two values has to be less than or equal to 2. For the soft constraint there are a lot of variables: each team pair has its own variable (we only show the variable for the teams 2 and 0), in which the maximum difference in all of the given slots is calculated and used to calculate the penalization term.

Listing 5.10: Example of a FA2 constraint

```
<FA2 intp="2" mode="H" penalty="10" slots="1;2;3;4;0" teams="2;0;1;4;3;5"/>

% hard constraint %
constraint
```

```
    forall(team1 in {3,1,2,5,4,6}, team2 in {3,1,2,5,4,6}
           where team1 != team2)(
      forall (slotTo in {2,3,4,5,1})(
        abs(sum(slot in 1..slotTo
                where venue_per_team_and_round[team1,slot] == 1)(1)-
        sum(slot in 1..slotTo
                where venue_per_team_and_round[team2,slot] == 1)(1) ) <= 2
      )
    );

% soft constraint %
var int: penalty_fa2_1_2_0 = 10 * max(max([
abs(sum(slot in 1..2 where venue_per_team_and_round[3,slot] == 1)(1) -
    sum(slot in 1..2 where venue_per_team_and_round[1,slot] == 1)(1)),
abs(sum(slot in 1..3 where venue_per_team_and_round[3,slot] == 1)(1) -
    sum(slot in 1..3 where venue_per_team_and_round[1,slot] == 1)(1)),
abs(sum(slot in 1..4 where venue_per_team_and_round[3,slot] == 1)(1) -
    sum(slot in 1..4 where venue_per_team_and_round[1,slot] == 1)(1)),
abs(sum(slot in 1..5 where venue_per_team_and_round[3,slot] == 1)(1) -
    sum(slot in 1..5 where venue_per_team_and_round[1,slot] == 1)(1)),
abs(sum(slot in 1..1 where venue_per_team_and_round[3,slot] == 1)(1) -
    sum(slot in 1..1 where venue_per_team_and_round[1,slot] == 1)(1))]) - 2,
    0);
```

The given SE1 constraint in Listing 5.11 states that there need to be at least ten games between mutual games of each pair of the given teams (0, 1, 2, 3, 4, 5). There are *min* number of hard constraints (we only show the first one). In each of the constraints the value that is added to r increases, starting with 1 as it can be seen in the listing below. This constraint checks that in consecutive slots there is not the same game played. The next constraint checks for slots that are two apart, the next for three apart, etc. For the soft constraint only the first penalization term is shown; there is again a variable for each pair of teams. The penalization term is generated by checking for each game span value i (1 up to *min*) if the two teams of that variable play against each other in two rounds that are i apart. If that is the case, the sum will return the value of how many games shy the separation was to *min*, the *max* function call will then return that sum because only one of these sum calls can return a value other than 0 as there are only two games in the schedule, e.g., they cannot be 3 and 7 apart.

Listing 5.11: Example of a SE1 constraint

```
<SE1 mode1="SLOTS" min="10" penalty="10" teams="2;0;1;4;3;5"/>

% hard constraint %
constraint
    forall(team1 in {3,1,2,5,4,6}, team2 in {3,1,2,5,4,6},
          r in 1..R-1 where team1 < team2)
      (opponents_per_team_and_round[team1, r] != team2 \/
       opponents_per_team_and_round[team1, r + 1] != team2);

% soft constraint %
var int: penalty_se1_1_3_5 = 10 * max([
sum(r in 1..R−1 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 1] == 5)(10 − (1 − 1)),
sum(r in 1..R−2 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 2] == 5)(10 − (2 − 1)),
sum(r in 1..R−3 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 3] == 5)(10 − (3 − 1)),
sum(r in 1..R−4 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 4] == 5)(10 − (4 − 1)),
sum(r in 1..R−5 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 5] == 5)(10 − (5 − 1)),
sum(r in 1..R−6 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 6] == 5)(10 − (6 − 1)),
sum(r in 1..R−7 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 7] == 5)(10 − (7 − 1)),
sum(r in 1..R−8 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 8] == 5)(10 − (8 − 1)),
sum(r in 1..R−9 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 9] == 5)(10 − (9 − 1)),
sum(r in 1..R−10 where opponents_per_team_and_round[3, r] == 5 /\
    opponents_per_team_and_round[3, r + 10] == 5)(10 − (10 − 1))]);
```

5.1.2 Strengthening Constraint Improvement

MiniZinc allows to profile the search for a solution and show the result of that profiling in a search tree [41, 3.4] (e.g. those in Figure 5.2 and 5.3). This can be used to improve constraints or add constraints that allow to prune a branch earlier. For example, in Figure 5.2 a search tree is displayed where the corresponding search did not have the first constraint seen in the basic structure above in Listing 5.2 that a team cannot play against itself. This constraint is implied by another constraint which says that each pair of the n distinct teams has to play against each other twice within $2(n-1)$ rounds, with only one game per team and round possible. Branch nodes are shown as blue circles, nodes where the remaining subproblem is unsatisfiable are shown as red squares and green diamonds represent solutions nodes. Whenever a subtree contains only failures, it is shown as a red triangle.

In Figure 5.2, we can see the search with Gecode as the solver, *input_order* as the variable choice heuristics and *indomain_min* as the value choice heuristics. The search had a lot of failures because the first try in assigning the first team to play against a team was always the first team itself. However, what happens if we add a constraint that teams may not play against themselves can be seen on the right of Figure 5.3. The search tree is smaller because the wrong branches in which a team plays against itself can be omitted and the domain to choose values from becomes smaller.

The labels of the search trees show the assigned values of the flattened *opponents per team and round* and *venue per team and round*, however, internally the two variables are replaced with variables for each position of the array. Unfortunately, the mapping of *venue per team and round* did not work in the case of those figures.

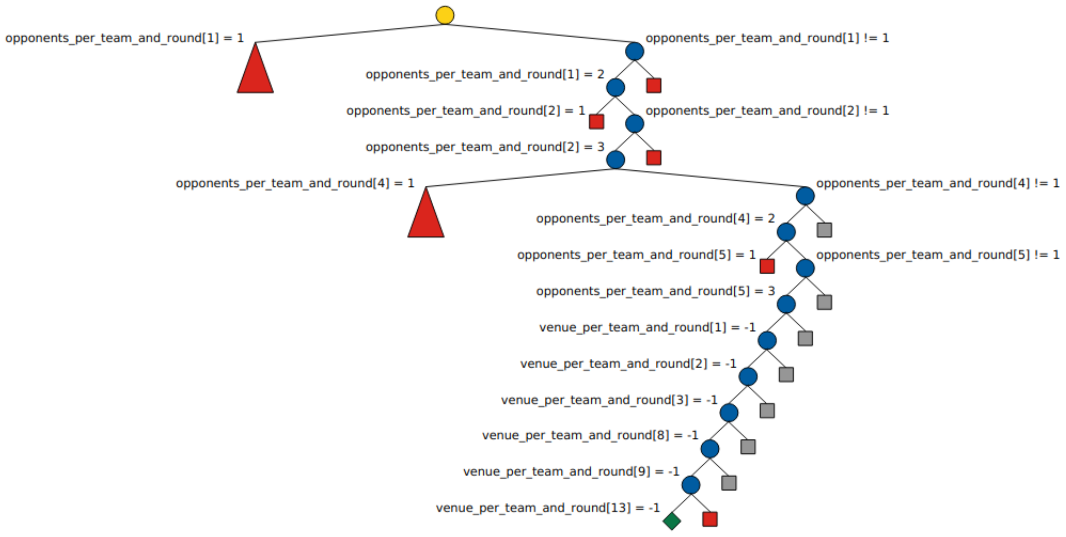
opponents_per_team_and_round[1] = 1
opponents_per_team_and_round[1] != 1
opponents_per_team_and_round[1] = 2
opponents_per_team_and_round[2] = 1
opponents_per_team_and_round[2] != 1
opponents_per_team_and_round[2] = 3
opponents_per_team_and_round[4] = 1
opponents_per_team_and_round[4] != 1
opponents_per_team_and_round[4] = 2
opponents_per_team_and_round[5] = 1
opponents_per_team_and_round[5] != 1
opponents_per_team_and_round[5] = 3
venue_per_team_and_round[1] = -1
venue_per_team_and_round[2] = -1
venue_per_team_and_round[3] = -1
venue_per_team_and_round[8] = -1
venue_per_team_and_round[9] = -1
venue_per_team_and_round[13] = -1

Figure 5.2: Search tree for TestInstanceDemo (4 teams) before adding a constraint that forbids teams to play against themselves.

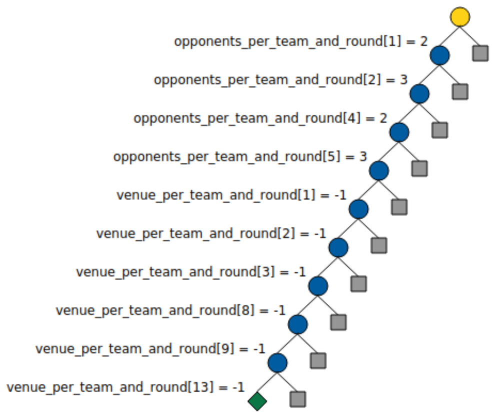
opponents_per_team_and_round[1] = 2
opponents_per_team_and_round[2] = 3
opponents_per_team_and_round[4] = 2
opponents_per_team_and_round[5] = 3
venue_per_team_and_round[1] = -1
venue_per_team_and_round[2] = -1
venue_per_team_and_round[3] = -1
venue_per_team_and_round[8] = -1
venue_per_team_and_round[9] = -1
venue_per_team_and_round[13] = -1

Figure 5.3: Search tree for TestInstanceDemo after adding that constraint.

5.2 Simulated Annealing

For the heuristic part of the approach, we implement a simulated annealing based on the work of Anagnostopoulos et al. [1]. They demonstrate a successful simulated annealing approach for the TTP, which they call TTSA. In addition to the classical simulated annealing, they also make use of reheating and strategic oscillation.

The sligthly adapted algorithm is shown in Algorithm 5.1. We add the option to minimize violations, which enables the algorithm to always accept a solution that has fewer violations than any other previously found solution even if its objective value is worse than the current one.

Algorithm 5.1: Slightly altered TTSA from Anagnostopoulos et al. [1]

```
input  : initial schedule $S$
output : objective value of best schedule
1  $bestFeasible \leftarrow \infty; nbf \leftarrow \infty$;
2  $bestInfeasible \leftarrow \infty; nbi \leftarrow \infty$;
3  $reheat \leftarrow 0$;
4  $iteration\_counter \leftarrow 0$;
5  $acceptance\_counter \leftarrow 0$;
6  $acceptance\_queue \leftarrow []$;
7  while $reheat \leq maxR$ do
8      $phase \leftarrow 0$;
9      while $phase \leq maxP$ do
10         $counter \leftarrow 0$;
11         while $counter \leq maxC$ do
12             monitor acceptance rate;
13             select a random move $m$ from $neighborhood(S)$;
14             let $S'$ be the schedule obtained from $S$ with $m$;
15             if $C(S') < C(S)$ OR $nbv(S') == 0$ and $C(S') < bestFeasible$ OR
                $nbv(S') > 0$ and $C(S') < bestInfeasible$ OR $\boldsymbol{lowestViol > 0}$ and
                $\boldsymbol{nbv(S') < lowestViol}$ then
16                 $accept \leftarrow$ true;
17             else
18                 $accept \leftarrow$ true with probability $\exp(-\Delta C/T)$ false otherwise;
19             end
20             acceptance handling;
21             $counter \leftarrow counter + 1$
22         end
23         $phase \leftarrow phase + 1$;
24         $T \leftarrow T \cdot \beta$;
25     end
26     $reheat \leftarrow reheat + 1$;
27     $T \leftarrow \gamma \cdot bestTemperature$
28 end
29 return $bestFeasible$
```

The algorithm takes an initial schedule as the input and returns the objective value of the best found feasible schedule. The outermost loop is the reheat loop and continues until the maximum number of reheats is reached ($maxR$). The next inner loop is the phase loop and goes on until the phase counter reaches the maximum number of phases ($maxP$). The innermost loop controls the number of iterations for each phase ($maxC$). After the counter reaches that number, the temperature is decreased and the current phase ends. After the middle loop finishes, a reheat is performed, which sets the temperature to γ

Algorithm 5.2: Acceptance handling part of altered TTSA from [1]

```
1  if $accept$ then
2      $S \leftarrow S'$;
3      if $nbv(S) == 0$ then
4          $nbf \leftarrow min(C(S), bestFeasible)$;
5      else
6          $nbi \leftarrow min(C(S), bestInfeasible)$;
7      end
8      if $nbf < bestFeasible$ or $nbi < bestInfeasible$ or $\mathbf{nbv(S)} < \mathbf{lowestViol}$
        then
9          $reheat \leftarrow 0; counter \leftarrow 0; phase \leftarrow 0$;
10         $bestTemperature \leftarrow T$;
11         if $nbf < bestFeasible$ then
12             $bestFeasible \leftarrow nbf$;
13         else if $nbi < bestInfeasible$ then
14             $bestInfeasbile \leftarrow nbi$;
15         end
16         if $\boldsymbol{nbv(S) < lowestViol}$ then
17             $\boldsymbol{lowestViol \leftarrow nbv(S)}$;
18         end
19         if $nbv(S) == 0$ then
20             $w \leftarrow w/\theta$;
21         else
22             $w \leftarrow w \cdot \delta$ ;
23         end
24 end
```

times the *bestTemperature*, which stores the temperature when the last improvement of the schedule was found. In each iteration of the innermost loop a random move is applied to the current schedule S to get to a new schedule S'. If the costs of the new schedule are lower than the costs of the current schedule or the new schedule has no violations and the costs are lower than the *bestFeasible* costs or there are violations and the costs are lower than the *bestInfeasible* costs, then a boolean *accept* is set to true, which marks that the new solution S' will be accepted as the new current solution S.

The costs are calculated with the function C which takes the objective value of a schedule as well as a weighted violation factor into account, see Algorithm 5.4. The function $nbv(S)$ calculates the number of violations of hard constraints of the schedule, the amount of the deviation of the constraint violation is considered as well; moreover, each game that violates the *phased* property of an instance is considered a hard constraint violation as well. The function f is a sublinear function such that $f(1) = 1$, they used $f(v) = 1 + (\sqrt{v} \ln v)/2$; it is used in order to reduce the amount additional violations

Algorithm 5.3: Acceptance rate monitoring for TTSA

```
1  $iteration\_counter++$ if $iteration\_counter \% ITER == 0$ then
2      $acceptance\_rate \leftarrow acceptance\_counter/ITER$;
3      if $length(acceptance\_rates) == QLN$ then
4          remove one entry from queue $acceptance\_rates$;
5      end
6      put $acceptance\_rate$ into queue $acceptance\_rates$;
7      if $sum\ of\ queue\ acceptance\_rates == 0$ then
8          reset $T$ to inital value;
9          $w \leftarrow w/DIV$;
10     end
11     $acceptance\_counter \leftarrow 0$;
12 end
```

contribute to the costs because the first violation should count the most, additional ones are not that important as the schedule is already infeasible. However, one could argue that in order to drive the solutions towards feasibility, each violation should count the same to reduce the infeasibilities like Rosati et al. [39] who also use a fixed weight for their cost function. Therefore, we add an alternative way of taking hard constraint violations into account where each violation counts the same. The weight factor w is used to alter the importance of the violations.

If the schedule is not accepted regarding the costs, it might still be accepted with a probability depending on the quality of the schedule (difference of the new costs to the current costs) and the temperature; the higher the temperature, the more likely the schedule is accepted.

What happens if the schedule is accepted can be seen in Algorithm 5.2 where the current schedule becomes the new one. If a new best feasible or a new best infeasible schedule was found, all loop counters are reset in order to restart the process. The $bestTemperature$ is set to the current temperature (which is later used for the reheats). If the new schedule does not have any violations, the weight factor w is decreased by dividing by θ (which is a value larger than 1). Otherwise, w gets increased by multiplying with δ (which is also a value larger than 1).

One of our adaptions to the original TTSA is that the original does not automatically accept a solution that has the fewest violations of hard constraints so-far, because it may have a far worse objective value regarding the soft constraints. Therefore, we added code so that the smallest value of hard constraint violations is also stored in a variable and if a schedule has fewer violations than the least violations of an already seen schedule, we accept it as the new current solution. We call this concept *minimize violations* or MV for short. This should drive the schedules more towards feasibility than the original TTSA where this is not necessary because an infeasible schedule for TTP usually has

Algorithm 5.4: Adapted function C [1, 39] to calculate the costs of a schedule

```
input  : schedule $S$, boolean $alternative$
output : costs of schedule $S$
1 if $S$ is feasible then
2     return $obj(S)$;
3 else
4     if $alternative$ then
5         return $obj(S) + w \cdot nbv(S)$
6     else
7         return $\sqrt{obj(S)^2 + (w \cdot f(nbv(S)))^2}$
8     end
9 end
```

fewer violations than an infeasible schedule for ITC2021 has since most instances have more constraints than TTP has.

Another adaption we make is that we track the acceptance rates and reset T to its initial value and divide the current weight w by the DIV, which is a number greater than 1 if the acceptance rate of the last $QLN \cdot ITER$ iterations is equal to 0. This is done in order to increase the acceptance rate when the algorithm is stuck in a phase where the temperature is too low to accept worse solutions and better solutions are not found over a longer period. The weight, however, is not reset to its initial value but divided by the DIV. This is done to drive the search towards feasibility because if we reset the weight, it makes it more likely for infeasible solutions to be accepted. Our implementation for this can be seen in Algorithm 5.3 where we put the acceptance rate of the last $ITER$ iterations period into a queue. We store QLN of these acceptance rates value and sum them up; if they equal 0, the resets are performed.

The last adaption we make is that we move the increment of the *counter* outside of the accept block; initially, as proposed by Anagnostopoulos et al. [1] it was only incremented if a neighbor was accepted but did not improve the best solutions (feasible or infeasible). However, this might lead to an infinite loop if the acceptance rates drop to 0 as then the algorithm never finishes its current phase, therefore, it does not reach a reheat, which would increase the acceptance rate again.

For the calculation of the hard constraint violations and the objective value resulting from soft constraint violations, we implemented a function that traverses the schedule round-wise and team-wise. We store the constraints in dictionaries to make them more easily accessible while calculating the violations.

5.2.1 Neighborhood Structures

We use the five neighborhood structures from Anagnostopoulos et al. [1] defined by different moves. All tables of example moves in this subsection are recreated from their work considering an instance with $n = 6$ teams. All of the neighborhoods maintain the DRR structure of the schedule, however, neighbors of feasible schedules are not guaranteed to also be feasible.

SwapHomes

SwapHomes(t_i, t_k): The home venues of the mutual games of the given teams are swapped. In Table 5.1, we can see the operation SwapHomes(t_2, t_4).

Table 5.1: Swap the home venues of the games between team t_2 and t_4.

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	4	3	-5	-4	-3	5	2	-6
2	5	1	-3	-6	**4**	3	6	**-4**	-1	-5
3	-4	5	2	-1	6	-2	1	-6	-5	4
4	3	6	-1	-5	**-2**	1	5	**2**	-6	-3
5	-2	-3	6	4	1	-6	-4	-1	3	2
6	-1	-4	-5	2	-3	5	-2	3	4	1

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	4	3	-5	-4	-3	5	2	-6
2	5	1	-3	-6	**-4**	3	6	**4**	-1	-5
3	-4	5	2	-1	6	-2	1	-6	-5	4
4	3	6	-1	-5	**2**	1	5	**-2**	-6	-3
5	-2	-3	6	4	1	-6	-4	-1	3	2
6	-1	-4	-5	2	-3	5	-2	3	4	1

SwapRounds

SwapRounds(r_i, r_k): Two whole rounds are swapped. In Table 5.2, we can see the operation SwapRounds(r_3, r_5).

Table 5.2: Swap the rounds r_3 and r_5.

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	**4**	3	**-5**	-4	-3	5	2	-6
2	5	1	**-3**	-6	**4**	3	6	-4	-1	-5
3	-4	5	**2**	-1	**6**	-2	1	-6	-5	4
4	3	6	**-1**	-5	**-2**	1	5	2	-6	-3
5	-2	-3	**6**	4	**1**	-6	-4	-1	3	2
6	-1	-4	**-5**	2	**-3**	5	-2	3	4	1

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	**-5**	3	**4**	-4	-3	5	2	-6
2	5	1	**4**	-6	**-3**	3	6	-4	-1	-5
3	-4	5	**6**	-1	**2**	-2	1	-6	-5	4
4	3	6	**-2**	-5	**-1**	1	5	2	-6	-3
5	-2	-3	**1**	4	**6**	-6	-4	-1	3	2
6	-1	-4	**-3**	2	**-5**	5	-2	3	4	1

SwapTeams

SwapTeams(t_i, t_k): The opponents of the given teams are swapped. In Table 5.3, we can see the operation SwapTeams(t_2, t_5). Of course, the mutual games are not swapped, otherwise, the teams would play against themselves.

PartialSwapRounds

PartialSwapRounds(t_i, r_k, r_l): The games of team t_i are swapped in the given rounds. Afterwards, the schedule is updated deterministically to make up a feasible schedule again.

In order to achieve that, an ejection chain of the teams that need to swap their games in rounds r_k and r_l is generated. The chain starts with t_i. The next team t_{next} added to

Table 5.3: Swap games of teams t_2 and t_5.

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	4	3	-5	-4	-3	5	2	-6
2	5	**1**	**-3**	**-6**	**4**	**3**	**6**	**-4**	**-1**	-5
3	-4	5	2	-1	6	-2	1	-6	-5	4
4	3	6	-1	-5	-2	1	5	2	-6	-3
5	-2	**-3**	**6**	**4**	**1**	**-6**	**-4**	**-1**	**3**	2
6	-1	-4	-5	2	-3	5	-2	3	4	1

T/R	1	2	3	4	5	6	7	8	9	10
1	6	**-5**	4	3	**-2**	-4	-3	**2**	**5**	-6
2	5	**-3**	**6**	**4**	**1**	**-6**	**-4**	**-1**	**3**	-5
3	-4	**2**	**5**	-1	6	**-5**	1	-6	**-2**	4
4	3	6	-1	**-2**	**-5**	1	**2**	**5**	-6	-3
5	-2	**1**	**-3**	**-6**	**4**	**3**	**6**	**-4**	**-1**	2
6	-1	-4	**-2**	**5**	-3	**2**	**-5**	3	4	1

the chain is always the opponent of the previously added team in the currently concerned round; the concerned round alternates between r_k and r_l. The chain is finished when the next team to add would be t_i. In the end, the teams contained in the chain swap their games in round r_k and r_l.

For example, in Table 5.4, the operation PartialSwapRounds(t_2, r_2, r_9) can be seen. The ejection chain starts with t_2, then the team that t_2 plays in round r_2, which is t_1, is added to the chain. Afterwards, we have to add the team that t_1 plays in round r_9, which is t_4. In the same way, team t_6 is added. Then, the next team to add would be t_2, which means the chain is finished. In the end, the teams in the chain swap their games in r_2 and r_9.

Table 5.4: Partially swap rounds r_2 and r_9 starting with team t_2.

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	2	3	-5	-4	-3	5	4	-6
2	5	**1**	-1	-5	4	3	6	-4	**-6**	-3
3	-4	5	4	-1	6	-2	1	-6	-5	2
4	3	6	-3	-6	-2	1	5	2	-1	-5
5	-2	-3	6	2	1	-6	-4	-1	3	4
6	-1	-4	-5	4	-3	5	-2	3	2	1

T/R	1	2	3	4	5	6	7	8	9	10
1	6	**4**	2	3	-5	-4	-3	5	**-2**	-6
2	5	**-6**	-1	-5	4	3	6	-4	**1**	-3
3	-4	5	4	-1	6	-2	1	-6	-5	2
4	3	**-1**	-3	-6	-2	1	5	2	**6**	-5
5	-2	-3	6	2	1	-6	-4	-1	3	4
6	-1	**2**	-5	4	-3	5	-2	3	**-4**	1

PartialSwapTeams

PartialSwapRounds(r_i, t_k, t_l): The games of the given teams are swapped in round r_i. Afterwards, the schedule is updated deterministically to make up a feasible schedule again.

Let the opponents of t_k in round r_i be op_k and the opponent of t_l in round r_i be op_l. First, we change the schedule so that the opponent of team t_k in round r_i is now op_l and vice versa and the opponent of team t_l is now op_k and vice versa. Thereby, t_k plays that game against op_l twice (once in round r_i and once in the original round r_o). Therefore, the opponents of t_k and t_l are also swapped in round r_o. This is done repeatedly until the new opponent of t_k in a round is the original opponent of round r_i, namely op_k.

In Table 5.5, we can see the operation PartialSwapRounds(r_9, t_2, t_4). First, the opponents are swapped for teams t_2 and t_4 in round r_9 (and the opponents of their opponents—being t_2 and t_4—are swapped for them as well). Now t_2 has two away games against t_6 in its schedule, one in r_9 and one in r_4. Therefore, we also swap the opponents of t_2 and t_4 in r_4. However, now t_2 has two away games against t_5, one in t_4 and one in t_{10}. Hence, we also swap the opponents of t_2 and t_4 in round r_{10}. Then, t_2 has two away games against t_3, one in r_{10} and one in r_3. Therefore, we swap the opponents of t_2 and t_4 in round r_3. Now t_2 has no doubled games because the opponent it received in the last swap was the original opponent in round r_9.

Table 5.5: Partially swap teams t_2 and t_4 starting in r_9.

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	4	3	-5	-4	-3	5	2	-6
2	5	1	-3	-6	4	3	6	-4	**-1**	-5
3	-4	5	2	-1	6	-2	1	-6	-5	4
4	3	6	-1	-5	-2	1	5	2	**-6**	-3
5	-2	-3	6	4	1	-6	-4	-1	3	2
6	-1	-4	-5	2	-3	5	-2	3	4	1

T/R	1	2	3	4	5	6	7	8	9	10
1	6	-2	**2**	3	-5	-4	-3	5	**4**	-6
2	5	1	**-1**	**-5**	4	3	6	-4	**-6**	**-3**
3	-4	5	**4**	-1	6	-2	1	-6	-5	**2**
4	3	6	**-3**	**-6**	-2	1	5	2	**-1**	**-5**
5	-2	-3	6	**2**	1	-6	-4	-1	3	**4**
6	-1	-4	-5	**4**	-3	5	-2	3	**2**	1

Algorithm 5.5: *RandomSchedule* function for generating an initial schedule [1]

```
   input  : teams $T$, slots $R$
   output : DRR schedule $S$
 1 $Q \leftarrow \{(t, s) | t \in T, s \in R\}$;
 2 $S \leftarrow$ empty schedule matrix with size $|T| \times |R|$;
 3 $GenerateSchedule(Q, S, 0)$;
 4 return $S$;
```

5.2.2 Initial Solution Generation

Since SA needs an initial solution, we propose an altered backtracking algorithm based on the work of Anagnostopoulos et al. [1]. In Algorithm 5.5 and 5.6, we can see our approach to construct an initial solution. The algorithm fills a schedule by generating all possible games for the team with the lowest index t that has no opponent assigned in one of its rounds r. Of these possible games, one is scheduled in round r and the next position of the schedule is filled with a recursive call. If the algorithm does not yield a feasible schedule, we backtrack and try other choices for that schedule position. If the number of backtracks surpasses a certain threshold, we start over because then it is likely that the algorithm is stuck in a branch that does not yield a feasible schedule at all.

The function *randomSchedule* takes the number of teams and the number of slots as an input. First, it initializes a queue Q of tuples in the form (t, s) consisting of a team t and a slot s, moreover, an empty schedule matrix S with the number of teams as the row count and the number of slots as the column count. The tuples $(t, s) \in Q$ represent that t has no scheduled game in slot s in the schedule S yet. After the preliminaries are done,

Algorithm 5.6: Altered backtracking algorithm *GenerateSchedule* to create an initial DRR tournament [1]

```
input  : queue $Q$, schedule $S$, teams $T$, backtrack counter $cnt$
output : returns true, if a game could be scheduled or the schedule is finished, false otherwise
if $size(Q) = 0$ then
    return $true$;
end
$t, s \leftarrow min(Q)$;
$choices \leftarrow []$;
$prevGames \leftarrow S[t]$;
foreach $x \in T$ do
    if $x \neq t$ then
        if $x \notin prevGames$ then
            add $x$ to choices;
        end
        if $-x \notin prevGames$ then
            add $-x$ to choices;
        end
    end
end
foreach $o \in choices$ randomly ordered do
    if $(abs(o), s) \in Q$ then
        $S[t][s] \leftarrow o$;
        if $o > 0$ then
            $S[o][s] \leftarrow -t$;
        else
            $S[-o][s] \leftarrow t$;
        end
        if $GenerateSchedule(Q - \{(t, s), (abs(o), s)\}, S, T, cnt)$ then
            return $true$;
        end
    end
end
$cnt \leftarrow cnt + 1$;
if $cnt = n_{\text{bt}}^{\text{lim}}$, where $n_{\text{bt}}^{\text{lim}}$ is the maximum number of backtracks allowed then
    restart RandomSchedule
end
return $false$;
```

the *GenerateSchedule* function is called, which directly alters the schedule S, which is returned afterwards.

The *GenerateSchedule* function is a recursive function that fixes one game played in each recursive step. Therefore, the recursive stopping condition is that no games need to be scheduled anymore, which means the queue Q is empty. The function returns *true* if a game was scheduled or if there are no games left to be scheduled.

Otherwise, we take the smallest tuple (t, s) from the queue, which is the smallest remaining team, with its smallest remaining slot to be filled in the schedule. Then the possible choices for that team and that slot are calculated, which is done by traversing the teams as x and checking if x has already played against t. Of course, there are two separate checks for home and away games. If t did not play against x, at home x is added to the possible choices; if t did not play against x on the road, $-x$ is added to the choices.

Then we traverse those choices in a random order, where o is the current choice. If team o is still in the queue for slot s, then the game between t and o is scheduled according to the sign of o. Afterwards, the function is called recursively but with (t, s) and $(abs(o), s)$ removed from the queue. If that call makes it down to the recursion stopping condition,

Algorithm 5.7: Thread Function Parallel Multi-Start Hybridization CP/SA

```
   input  : instance file $I$
   output : file with info about run
1  $perm$ ← random shuffle of list $[1, \ldots, n]$;
2  $perm\_lookup$ ← generate lookup from $perm$;
3  $initialSchedule$ ← start MiniZinc with $perm$ and a model for $I$ with a defined
     timeout;
4  if not $initialSchedule$ then
5      $initialSchedule$ ← start MiniZinc with $perm$ and a model for $I$ which ignores
         all break constraints and has a defined timeout
6  end
7  if not $initialSchedule$ then
8      $initialSchedule$ ← start MiniZinc with $perm$ and a model for $I$ which ignores
         all constraints and has a defined timeout
9  end
10 start SA with $initialSchedule$
```

we return *true* here as well.

If all choices of the current state are traversed and none of them managed to complete a schedule, we return $false$ to signal the upper level that it needs to try another possible choice. We also monitor the number backtracking steps performed and limit it by $n_{\text{bt}}^{\text{lim}}$. Thereby, we can completely restart from scratch if we are caught in a branch where it is difficult or unlikely to find a feasible DRR schedule.

5.3 Parallel Multi-Start Hybridization

In this section, we describe our approach to combine constraint programming with simulated annealing in a parallel multi-start approach.

The main idea is to start a predefined number of threads that search for a solution in parallel, a straightforward multi-start approach. In Algorithm 5.7, we can see the thread function in pseudocode. First, we try to find a solution for the instance with all hard constraints enabled but without soft constraints, thereby, the instances become ITC2021-NSC instances. If this does not yield a solution after a predefined time, we kill the process and run MiniZinc with a model where we ignore the break constraints as those are the hardest constraints to solve, thereby, the instances become ITC2021-NSC-NBC instances. If this still does not lead to an initial solution, we run MiniZinc with a model where we ignore all constraints leading to a random DRR. We then give the resulting schedule to the SA as an initial solution. The SA has two tasks: improve a feasible solution and secondly make infeasible solutions feasible. In the end, the results of each thread are collected, written into a file, and the best found solution is returned.

CHAPTER 6

Computational Study

The experiments in this section were performed on Intel Xeon E5-2640 processors with 2.40 GHz in single-threaded mode, except for when it is explicitly defined that they ran in multi-threaded mode. The translation from the XML instance files to MiniZinc models and the hybridization approach were implemented in Python 3.7. The simulated annealing approach was implemented and tested in Julia 1.7.3. The first experiments on the MiniZinc models were performed with MiniZinc 2.5.5 and later with 2.6.1, which is mentioned at the beginning of the corresponding sections.

6.1 Constraint Programming with MiniZinc

We first discuss different parameters for our experiments with MiniZinc, namely the backend solver, variable and value choice heuristics, and randomization aspects.

6.1.1 Solvers

In this section, we briefly describe the tested CP solvers for MiniZinc.

Gecode

Gecode (short for generic constraint development environment) [44] is a software toolkit written in C++ started by Christian Schulte in 2005 for solving CSPs. It is open source, distributed under the MIT license and provides many features, for example, advanced branching heuristics, many search engines and automatic symmetry breaking. It is the default solver in the MiniZinc bundled binary distribution. It supports many of MiniZinc's global constraints natively and supports integer, float and set variables [41, 3.5].

Chuffed

Chuffed [6] uses lazy clause generation to solve CSPs. Lazy clause generation is a hybrid approach that combines features of finite domain propagation and SAT solving. The former is used to record the reasons for the propagation steps, thereby creating an implication graph. Implication graphs are also built by SAT solvers and with the help of them one can create so-called *nogoods* that record the reason for failure, which can be propagated using SAT unit propagation technology.

Gurobi

Gurobi [20][4] was founded by Gu, Rothenberg and Bixby (hence GuRoBi) in 2008. The Gurobi Solver is a commercial mathematical programming solver for solving linear programming (LP), quadratic programming and mixed integer programming (mixed integer linear programming, mixed integer quadratic programming, mixed integer quadratically constrained programming) problems. However, it can also be used to some extent for CSPs and COPs in MiniZinc when MiniZinc was compiled with its support [41, 3.5], where a CP model is transformed into an integer programming (IP) model.

or-tools

Google's or-tools (operations research tools) [35] are an open source software suite for solving SAT, LP and CP problems; they were first introduced by Perron [34].

6.1.2 Search Parameters

There are several parameters one can set when solving a MiniZinc model [41, 2.5]. In the following subsection, we describe those we used in our experiments.

As we have two arrays of decision variables, we can decide for which we want to define the search process with the *int_search* annotation; the other one will be searched freely (without a manual definition) after all variables of the first array are assigned.

In our preliminary experiments, we found that it is best to define the search for *opponents per team and round* as we found solutions more quickly that way than when we defined the search for *venue per team and round*. This is due to the larger domain size of the former. One could also use *seq_search* to define a search process for both, which we did not consider.

Variable Choice

For deciding the next variable, i.e. team and round, on which to branch, we consider the following variable choice heuristics:

- *input_order* takes the variables one by one in the order they were defined; in our case, the teams and rounds in lexicographic order.

- *first_fail* chooses the variable with the smallest domain size; in our case the team and round with the least opponents left.
- *smallest* chooses the variable with the smallest value in its domain.
- *dom_w_deg* takes the variable with the smallest domain size divided by a weighted degree of how often that variable caused a failure earlier during the search.
- *random/random_order* chooses the variables in a random order.

The first three variable choice heuristics work on all described solvers, *dom_w_deg* only works with Gecode and Gurobi. The random choice is called *random* for Gecode and *random_order* for Chuffed; it does not work for or-tools and Gurobi.

Value Choice

For constraining the domain of the chosen variable in the left branch (with the corresponding complement in the right branch), we consider the following different value choice heuristics:

- *indomain_min* assigns the smallest value of the domain.
- *indomain_median* assigns the median value of the domain.
- *indomain_split* bisects the variable's domain, excluding the upper half first.
- *indomain_random* assigns a random value of the domain.

The first three work for every described solver. *indomain_random* works only for Gurobi and Gecode. Since we wanted some sort of randomness for chuffed regarding the value choice, we implemented our own parameter for chuffed (see 6.1.3).

Restarting

As restarting options, we consider the following settings:

- no restarting
- $restart\_constant(n)$ where n defines after how many nodes of the search tree the search is restarted.
- $restart\_linear(n)$ where n defines after how many nodes the search is restarted; the second restart happens after $2n$, the third after $3n$, etc.
- $restart\_geometric(b, n)$ where b is a float base value and n is the factor, the k-th restart happens after $n \cdot b^k$ nodes.

- *restart_luby*(n) - the k-th restart happens after $n \cdot L[k]$ where L is the Luby sequence that goes 112112411211248... (it always repeats itself before adding the next power of two).

6.1.3 Randomization for Value Choice in Chuffed

Since Chuffed does not support *indomain_random* and we wanted to make use of randomness when choosing a value for a chosen variable in our randomized multi-start approach, we implemented it on our own for Chuffed. The reason why we wanted randomness in value choice is that the other value choice options (e.g., *indomain_min* or *indomain_median*) only restrict the value choice in a certain way, but do not improve the probability to find a feasible solution for scheduling problems when using restarting. Since the code for that *random* value choice option seems to be just commented out in the code basis of Chuffed, we first tried to implement it there. However, only commenting in that code did not work, and all other attempts to get it to work failed, which is why we went with a different approach.

The idea is to randomize the model by generating a random permutation *perm* of the teams and use this permutation in all of the constraints when accessing the *venue_per_team_and_round* or *opponents_per_team_and_round* variables. In Listing 6.1, we can see this approach in action. The *perm_lookup* is needed to map the teams back according to the permutation for the output of the schedule. This can be seen at the end of the listing.

Listing 6.1: Example of a CA1 constraint with randomness for Chuffed

```
<CA1 max="1" min="0" mode="H" penalty="5" slots="9;6;7" teams="0">

perm = [2,4,6,3,1,5]
perm_lookup = [5,1,4,2,6,3]
constraint sum(slot in {10,7,8}
               where venue_per_team_and_round[perm[1],slot] == 1)(1) <= 1;

output [ format(3, 2,
        perm_lookup[opponents_per_team_and_round[perm[i],j]] - 1) ++
        if j == R then "\n" else " " endif |
        i in TEAMS, j in SLOTS
];
```

6.2 Instances

The competition provided four instance groups [1], namely Test, Early, Middle and Late, with Test containing nine instances and the others containing 15 instances each. These instances were used to test our approaches. Further information about the instances can be seen in Table 6.1 where we show how many of each constraint types there are for every instance.

[1]https://www.sportscheduling.ugent.be/ITC2021/instances.php

Table 6.1: Instances provided by the competition

					CA1		CA2		CA3		CA4		GA1		BR1		BR2		FA2		SE1	
Instance	teams	phased	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S
Demo	4	✓	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Test1	6	✓	17	44	10	17	0	0	1	16	0	0	5	10	0	0	1	0	0	0	0	1
Test2	6	X	10	43	10	20	0	12	0	0	0	0	0	0	0	10	0	0	0	1	0	0
Test3	6	X	47	67	15	25	8	0	2	20	22	22	0	0	0	0	0	0	0	0	0	0
Test4	6	✓	51	197	15	30	15	60	1	40	4	44	10	10	5	10	1	1	0	1	0	1
Test5	16	✓	68	75	8	29	16	0	0	0	0	21	31	5	13	20	0	0	0	0	0	0
Test6	18	X	214	575	40	31	66	172	2	72	85	297	0	2	21	0	0	1	0	0	0	0
Test7	20	✓	178	1091	40	0	72	619	2	112	50	340	0	19	13	0	1	0	0	0	0	1
Test8	20	✓	47	672	5	15	12	264	1	45	29	305	0	41	0	0	0	1	0	0	0	1
Early1	16	✓	83	113	25	17	10	0	0	0	0	84	12	10	35	0	1	0	0	1	0	1
Early2	16	✓	53	114	38	30	0	0	2	82	0	0	0	1	12	0	1	0	0	1	0	0
Early3	16	✓	148	186	24	0	72	21	0	112	0	0	34	51	18	0	0	1	0	1	0	0
Early4	18	✓	164	268	0	32	0	235	0	0	85	0	34	0	44	0	1	0	0	0	0	1
Early5	18	✓	207	587	41	27	36	331	2	111	81	117	23	0	23	0	1	0	0	0	0	1
Early6	18	✓	192	797	38	31	71	591	2	54	81	115	0	3	0	0	0	1	0	1	0	1
Early7	18	X	175	1159	42	31	30	620	1	112	84	340	5	55	12	0	1	0	0	0	0	1
Early8	18	X	70	582	19	0	8	57	0	112	0	339	4	73	39	0	0	0	0	1	0	0
Early9	18	X	90	102	39	0	14	0	0	88	0	0	14	2	23	10	0	1	0	1	0	0
Early10	20	✓	246	1015	42	32	72	620	2	23	85	339	0	0	44	0	1	0	0	0	0	1
Early11	20	X	246	1108	42	32	72	620	2	112	85	340	0	3	44	0	1	0	0	0	0	1
Early12	20	✓	179	35	37	0	72	0	2	20	31	0	17	1	20	13	0	1	0	0	0	0
Early13	20	X	100	432	41	31	27	257	1	110	0	0	10	24	20	10	1	0	0	0	0	0
Early14	20	X	56	56	5	30	0	0	0	0	0	0	34	0	17	24	0	1	0	1	0	0
Early15	20	X	187	1224	42	0	72	620	2	112	71	340	0	126	0	24	0	1	0	1	0	0
Middle1	16	✓	144	993	0	32	14	620	0	0	85	340	0	0	44	0	1	0	0	0	0	1
Middle2	16	✓	246	1231	42	32	72	620	2	112	85	340	0	126	44	0	1	0	0	0	0	1
Middle3	16	X	237	1212	42	0	72	617	2	107	85	338	0	126	36	21	0	1	0	1	0	1
Middle4	18	✓	97	168	31	18	17	0	1	0	0	41	25	85	23	24	0	0	0	0	0	0
Middle5	18	✓	151	197	41	24	40	33	0	12	0	0	26	126	44	0	0	1	0	1	0	0
Middle6	18	✓	162	154	39	0	41	0	2	111	30	27	6	1	44	13	0	1	0	0	0	1
Middle7	18	X	141	476	0	30	0	355	1	0	78	51	34	17	28	21	0	1	0	0	0	1
Middle8	18	X	62	224	16	0	0	27	2	108	0	34	12	41	32	14	0	0	0	0	0	0
Middle9	18	X	94	201	42	0	0	37	1	100	19	39	4	0	28	23	0	1	0	1	0	0
Middle10	20	✓	198	714	41	15	71	363	0	0	46	262	0	74	39	0	1	0	0	0	0	0
Middle11	20	✓	176	1048	7	0	71	612	2	88	84	340	0	7	12	0	0	0	0	1	0	0
Middle12	20	✓	63	241	0	32	28	168	1	13	0	0	5	4	29	21	0	1	0	1	0	1
Middle13	20	X	219	350	42	29	72	242	1	0	85	76	7	2	12	0	0	0	0	0	0	1
Middle14	20	X	63	817	5	18	11	319	2	112	0	338	6	19	38	10	1	0	0	1	0	0
Middle15	20	X	95	133	12	0	23	0	0	77	0	0	23	44	37	10	0	1	0	0	0	1
Late1	16	X	235	542	42	32	72	198	1	13	82	283	0	15	38	0	0	0	0	1	0	0
Late2	16	X	246	1077	42	0	72	620	2	112	85	340	0	5	44	0	1	0	0	0	0	0
Late3	16	X	127	439	42	0	72	326	1	43	0	60	0	7	12	0	0	1	0	1	0	1
Late4	18	✓	96	34	0	32	0	0	0	0	18	0	34	1	44	0	0	0	0	0	0	1
Late5	18	✓	176	747	0	19	69	614	2	0	81	109	23	4	0	0	1	0	0	1	0	0
Late6	18	✓	163	159	0	32	0	125	0	0	85	0	34	0	44	0	0	1	0	0	0	1
Late7	18	X	126	738	42	32	40	601	1	61	0	0	5	43	37	0	1	0	0	0	0	1
Late8	18	✓	110	195	37	15	0	14	0	111	0	0	32	29	41	24	0	1	0	0	0	1
Late9	18	X	102	402	40	0	20	250	2	112	0	0	0	18	40	20	0	1	0	1	0	0
Late10	20	✓	233	694	0	31	67	447	2	0	85	205	34	10	44	0	1	0	0	0	0	1
Late11	20	✓	52	366	6	0	16	274	0	88	0	0	17	3	12	0	1	0	0	1	0	0
Late12	20	X	244	1009	40	32	72	620	2	16	85	340	0	0	44	0	1	0	0	0	0	1
Late13	20	X	169	134	14	32	72	15	2	0	81	71	0	13	0	0	0	1	0	1	0	1
Late14	20	X	116	993	42	0	72	390	2	112	0	340	0	126	0	24	0	0	0	1	0	0
Late15	20	X	51	41	5	0	0	0	0	15	0	0	34	0	12	24	0	1	0	1	0	0

6.3 First Comparison of MiniZinc Configurations

The experiments in this section were performed on MiniZinc 2.5.5.

First, we tried different configurations on the provided test instances. Because of the low-level solver input language FlatZinc, there are many solvers that can be used to solve MiniZinc models. Our first experiments were performed using the solvers Gecode, Chuffed and Gurobi. We used the variable choice annotations *input_order*, *first_fail*, *smallest*, *dom_w_deg*. As value choice heuristics, we used *indomain_min*, *indomain_median*, *indomain_random* and *indomain_split*. *indomain_random* and *dom_w_deg* do not work for Chuffed; *random* does not work for Gurobi.

Moreover, we also tried different settings for restarting with Gecode. We used constant, linear, geometric and luby restarting with different values (10, 100, 1000, 10000) with random variable choice and random value choice. Furthermore, we tried complete random

Table 6.2: Best results on Test instances after running the experiments for 24 hours. obj_{best}: the objective value of the best solution, gap_{best}: relative deviation to the best known solution of our best solution, $config$: the configuration that achieved that solution, $t_{best}[s]$: time to best solution.

	Gecode				Chuffed				Gurobi			
	$config$	obj_{best}	gap_{best}	$t_{best}[s]$	$config$	obj_{best}	gap_{best}	$t_{best}[s]$	$config$	obj_{best}	gap_{best}	$t_{best}[s]$
TestInstanceDemo	sm_min	0	0	1	sm_min	0	0	1	sm_split	0	0	1
ITC2021Test1	dwd_min	1066	0	14	ff_min	1066	0	17	ff_split	1066	0	2550
ITC2021Test2	sm_median	176	0	84	sm_median	176	0	160	ff_rand	176	0	56
ITC2021Test3	dwd_min	1253	0	6	dwd_median	1253	0	4	dwd_median	1253	0	8
ITC2021Test4	sm_median	4535	0	6	io_median	4535	0	7	dwd_median	4535	0	11
ITC2021Test5	rand_rand_luby(1000)	200	9900	51849	sm_min	152	7500	83884	dwd_min	2	0	12122
ITC2021Test6	-	-	-	-	io_median	4396	40	30214	-	-	-	-
ITC2021Test7	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Test8	-	-	-	-	-	-	-	-	-	-	-	-

runs with Gecode, variable choice *random* and value choice *indomain_random*. Overall, we tested 57 configurations in this experiment.

In Table 6.2, we can see the results of these experiments. We show the configurations that found our best solution for each instance and each solver. Moreover, we see the objective value, the gap to the best known solution expressed as a percentage and the time it took the configuration to obtain its solution. In Table 6.3, we can see the fastest time to first solution for each instance and solver.

We can see that Chuffed finds a solution to one more instance than the other two solvers. Furthermore, we see that Gurobi finds the best solution to each of the instances where it finds a solution, however, it takes the longest to find its first solutions. Gecode is the fastest to find its first solutions and its best solutions, however, the objective value for test instance 5 is the worst among the three solvers.

In Figure 6.1, we see the success rate of each solver for the test instances 1 to 6 and the demo instance over different configurations. The runs that found a solution are shown in green, the ones that did not are shown in red. On the left, the domain value choice *indomain_random* is not considered since Chuffed does not support it. On the right, we show the diagram where it is considered for Gecode and Gurobi. As already stated, Chuffed also finds a solution for test instance 6. However, Gurobi finds a solution for each test run except for those with test instance 6. Therefore, the success rate of these solvers is higher than that of Gecode.

After performing our initial experiments only on the test instances, we also ran experiments on the Early, Middle and Late instances. We chose a promising looking configuration without restarting for each of the solvers to run on the other instances, namely Chuffed and Gurobi with *input_order* and *smallest* to choose the variable and *indomain_median* to choose the value of the variable. Moreover, we tried Gecode with *input_order* and *indomain_median*. Furthermore, we also tried Chuffed and Gurobi with *first_fail* and *indomain_min*. In addition, we tested one configuration with restarting, namely Gecode with *dom_w_deg* and *indomain_random*, with constant restarting and restarts after 10 000 fails. Overall we tested 8 configurations with Early, Middle and Late instances.

Table 6.3: Fastest time to first solution on Test instances after running the experiments for 24 hours. obj_{first}: the objective value of the first solution, gap_{first}: relative deviation to the best known solution of our first solution, $config$: the configuration that achieved that solution, $t_{first}[s]$: time to first solution.

	Gecode				Chuffed				Gurobi			
	$config$	obj_{first}	gap_{first}	$time_{first}$	$config$	obj_{first}	gap_{first}	$time_{first}$	$config$	obj_{first}	gap_{first}	$time_{first}$
TestInstanceDemo	sm_min	0	0	1	sm_min	0	0	1	sm_split	20	1900	1
ITC2021Test1	rand_rand_geometric(1.5,1000)	1162	9	3	io_split	1103	3	2	ff_split	1137	7	10
ITC2021Test2	rand_rand_luby(100)	297	69	3	dwd_min	197	12	3	ff_rand	339	93	24
ITC2021Test3	dwd_min	1253	0	6	dwd_median	1253	0	4	dwd_median	1253	0	8
ITC2021Test4	sm_median	4535	0	6	io_median	4535	0	7	dwd_median	4535	0	11
ITC2021Test5	dwd_min	375	18650	6	io_median	230	11400	18	sm_median	259	12850	302
ITC2021Test6	-	-	-	-	io_median	4576	46	13623	-	-	-	-
ITC2021Test7	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Test8	-	-	-	-	-	-	-	-	-	-	-	-

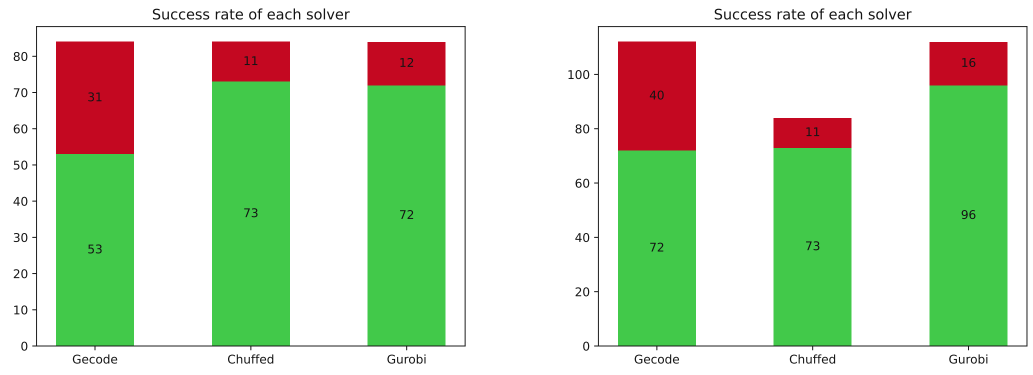

Figure 6.1: Experiments on test instances with the configurations of the experiment shown in 6.2, only for instances Demo and 1-6; green bars show the number of runs that found a solution with that solver, red bars show the amount that did not. Left: *indomain_random* is ignored as Chuffed does not support it, right: *indomain_random* is included (for Gecode and Gurobi).

Unfortunately, Gurobi only found a solution for one of the 45 instances. Moreover, it also resulted in a wrong solution in which nearly all of the games should have been played by team 1 (against itself), indicating an issue with the CP/IP model conversion. Therefore, we did not further consider Gurobi.

For all of the instances where one of the runs found a solution, we also ran experiments to compare the restarting results with the experiments without restarting. We used various restarting configurations, with *dom_w_deg* (since it remembers part of the search tree for the restart) and *random_order*; for each configuration three runs were performed. Usually, the goal of restarting is to find solutions to difficult instances; however, we restricted ourselves to the instances where we found a solution without restarting in order to reduce the number of runs we needed to perform and to compare the resulting objective values with the non-restarting runs. Moreover, we wanted to try restarting on the easier instances to test if they find a solution to these.

Table 6.4: Best results on Early, Middle and Late instances after running the experiments for 24 hours. obj_{best}: the objective value of the best solution, gap_{best}: relative deviation to the best known solution of our best solution, $config$: the configuration that achieved that solution. Entries with $(*)$ mean that no restarting configuration found a solution for that instance.

	Gecode			Chuffed		
	obj_{best}	$gap_{best}(\%)$	$config$	obj_{best}	$gap_{best}(\%)$	$config$
Early3	**3626**	**265**	dwd_rand_luby(10)	-	-	-
Early12	-	-	-	**1775**	**367**	sm_median $(*)$
Middle4	257	3571	dwd_rand_const(10 000)	**225**	**3114**	io_median
Middle6	-	-	-	**2560**	**129**	io_median $(*)$
Middle7	-	-	-	**8521**	**378**	ff_min $(*)$
Middle8	1726	1238	io_median $(*)$	**1218**	**844**	ff_min $(*)$
Middle13	-	-	-	**8417**	**3240**	ff_min $(*)$
Middle15	**7055**	**1355**	dwd_rand_const(10 000)	-	-	-
Late1	-	-	-	**3167**	**64**	**ff_min** $(*)$
Late3	7317	209	dwd_rand_const(10 000)	**6958**	**194**	rand_min_geo(1.5,100)
Late4	-	-	-	**0**	**0**	io_median
Late8	**3526**	**278**	io_median	-	-	-

Table 6.5: Fastest time to first solution on Early, Middle and Late instances after running the experiments for 24 hours. obj_{first}: the objective value of the first solution, t_{first}: the time to find the first solution, gap_{first}: gap to the best known solution of the first solution, $config$: the configuration that achieved that solution.

	Gecode				Chuffed			
	obj_{first}	$gap_{first}(\%)$	$t_{first}[s]$	$config$	obj_{first}	$gap_{first}(\%)$	$t_{first}[s]$	$config$
Early3	6082	513	92	dwd_rand_const(10 000)	-	-	-	-
Early12	-	-	-	-	1915	404	400	io_median
Middle4	311	4343	79	dwd_rand_const(10 000)	316	4414	24	ff_min
Middle6	-	-	-	-	2685	140	16159	io_median
Middle7	-	-	-	-	11328	535	4018	io_median
Middle8	1765	1268	111	io_median	1349	946	306	ff_min
Middle13	-	-	-	-	12953	5040	1333	io_median
Middle15	9321	1822	62	dwd_rand_luby(10)	-	-	-	-
Late1	-	-	-	-	3735	94	3128	ff_min
Late3	8803	272	952	dwd_rand_linear(1000)	10674	351	1711	ff_min
Late4	-	-	-	-	285	-	128	io_median
Late8	4785	412	82	dwd_rand_const(10 000)	-	-	-	-

The best solutions found after 24 hours are shown in 6.4 (the instances where no restarting run found a solution are marked with $(*)$). The runs with the fastest time to first solution are shown in 6.5. The results show that Chuffed is able to find solutions for more instances than Gecode. Moreover, restarting is not to favor for Chuffed as it does not have an out of the box randomization whereas for Gecode those runs find the best solutions of the Gecode runs.

Table 6.6: Best solutions after running the experiments for 24 hours with the newer MiniZinc version 2.6.1. If the best known solution is 0 (and we do not find such a solution), we calculate the gap by setting the best known solution to 1.

instance	*configuration*	obj_{best}	%	t_{best} [s]	*restart*
Early3	Gecode-dom-w-deg-indomain-random-restart-linear(1000)	2673	169	71509	✓
Early8	Gecode-dom-w-deg-indomain-random-restart-geometric(1.5,100)	3455	229	79425	✓
Early9	or-tools-input-order-indomain-median	2763	4833	37866	✓
Early12	Chuffed	1680	425	19184	X
Early14	or-tools-input-order-indomain-median	4250	106150	84626	✓
Early15	Chuffed-first-fail-indomain-min	6572	115	79378	X
Middle4	Chuffed-input-order-indomain-median	225	3114	74140	✓
Middle5	Gecode-dom-w-deg-indomain-random-restart-geometric(1.5,100)	2490	744	44369	✓
Middle6	Chuffed-input-order-indomain-median	2560	129	60736	X
Middle7	Chuffed-first-fail-indomain-min	7985	348	65446	X
Middle8	or-tools-first-fail-indomain-median	1203	833	85135	X
Middle9	or-tools-input-order-indomain-min	3105	606	74346	X
Middle12	Gecode-input-order-indomain-median	3229	439	12436	✓
Middle13	Chuffed-first-fail-indomain-median	7084	2711	15766	X
Middle15	Gecode-dom-w-deg-indomain-random-restart-luby(10)	6774	1297	74250	✓
Late1	Chuffed-first-fail-indomain-median	3062	59	23155	X
Late3	Chuffed-random-order-indomain-min-restart-geometric(1.5,100)	6859	190	65569	✓
Late4	Chuffed-input-order-indomain-median	0	0	157	✓
Late8	or-tools-input-order-indomain-min	3101	232	17773	✓
Late9	or-tools-input-order-indomain-median	2824	436	68709	X
Late13	Chuffed-first-fail-indomain-median	10900	499	81137	X
Late14	or-tools-first-fail-indomain-min	3520	193	83388	X
Late15	or-tools-input-order-indomain-median	3845	3745	83799	✓

Table 6.7: Fastest time to first solution for the same experiments as used in Table 6.6. If the best known solution is 0 (and we do not find such a solution), we calculate the gap by setting the best known solution to 1.

instance	*configuration*	obj_{first}	%	t_{first} [s]	*restart*
Early3	Gecode-dom-w-deg-indomain-random-restart-linear(1000)	6119	517	19	✓
Early8	Gecode-random-indomain-random-restart-luby(1000)	7616	625	17	✓
Early9	Gecode-dom-w-deg-indomain-random-restart-luby(10)	8088	14342	16	✓
Early12	Chuffed-input-order-indomain-min	1805	464	274	X
Early14	Gecode-dom-w-deg-indomain-random-restart-constant(10000)	10651	26617	23	✓
Early15	Chuffed-first-fail-indomain-min	6583	115	48751	X
Middle4	Chuffed-first-fail-indomain-min	316	4414	16	✓
Middle5	Gecode-dom-w-deg-indomain-random-restart-geometric(1.5,100)	7949	2595	20	✓
Middle6	Chuffed-first-fail-indomain-min	3415	205	11649	X
Middle7	Chuffed-first-fail-indomain-min	8792	393	4027	X
Middle8	Gecode	1865	1346	13	X
Middle9	or-tools-first-fail-indomain-median	3590	716	25313	X
Middle12	Gecode-input-order-indomain-median	4060	578	48	✓
Middle13	Chuffed-input-order-indomain-median	11995	4660	809	X
Middle15	Gecode-dom-w-deg-indomain-random-restart-geometric(1.5,100)	9824	1926	36	✓
Late1	Chuffed-first-fail-indomain-min	3210	67	4356	X
Late3	Gecode-dom-w-deg-indomain-random-restart-luby(10)	7944	235	119	✓
Late4	Chuffed-first-fail-indomain-min	984	884	71	✓
Late8	Gecode-first-fail-indomain-min	4863	421	27	✓
Late9	Gecode-input-order-indomain-median	3179	503	38	X
Late13	Chuffed-first-fail-indomain-median	11632	539	47655	X
Late14	or-tools-input-order-indomain-median	3789	215	3228	X
Late15	Gecode-dom-w-deg-indomain-random-restart-luby(10)	13735	13635	16	✓

6.4 Comparison between Versions

From that section on, all experiments are performed on MiniZinc version 2.6.1.

To compare the versions, we redid the previous experiments. We also added or-tools as a possible solver. Again, we only started experiments with restarts on instances where we found solutions without restarts. As we can see in Table 6.6 and Table 6.7, the restart

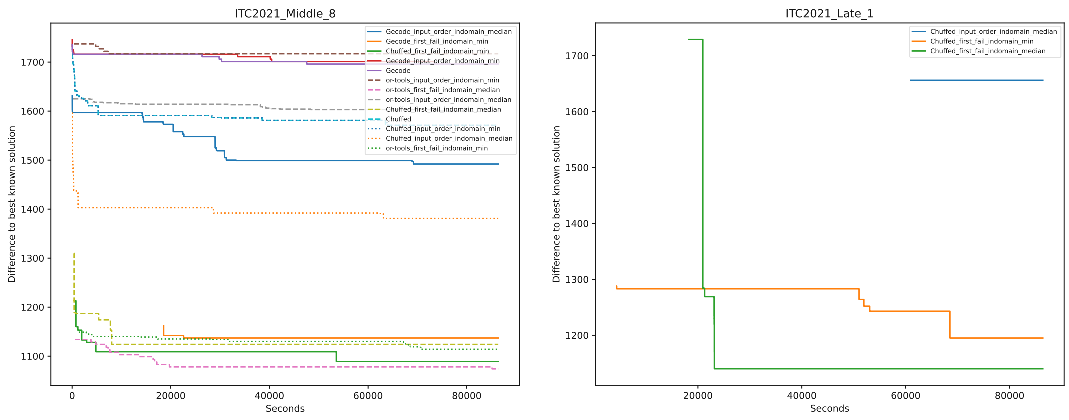

Figure 6.2: Left: Minizinc runs that found solutions and their improvements over time for Middle 8. Right: The same for Late 1.

runs often lead to the best solution; however, there are also a lot of instances where non-restart runs find a solution and restart runs do not, which can be seen in the last column of the tables, which indicate if the given instance was solved by a restart run. For the runs where we do not find a solution with restart runs, a reason for this might be that the restarting is performed prematurely. In general, Gecode is often the fastest to find a solution; however, Chuffed finds solutions to instances that or-tools and Gecode cannot find, which makes it preferable in our understanding.

In Figure 6.2 (left) we can see an anytime plot of the runs for Middle 8 for which all solvers found a solution. The y-axis shows the difference to the best known solutions; the x-axis shows the time. When no further definition apart from the solver is given for the configuration, it means that we omitted to define a variable and value choice manually. On the other hand, in Figure 6.2 (right), we can see the runs for Late 1 for which only Chuffed found solutions.

6.5 Globals vs Non-globals

In this section, we describe the global constraints and compare our results of experiments that used global constraints and those which did not. The global constraints of MiniZinc are high-level abstractions for which the different solvers offer efficient implementations [41, 4.2.2].

We experimented with the constraints $all\_different(array : x)$ and $global\_cardinality$ $(array : x, array : cover, array : count)$. The first one guarantees that all elements of the given array x are different. For the second one x is the array in which we want to count elements, $cover$ is the array with the elements that should be found in x and $count$ is the array for the number of times an element of cover should be in x. For example, $cover[i]$ should appear $count[i]$ times in x.

In order to use these functions one needs to include them in the model file. Then we could use them for the core constraints of the model. The altered model with the adapted core constraints with respect to the original version (see Listing 5.2) can be seen in Listing 6.2. The constraint that ensures that teams do not play against themselves as well as the opponent and the venue connection constraints are left as they were. The constraint that each team plays against each other team twice, once at home and once on the road, is changed such that all games of each team need to be different (alldifferent). To distinguish home games from away games, we multiply the opponent by the venue (-1 or 1).

The last constraint that every team needs to play exactly twice against all other teams or, for phased instances, every team needs to play exactly once against all other teams in each SRR phase of the tournament, is implemented using *global_cardinality*. In Listing 6.2, we see a phased tournament, therefore, we have two *global_cardinality* constraints, one for each phase. For x we build an array with all the opponents of a team i in the rounds 1 to the $R/2$ and from $R/2$ to R; for *cover* we take all teams except i and finally *count* is an array consisting of 1s with the same length as *cover*.

Listing 6.2: Altered core constraints of the model using global constraints

```
% team does not play against itself
constraint
    forall (i in TEAMS, r in SLOTS) (
        opponents_per_team_and_round[i, r] != i
    );

% all games (opponent+venue) per team must be unique
constraint
    forall (i in TEAMS) (
        all_different(
            [venue_per_team_and_round[i,r] *
             opponents_per_team_and_round[i,r] | r in SLOTS]
        )
    );


% opponent connection constraint
constraint
    forall (i in TEAMS, r in SLOTS) (
        opponents_per_team_and_round[opponents_per_team_and_round[i,r],r]
          == i
    );
% venue connection constraint
constraint
    forall (i in TEAMS, r in SLOTS) (
        venue_per_team_and_round[opponents_per_team_and_round[i,r],r] ==
            -venue_per_team_and_round[i,r]
    );

% every team must play exactly once against every other team in
% each phase (SRR) of the schedule
```

```
constraint
    forall (i in TEAMS) (
        global_cardinality(
            [opponents_per_team_and_round[i,r] | r in SLOTS1],
            [j | j in TEAMS where j != i],
            [1 | j in TEAMS where j != i]
        ) /\
        global_cardinality(
            [opponents_per_team_and_round[i,r] | r in SLOTS2],
            [j | j in TEAMS where j != i],
            [1 | j in TEAMS where j != i]));
```

We tested these constraints with Gecode, Chuffed and or-tools with *first_fail* and *input_order* as the variable choice heuristics and *indomain_min* and *indomain_median* as the value choice heuristics on the Early instances for 24 hours and compared them to the approach when not using global constraints. In these experiments we ignored the soft constraints by leaving them out of the model since we wanted to focus on finding a feasible solution. Thereby, we converted the instances to ITC2021-NSC instances.

In Table 6.8, we can see a comparison of the runtimes of the different configurations. The columns equal the different Early instances from 1 to 15, the rows show the different configurations (G - globals, NG - non globals, io - input order, ff - first fail). When the process did not find a solution for an instance it is shown with a dash in the table. As we can see, there is no clear trend observable which of the two options performs better. Sometimes a global run is faster while other times a non global run is faster.

We also summed up the runtimes to compare them (see Table 6.9) and ranked the configurations according to the sum. The runtime of unsolved instances is set to 86400 to penalize configurations that solved fewer instances than other configurations but in a faster time. In the last column, we can see a ranking of the configurations based on the sum of the runtimes.

In Table 6.10, we ranked the configurations for each instance separately. To penalize the last place (which is taken by configurations that do not find a solution), ties are assigned the highest rank, so the configurations that have not found a solution are assigned rank 24. Otherwise, we would have the ranks 1, 2 and 3, where 3 would be the ranking of all configurations that did not find a solution. In the two rightmost columns we can see the sum of the assigned ranks for each configuration and a ranking based on that sum.

In both cases Chuffed without globals with *first_fail* and *indomain_min* is the best configuration, as it solved seven configurations. The same configuration with *indomain_median* is second regarding time and fourth regarding the rankings. Second place regarding the rankings takes Chuffed without globals with *input_order* and *indomain_median*. Chuffed with globals, *first_fail* and *indomain_median* was fourth best regarding the runtime and third regarding the ranking. The worst four rankings are taken by Gecode using *input_order* as the variable choice heuristics, regardless of using globals or not and *indomain_min* or *indomain_median*.

Table 6.8: Comparison of the runtimes of different configurations on the Early instances.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Gecode G-ff-median	-	-	-	-	-	-	-	**0.1**	**0.1**	-	-	-	-	**0.2**	-
Gecode G-ff-min	-	-	**0.1**	-	-	-	-	**0.1**	**0.1**	-	-	-	-	**0.2**	-
Gecode G-io-median	-	-	-	-	-	-	-	-	**0.1**	-	-	-	-	-	-
Gecode G-io-min	-	-	121.0	-	-	-	-	59.7	-	-	-	-	-	-	-
Gecode NG-ff-median	-	-	-	-	-	-	-	0.2	0.2	-	-	-	-	0.3	-
Gecode NG-ff-min	-	-	**0.1**	-	-	-	-	0.2	-	-	-	-	-	0.5	-
Gecode NG-io-median	-	-	-	-	-	-	-	-	0.2	-	-	-	-	-	-
Gecode NG-io-min	-	-	166.0	-	-	-	-	57.3	-	-	-	-	-	-	-
Chuffed G-ff-median	-	-	0.2	-	-	-	-	0.2	0.9	-	-	530.3	-	1.5	786.1
Chuffed G-ff-min	-	-	0.2	-	-	46257.3	-	**0.1**	0.3	-	-	-	-	0.3	-
Chuffed G-io-median	-	-	0.3	-	-	-	-	**0.1**	**0.1**	-	-	129.9	-	0.5	-
Chuffed G-io-min	-	-	0.3	-	-	-	-	2.2	0.4	-	-	121.7	-	-	-
Chuffed NG-ff-median	-	-	0.6	-	-	-	-	0.6	1.3	-	-	638.7	-	2.7	266.0
Chuffed NG-ff-min	-	-	0.2	-	-	**19011.9**	-	**0.1**	0.2	-	-	47414.1	-	0.9	**221.5**
Chuffed NG-io-median	-	-	0.3	-	-	-	-	**0.1**	**0.1**	-	-	212.3	-	2.6	724.4
Chuffed NG-io-min	-	-	0.2	-	-	-	-	5.6	1.1	-	-	**106.9**	-	-	-
or G-ff-median	-	-	57.1	-	-	-	-	141.4	135.5	-	-	-	-	282.8	-
or G-ff-min	-	-	46.1	-	-	-	-	128.6	226.4	-	-	-	-	419.8	-
or G-io-median	-	-	79.0	-	-	-	-	96.7	103.0	-	-	22340.1	-	232.4	-
or G-io-min	-	-	44.3	-	-	-	-	95.8	142.8	-	-	9171.5	-	179.6	-
or NG-ff-median	-	-	64.9	-	-	-	-	126.5	-	-	-	-	-	356.5	-
or NG-ff-min	-	-	29.9	-	-	-	-	109.1	140.0	-	-	-	-	2443.6	-
or NG-io-median	-	-	104.5	-	-	-	-	103.4	97.1	-	-	26912.2	-	277.2	-
or NG-io-min	-	-	70.1	-	-	-	-	98.9	382.9	-	-	13971.5	-	115.6	-

Our conclusion of that experiment is that Chuffed looks most promising as it has the best performance with and without globals. More details on these runs can be found in the Appendix (Table A.1 to A.12).

The break constraints seemed to be the constraints that made it the hardest to solve the problem instances. Since Chuffed with $first\_fail$ looked the most promising in the previous experiments, we tested with Chuffed ignoring the break constraints and using globals as well as non-globals. The results can be seen in Table A.13 and A.14. By ignoring the break constraints we were able to solve twelve of 15 instances regarding the remaining constraints. Surprisingly, the configuration that did not use global constraints and used *indomain_min* was most successful, solving twelve instances. The configuration with *indomain_median* solved only ten instances, the same amount was also solved for the same configurations but using global constraints. Therefore, the non-globals approach looked more promising to us.

Regardless of which value choice was the best, the best variable choice was clearly $first\_fail$. However, to alter the results we needed some sort of randomness to the process. Unfortunately, Chuffed does not provide a default option to use *indomain_random* for choosing a value to assign to a variable. Therefore, we implemented our own randomness for value choice for Chuffed as already described in Section 6.1.3. The experiments with our own randomization implementation for Chuffed are described in the next section, Section 6.6.

Table 6.9: Comparison of the runtimes of different configurations on the Early instances where the unsolved runs are shown with the maximum time. On the right, the times are summed up and their ranking according to that runtime sum is shown.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Sum	Rank
GecodeG-ff-median	86400	86400	86400	86400	86400	86400	86400	0.1	0.1	86400	86400	86400	86400	0.2	86400	1036800.4	17
GecodeG-ff-min	86400	86400	0.1	86400	86400	86400	86400	0.1	0.1	86400	86400	86400	86400	0.2	86400	950400.5	11
GecodeG-io-median	86400	86400	86400	86400	86400	86400	86400	86400	0.1	86400	86400	86400	86400	86400	86400	1209600.1	23
GecodeG-io-min	86400	86400	121.0	86400	86400	86400	86400	59.7	86400	86400	86400	86400	86400	86400	86400	1123380.7	21
GecodeNG-ff-median	86400	86400	86400	86400	86400	86400	86400	0.2	0.2	86400	86400	86400	86400	0.3	86400	1036800.7	18
GecodeNG-ff-min	86400	86400	0.1	86400	86400	86400	86400	0.2	86400	86400	86400	86400	86400	0.5	86400	1036800.8	19
GecodeNG-io-median	86400	86400	86400	86400	86400	86400	86400	86400	0.2	86400	86400	86400	86400	86400	86400	1209600.2	24
GecodeNG-io-min	86400	86400	166.0	86400	86400	86400	86400	57.3	86400	86400	86400	86400	86400	86400	86400	1123423.3	22
ChuffedG-ff-median	86400	86400	0.2	86400	86400	86400	86400	0.2	0.9	86400	86400	530.3	86400	1.5	786.1	778919.2	4
ChuffedG-ff-min	86400	86400	0.2	86400	86400	46257.3	86400	0.1	0.3	86400	86400	86400	86400	0.3	86400	910258.2	10
ChuffedG-io-median	86400	86400	0.3	86400	86400	86400	86400	0.1	0.1	86400	86400	129.9	86400	0.5	86400	864130.9	5
ChuffedG-io-min	86400	86400	0.3	86400	86400	86400	86400	2.2	0.4	86400	86400	121.7	86400	86400	86400	950524.6	13
ChuffedNG-ff-median	86400	86400	0.6	86400	86400	86400	86400	0.6	1.3	86400	86400	638.7	86400	2.7	266.0	778509.9	2
ChuffedNG-ff-min	86400	86400	0.2	86400	86400	19011.9	86400	0.1	0.2	86400	86400	47414.1	86400	0.9	221.5	757848.9	1
ChuffedNG-io-median	86400	86400	0.3	86400	86400	86400	86400	0.1	0.1	86400	86400	212.3	86400	2.6	724.4	778539.8	3
ChuffedNG-io-min	86400	86400	0.2	86400	86400	86400	86400	5.6	1.1	86400	86400	106.9	86400	86400	86400	950513.8	12
orG-ff-median	86400	86400	57.1	86400	86400	86400	86400	141.4	135.5	86400	86400	86400	86400	282.8	86400	951016.8	14
orG-ff-min	86400	86400	46.1	86400	86400	86400	86400	128.6	226.4	86400	86400	86400	86400	419.8	86400	951220.9	15
orG-io-median	86400	86400	79.0	86400	86400	86400	86400	96.7	103.0	86400	86400	22340.1	86400	232.4	86400	886851.2	8
orG-io-min	86400	86400	44.3	86400	86400	86400	86400	95.8	142.8	86400	86400	9171.5	86400	179.6	86400	873634.0	6
orNG-ff-median	86400	86400	64.9	86400	86400	86400	86400	126.5	86400	86400	86400	86400	86400	356.5	86400	1037347.9	20
orNG-ff-min	86400	86400	29.9	86400	86400	86400	86400	109.1	140.0	86400	86400	86400	86400	2443.6	86400	953122.6	16
orNG-io-median	86400	86400	104.5	86400	86400	86400	86400	103.4	97.1	86400	86400	26912.2	86400	277.2	86400	891494.4	9
orNG-io-min	86400	86400	70.1	86400	86400	86400	86400	98.9	382.9	86400	86400	13971.5	86400	115.6	86400	878639.0	7

Table 6.10: Ranking of the runtimes of the different configurations on the Early instances. Ties are assigned the maximum rank. On the right, the sum of the ranks is shown and a rank for each configuration based on that sum.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Sum	Rank
GecodeG-ff-median	24	24	24	24	24	24	24	6	5	24	24	24	24	2	24	301	10
GecodeG-ff-min	24	24	2	24	24	24	24	6	5	24	24	24	24	2	24	279	7
GecodeG-io-median	24	24	24	24	24	24	24	24	5	24	24	24	24	24	24	341	21
GecodeG-io-min	24	24	19	24	24	24	24	14	24	24	24	24	24	24	24	345	23
GecodeNG-ff-median	24	24	24	24	24	24	24	9	8	24	24	24	24	4	24	309	13
GecodeNG-ff-min	24	24	2	24	24	24	24	9	24	24	24	24	24	6	24	305	12
GecodeNG-io-median	24	24	24	24	24	24	24	24	8	24	24	24	24	24	24	344	22
GecodeNG-io-min	24	24	20	24	24	24	24	13	24	24	24	24	24	24	24	345	23
ChuffedG-ff-median	24	24	6	24	24	24	24	9	11	24	24	5	24	8	4	259	3
ChuffedG-ff-min	24	24	6	24	24	2	24	6	9	24	24	24	24	4	24	267	4
ChuffedG-io-median	24	24	9	24	24	24	24	6	5	24	24	3	24	6	24	269	6
ChuffedG-io-min	24	24	9	24	24	24	24	11	10	24	24	2	24	24	24	296	9
ChuffedNG-ff-median	24	24	10	24	24	24	24	10	13	24	24	6	24	10	2	267	4
ChuffedNG-ff-min	24	24	6	24	24	1	24	6	8	24	24	11	24	7	1	232	1
ChuffedNG-io-median	24	24	9	24	24	24	24	6	5	24	24	4	24	9	3	252	2
ChuffedNG-io-min	24	24	6	24	24	24	24	12	12	24	24	1	24	24	24	295	8
orG-ff-median	24	24	14	24	24	24	24	22	16	24	24	24	24	15	24	331	18
orG-ff-min	24	24	13	24	24	24	24	21	19	24	24	24	24	17	24	334	19
orG-io-median	24	24	17	24	24	24	24	16	15	24	24	9	24	13	24	310	14
orG-io-min	24	24	12	24	24	24	24	15	18	24	24	7	24	12	24	304	11
orNG-ff-median	24	24	15	24	24	24	24	20	24	24	24	24	24	16	24	339	20
orNG-ff-min	24	24	11	24	24	24	24	19	17	24	24	24	24	18	24	329	17
orNG-io-median	24	24	18	24	24	24	24	18	14	24	24	10	24	14	24	314	16
orNG-io-min	24	24	16	24	24	24	24	17	20	24	24	8	24	11	24	312	15

6.6 Randomized Restart Experiments for Chuffed

To test our random value choice functionality for Chuffed, we tested it with *first_fail*, *input_order* and *random_order* only using a runtime of two hours here to avoid the long runs of the heavy-tailed distribution. We also left out the soft constraints again. For each configuration we ran 60 test runs. In Table 6.11, we can see the results of that

Table 6.11: 2h experiments for own randomization for Chuffed with no soft constraints. Each of the three configurations was tested 60 times. For each variable choice heuristics we show the minimum solve time, maximum solve time and the percentiles of the solve time.

	first-fail							input-order							random-order						
	min	*max*	10	25	50	75	90	*min*	*max*	10	25	50	75	90	*min*	*max*	10	25	50	75	90
Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early2	-	-	-	-	-	-	-	84.03	649.7	7200	7200	7200	7200	7200	-	-	-	-	-	-	-
Early3	0.06	0.62	0.09	0.11	0.14	0.17	0.24	0.05	5.66	0.1	0.13	0.17	0.28	0.64	0.08	3.24	0.34	0.57	1.12	1.62	2.07
Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early6	880.25	6682.74	3998.86	7200	7200	7200	7200	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early8	0.07	4794.49	0.09	0.11	0.2	1.17	9.63	0.09	184.02	0.1	0.11	0.23	5.27	31.21	0.08	0.37	0.1	0.11	0.14	0.16	0.21
Early9	0.1	10.28	0.11	0.14	0.26	0.42	1.32	0.1	19.43	0.12	0.16	0.3	0.94	6.32	0.1	1.08	0.15	0.21	0.31	0.49	0.61
Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early12	80.13	6522.28	996.88	1971.29	7200	7200	7200	38.02	6528.67	75.34	288.17	3132.04	7200	7200	-	-	-	-	-	-	-
Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Early14	0.16	1426.6	0.28	0.36	0.56	1.84	17.45	0.34	3251.55	0.86	4.32	432.79	7200	7200	0.18	1.31	0.24	0.33	0.51	0.62	0.81
Early15	112.73	4075.37	222.47	356.19	530.14	991.31	2558.41	-	-	-	-	-	-	-	-	-	-	-	-	-	-

experiment. It shows the minimum and maximum runtime for finding a feasible solution as well as several percentiles of the runs. When a run does not find a solution, it is counted as 7200 seconds for calculating the percentiles.

The variable choice *first_fail* finds the most feasible solutions after two hours and looking at the percentile is also more consistent in finding solutions, which is why we will use this configuration for further experiments although *input_order* is the only configuration that finds a solution for Early 2, but only in two of the test runs, which is why we choose *first_fail*. The variable choice *random_order* does not perform well in such short experiments; randomizing both variable choice and value choice seems to be too unguided.

6.7 Preliminary Experiments with TTSA

We implemented the classic Traveling Tournament Simulated Annealing (TTSA) approach by Anagnostopoulos et al. [1] in Julia and reran experiments on the TTP NL instances[2] as a sanity check since it serves as our basis for the ITC2021 SA variant with its different objective function and constraints. The algorithm is local search based and uses the five neighborhood structures (see Section 5.2.1) to perform a guided random walk through the solution space of double round robin tournaments. During that, it permits constraint violations and uses strategic oscillation to traverse infeasible regions of the search space. Its highly parallelized variant by Van Hentenryck and Vergados [53] is responsible for many best feasible solutions found so far over the benchmark instances. A version of the following study of TTSA for the TTP has also been published in one of our related works in the context of the comparison with a randomized beam search approach to the TTP [15].

[2] https://mat.tepper.cmu.edu/TOURN/

Table 6.12: Parameters used for our TTSA(FC) experiments with a time limit of two hours. The parameters for NL16 are taken from the fast cooling experiment of [1], the parameters for the other instances were changed accordingly based on the parameters of their full run experiments.

n	T_0	β	w_0	δ	θ	$maxC$	$maxP$	$maxR$	γ
8	400	0.98	4000	1.04	1.04	2500	70	2000	2
10	400	0.98	6000	1.04	1.04	2500	70	2000	2
12	600	0.98	10000	1.03	1.03	2000	14	10000	1.6
14	600	0.98	20000	1.03	1.03	2000	70	6000	1.8
16	700	0.98	60000	1.05	1.05	5000	70	10000	2

After preliminary experiments, we made the following implementation detail choices, which are not specified in the original paper: We add a penalty proportional to the streak limit excess, to penalize longer streaks more strongly. For example, a stand of five home games counts as two violations since it is two games longer than the maximum length of $U = 3$. Moreover, violations of the no-repeat constraint are counted team-wise, therefore, we count a no-repeat constraint violation twice. Anagnostopoulos et al. [1] use five neighborhoods: Swap Homes, Swap Rounds, Swap Teams, Partial Swap Rounds and Partial Swap Teams. In our implementation each of those neighborhoods is equally likely to be chosen when generating the next neighbor. We managed to obtain a moderate decrease in runtime using incremental evaluation for calculating changes of the objective value for the more round-local moves, but found to achieve this rather difficult, likely due to the rather small schedule size $n \times (2n - 2)$ and that we did not consider large n for the TTP.

To generate an initial double round tournament (possibly with constraint violations), a randomized backtracking algorithm is used in the original paper, which constructs the schedule team-wise. It is observed that this does not scale well starting already from 16 teams with runtime outliers in the range of hours. As used in the ant colony optimization approach by [46], we restart the construction in a Las Vegas algorithm fashion after a backtracking limit is hit, which we set empirically to 30 000. This allows us to construct random double round robin tournaments even up to 20 teams below one minute. A faster method can be found in the Appendix (Figure A.1), where Gecode with restarting and global constraints (Listing A.1) allows us to construct random feasible DRR and TTP schedules within at most a couple of seconds for up to 40 teams.

There are nine parameters for the algorithm that can be set: T_0 defines the initial temperature for the SA, β declares the cooling rate, w_0 is the initial weight with which constraint violations are penalized, δ and θ define how the weight changes when a new best feasible or a new best infeasible schedule is found. The number of iterations until equilibrium is reached is specified by $maxC$, the number of phases by $maxP$, and the number of reheats by $maxR$. While γ was not explicitly defined in the paper, we reasonably assumed it to be the reheating factor which is the factor by which the

Table 6.13: Results of 30 TTSA runs with fast-cooling and a time limit of two hours on the NL instances.

instance	min	max	avg	std
NL8	39721	39721	39721.0	0.0
NL10	59583	60769	60113.9	346.0
NL12	114920	130829	120651.4	2916.0
NL14	195144	201506	198947.0	1363.9
NL16	279226	287312	284137.1	2209.1

temperature is multiplied when reheating.

We performed 30 TTSA runs using fast cooling, which is as also described by Anagnostopoulos et al. [1], to quickly generate good solutions by faster intensification of the search and more reheats. The used parameters are shown in Table 6.12, together with a fixed time limit of two hours; the results are shown in Table 6.13. These results compare well with [1], leading to the conclusion that we have derived a sound TTSA implementation to be suitably modified further for the ITC2021 problem.

6.8 First Hybridization with Simulated Annealing

We subsequently tested our hybridization approach, combining CP and SA still following TTSA's algorithmic pattern, but with the corresponding objective functions and constraints adapted for the ITC2021 instances instead of the ones for the TTP. As a starting point, we took the parameter sets for the TTP instances with 16 teams [1], for fast cooling ($\beta = 0.98$) and their long runs with slow cooling ($\beta = 0.9999$) respectively, see Table 6.14. For the CP part we used Chuffed as the solver, $first\_fail$ as the variable choice and $indomain\_min$ with our own randomization for Chuffed for the value choice as it looked most promising during prior experiments and to bring randomness into the procedure.

We tested the approach with four threads, which ran for 24 hours each. For the MiniZinc runs all soft constraints are ignored to speed up the search for an initial solution as then the solver does not need to calculate the objective value. Thereby, the instances used become ITC2021-NSC instances. For each thread the first stage of the CP was given a one-hour time limit, the second stage of CP, where we dropped the break constraints, converting the instances to ITC2021-NSC-NBC instances, was given a 30-minute time limit and for the last stage, which uses only the DRR constraints, the process was given five minutes. Whenever a solution was found, the SA took over and used up the rest of the time up to 24 hours. The probability of the neighborhoods to be chosen was defined according to their number of possible moves, i.e., $SwapHomes$ was less likely than $PartialSwapTeams$ as the latter has much more possible moves.

For the Late instances we passed on the classic TTSA and stuck with the fast cooling

Table 6.14: parameters of the different SA configurations that are tested in this section

config	T	β	w	δ	θ	$maxC$	$maxP$	$maxC$	γ	MV
TTSA-FC	700	0.98	60000	1.05	1.05	5000	70	10000	2	X
TTSA-FC-MV	700	0.98	60000	1.05	1.05	5000	70	10000	2	✓
TTSA	700	0.9999	60000	1.05	1.05	10000	7100	50	2	X
TTSA-MV	700	0.9999	60000	1.05	1.05	10000	7100	50	2	✓

Table 6.15: Hybridization experiments with four threads, *avg* shows the mean objective value of that configuration, *min* shows the best objective value and % shows the percentage gap of the best solution found with that configuration to the best known solution. *FC* means fast cooling, *MV* means minimize violations; where the best known solution is 0 we calculated the gap using an objective value of 1.

	TTSA-FC				TTSA-FC-MV				TTSA				TTSA-MV				known sol
	avg	CP sol	min	%	avg	CP sol	min	%	avg	CP sol	min	%	avg	CP sol	min	%	
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	362
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	145
ITC2021Early3	1282.25	**4590**	**1239**	**24.90**	1410.00	4616	1272	28.23	2950.75	5609	2927	195.06	2930.75	5473	2713	173.49	992
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	507
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3127
ITC2021Early6	-	-	-	-	5700.00	**5700**	**5700**	**71.43**	-	-	-	-	-	-	-	-	3325
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4763
ITC2021Early8	3103.75	5285	3047	189.91	3537.33	**5728**	**2902**	**176.12**	5132.25	5593	5087	384.02	4903.00	4684	4684	345.67	1051
ITC2021Early9	554.00	3982	518	825.00	496.75	**4078**	**443**	**691.07**	2958.75	4588	2868	5021.43	3015.25	3988	2998	5253.57	56
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3400
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4426
ITC2021Early12	2120.00	2120	2120	573.02	-	-	-	-	1911.00	1911	1911	506.67	1900.00	**1900**	**1900**	**503.17**	315
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	121
ITC2021Early14	144.50	**5628**	**105**	**2525.00**	130.50	5894	105	2525.00	3358.00	6142	3305	82525.00	3351.25	6381	3177	79325.00	4
ITC2021Early15	7007.75	6748	6748	128.36	6818.67	6577	6577	122.57	6917.00	6818	6818	130.73	6789.25	**6543**	**6543**	**121.42**	2955
ITC2021Middle1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	5177
ITC2021Middle2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	7381
ITC2021Middle3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	9542
ITC2021Middle4	332.50	**295**	**295**	**4114.29**	354.50	337	337	4714.29	364.00	353	353	4942.86	344.75	323	323	4514.29	7
ITC2021Middle5	834.25	5164	810	190.32	801.75	**5621**	**775**	**177.78**	2595.25	4704	2494	793.91	2642.75	5477	2524	804.66	279
ITC2021Middle6	3433.75	**3110**	**3110**	**177.68**	3407.50	3385	3385	202.23	3450.00	3315	3315	195.98	3426.25	3175	3175	183.48	1120
ITC2021Middle7	9179.75	8285	8285	364.67	9338.75	**8157**	**8157**	**357.49**	9005.25	8413	8413	371.85	9496.00	8519	8519	377.79	1783
ITC2021Middle8	549.00	**1561**	**374**	**189.92**	397.00	1473	388	200.78	1055.00	1515	1032	700.00	1049.75	1521	1039	705.43	129
ITC2021Middle9	3278.33	**3125**	**3125**	**653.01**	3401.25	3160	3160	661.45	3312.50	3135	3135	655.42	3360.00	3230	3230	678.31	415
ITC2021Middle10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1250
ITC2021Middle11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2446
ITC2021Middle12	1530.25	5140	1443	140.90	1499.50	**5128**	**1431**	**138.90**	3338.75	5154	3326	455.26	3365.00	4836	3286	448.58	599
ITC2021Middle13	8509.50	7532	7532	2888.89	8973.00	7802	7802	2996.03	8537.00	**7442**	**7442**	**2853.17**	9041.75	7993	7993	3071.83	252
ITC2021Middle14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1140
ITC2021Middle15	1469.75	12164	1394	187.42	1417.75	**12211**	**1338**	**175.88**	6247.75	13557	6190	1176.29	6306.00	13234	6275	1193.81	485
ITC2021Late1	3424.50	**3160**	**3160**	**64.41**	3450.00	3263	3263	69.77	-	-	-	-	-	-	-	-	1922
ITC2021Late2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	5400
ITC2021Late3	3420.00	**10209**	**3249**	**37.15**	3450.25	9099	3313	39.85	-	-	-	-	-	-	-	-	2369
ITC2021Late4	1217.00	1086	1086	108500.00	1180.25	**1004**	**1004**	**100300.00**	-	-	-	-	-	-	-	-	0
ITC2021Late5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1923
ITC2021Late6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	923
ITC2021Late7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1558
ITC2021Late8	1491.50	4514	1427	52.78	1482.75	**4567**	**1420**	**52.03**	-	-	-	-	-	-	-	-	934
ITC2021Late9	1705.00	3243	1643	229.92	1593.00	**3120**	**1149**	**130.72**	-	-	-	-	-	-	-	-	498
ITC2021Late10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1945
ITC2021Late11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	202
ITC2021Late12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3428
ITC2021Late13	11792.25	**10540**	**10540**	**479.12**	11634.00	10702	10702	488.02	-	-	-	-	-	-	-	-	1820
ITC2021Late14	3855.50	**3717**	**3717**	**216.61**	3855.50	3753	3753	219.68	-	-	-	-	-	-	-	-	1174
ITC2021Late15	128.75	**6935**	**115**	**11400.00**	120.00	7155	120	11900.00	-	-	-	-	-	-	-	-	0

TTSA (TTSA-FC and TTSA-FC-MV) as it outperformed the former in the Early and Middle instances.

The results of runs with four threads on the ITC2021 instances are shown in Table 6.15. TTSA was able to improve some of the solutions found by MiniZinc. However, it did not manage to find feasible solutions for the other instances that were not solved to feasibility by CP, i.e., it could not handle infeasible initial solutions effectively.

Table 6.16: Parameters for SA tuning

config	T	β	w	δ & θ	$maxC$	$maxP$	$maxC$	γ	MV	alt
ttsa-FC	{700, 1200}	{0.98, 0.999}	{10, 50, 100}	{ 1.03, 1.05 }	5000	70	10000	2	{X, ✓}	{X, ✓}

6.9 Simulated Annealing Standalone

Tuning the SA with its numerous parameters deserves more attention. Therefore, we tested different configurations of the parameters on the Early instances. Each configuration was tested three times; each test run lasted two hours. The configurations are the Cartesian product of the parameters shown in Table 6.16, which results in 96 different configurations; to the already known parameters, we added *alt* whether to use the alternative objective function or not. Moreover, we also altered the SA process to monitor the acceptance rates and reset temperature and weights when it drops to 0 for some time (described in detail in Section 5.2). For the parameters of QLN, $ITER$ and DIV we performed preliminary tests on a few instances and set them to $QLN = 5$, $ITER = 1000$ and $DIV = 4$ for further experiments, as these values looked the most promising.

For example, if we start with a weight of 10, then the weight is increased by lots of infeasible solution improvements (for example up to 150) and afterwards the temperature is so low that the acceptance rate drops to 0. Then we reset the temperature to its initial value and set the weight to 37.5.

We also tried to only adapt the weight if the violations have been down to 0 for once, but first tests showed that this is not beneficial. Furthermore, we tried to reheat the temperature to the initial temperature instead of setting it to the $bestTemperature$ times γ. However, that did not work out in initial test runs either. The goal was to first find configuration that brings most of the instances to feasibility from an initial infeasible solution.

In Figure 6.3, we can see the distribution of the runs that found a feasible solution in the two-hour time frame. The labels of the bars show the value of one of the variables of the configuration. All of our tested parameters performed similarly, except for the value $\beta = 0.98$, the fast cooling regime, which clearly outperformed $\beta = 0.999$—not too surprisingly as the latter is designed for longer runs, and intensification is reached later.

In Table 6.17, we can see the sorted results of the configurations. The column $\#sols$ shows how many solutions were found with that configuration. Each configuration was used in 45 runs (3 runs per instance and configuration over 15 instances). When we look at the temperature, it does not matter if we start with 700 or 1200. For β it is essential to use a lower cooling factor. For the weight factor it appears to be beneficial to use a factor on the higher end. For β and θ it does not make a difference if one uses 1.03 or 1.05. For MV and *alt* it appears to be slightly beneficial to use both with *true* as the first five entries were all found with that configuration. In Table 6.18 we can see the best results of our first SA experiments.

To test the ability of our SA to further improve the solutions when starting with a feasible

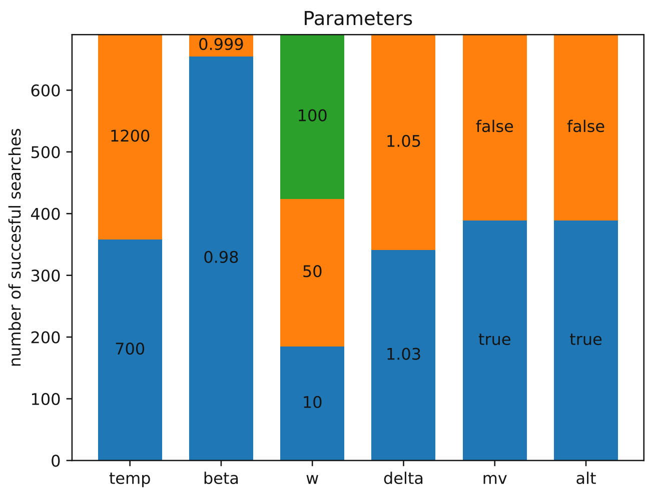

Figure 6.3: Distribution of the successful runs regarding the parameters.

Table 6.17: Sorted configurations of the SA runs, shown are best and worst configurations $\#sols$ is how many solutions were found, F_{ks} is the average factor our solutions are worse than the best known solution.

T	β	w	δ, θ	MV	alt	$\#sols$	F_{ks}	$\#insts$
700.0	0.98	10.0	1.05	true	true	18	22.31	7
1200.0	0.98	100.0	1.05	true	true	18	24.48	6
700.0	0.98	50.0	1.03	true	true	17	21.04	7
700.0	0.98	100.0	1.03	true	true	17	25.14	6
1200.0	0.98	100.0	1.03	true	true	17	28.47	6
700.0	0.98	100.0	1.03	false	false	16	21.51	6
1200.0	0.98	100.0	1.03	false	true	16	23.21	6
700.0	0.98	100.0	1.03	false	true	16	23.82	6
1200.0	0.98	50.0	1.05	true	true	16	27.83	6
700.0	0.98	100.0	1.03	true	false	16	30.80	6
700.0	0.98	50.0	1.05	true	true	16	31.52	6
1200.0	0.98	100.0	1.05	true	false	16	37.41	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
700.0	0.999	100.0	1.03	true	true	4	7.39	3
700.0	0.999	50.0	1.05	true	true	4	8.42	2
700.0	0.999	100.0	1.05	true	false	3	5.50	2
1200.0	0.999	50.0	1.05	true	true	2	6.42	2
1200.0	0.999	50.0	1.03	true	true	1	10.24	1

Table 6.18: Best objective values of each instance where the previous experiment found a solution and the configuration it was found with.

config	T	β	w	δ, θ	MV	alt	obj	known solution
Early1	1200.0	0.98	100.0	1.03	false	true	1195	362
Early2	1200.0	0.98	50.0	1.03	false	false	548	145
Early3	1200.0	0.98	10.0	1.05	false	true	1428	992
Early8	700.0	0.98	10.0	1.03	false	false	2930	1051
Early9	1200.0	0.98	10.0	1.05	false	true	608	56
Early13	700.0	0.98	10.0	1.03	false	true	398	121
Early14	1200.0	0.98	100.0	1.03	false	false	186	4

Table 6.19: Sorted configurations of the SA feasible runs, shown are the best and worst configuration, $\#improve$ is how many initial solutions were improved, F_{is} is the average factor of how much the initial solution was improved.

T	β	w	δ, θ	MV	alt	$\#improve$	F_{is}	$\#insts$
1200	0.98	10	1.05	true	true	12	0.30	4
700	0.98	10	1.03	true	true	11	0.28	4
700	0.98	10	1.05	false	true	11	0.29	4
700	0.98	10	1.03	false	true	11	0.30	4
700	0.98	10	1.05	true	true	10	0.27	4
700	0.98	10	1.05	true	false	10	0.29	4
700	0.98	50	1.05	false	false	10	0.31	4
1200	0.98	50	1.03	false	false	9	0.29	4
1200	0.98	50	1.05	true	false	9	0.29	4
700	0.98	50	1.03	false	false	9	0.29	4
700	0.98	50	1.03	true	false	9	0.36	4
1200	0.98	10	1.03	true	false	9	0.36	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1200	0.999	100	1.03	true	false	1	0.93	1
700	0.98	50	1.03	true	true	1	0.96	1
1200	0.999	100	1.05	true	true	1	0.97	1
700	0.999	50	1.05	false	true	1	0.97	1
700	0.999	50	1.03	false	true	1	0.98	1
1200	0.98	50	1.03	false	true	1	0.98	1
1200	0.98	100	1.05	false	true	1	0.98	1
700	0.999	10	1.05	true	false	1	0.98	1
1200	0.999	10	1.03	false	false	1	0.98	1
700	0.999	10	1.03	true	false	1	0.98	1

Table 6.20: Best objective values of each instance where the previous experiment found an improved solution and the configuration it was found with.

config	T	β	w	δ, θ	MV	alt	obj	known solution
Early3	1200	0.98	50	1.03	false	false	1336	992
Early8	1200	0.98	10	1.03	false	false	2923	1051
Early9	1200	0.98	10	1.05	true	false	598	56
Early14	1200	0.98	10	1.05	false	false	183	4

solution we started it from feasible initial solutions and the same parameters as in the above experiments. However, since we were not able to find initial solutions for all of the Early instances, we only tested it for 10 of the 15 instances. Where possible, we used a solution from a MiniZinc run since on those solutions there was no SA performed yet; however, for those instances we already found a solution with SA but did not with MiniZinc, we used the SA solution. We took the solution from a previous SA run for Early 1, Early 2 and Early 13. For Early 3, Early 6, Early 8, Early 9, Early 12, Early 14 and Early 15 we used a solution from a MiniZinc run.

The results of this experiment can be seen in Table 6.19. The improvement factor F_{is} describes how much the initial solution was improved on average (0.3 means a value of 100 was improved to 30 on average).

In Table 6.20 we can see the instances where the runs of this experiment found improved solutions regarding their initial solution.

Since we were only able to improve 4 of the 10 instances, we decided to rerun the tests but with higher weights. The results of these experiments can be seen in Table 6.21.

Table 6.21: Sorted configurations of the SA feasible runs with higher weights, shown are the best and worst configurations, *#improve* is how many initial solutions were improved, F_{is} is the average factor of how much the initial solution was improved.

T	β	w	δ, θ	MV	alt	*#improve*	F_{is}	*#insts*
700	0.999	10000	1.05	false	false	14	0.88	5
700	0.98	100000	1.03	true	true	13	0.37	5
700	0.98	10000	1.05	true	false	13	0.37	5
700	0.98	10000	1.03	true	false	13	0.40	5
1200	0.98	100000	1.03	false	false	13	0.42	5
700	0.98	100000	1.05	true	true	13	0.42	5
1200	0.98	100000	1.05	true	false	13	0.43	5
700	0.999	100000	1.05	true	false	13	0.83	5
700	0.999	10000	1.03	false	true	13	0.84	5
700	0.999	10000	1.03	true	false	13	0.85	5
700	0.999	10000	1.05	true	false	13	0.87	5
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
700	0.999	1000	1.03	false	true	2	0.95	2
1200	0.999	1000	1.05	true	true	2	0.99	1
1200	0.999	1000	1.05	false	true	2	0.99	1
700	0.999	1000	1.05	true	false	2	0.99	1
700	0.999	1000	1.05	false	true	2	1.00	2
1200	0.98	1000	1.05	true	false	1	0.92	1
1200	0.999	1000	1.03	false	false	1	0.94	1
700	0.999	1000	1.05	false	false	1	0.97	1
1200	0.98	1000	1.03	true	false	1	0.99	1

Table 6.22: Best objective values of each instance where the previous experiment with higher weights found an improved solution and the configuration it was found with.

config	T	β	w	δ, θ	MV	alt	obj	known solution
Early2	700	0.98	100000	1.05	true	true	439.00	145
Early3	1200	0.98	100000	1.03	false	true	1352.00	992
Early8	1200	0.98	10000	1.05	true	false	2929.00	1051
Early9	700	0.98	1000	1.05	false	false	688.00	56
Early14	700	0.98	1000	1.03	false	true	202.00	4

Here we can see that more configurations found improvements than in the previous run in Table 6.19. In Table 6.22, we can see the instances where we found improvements with the higher weights. In contrast to Table 6.20 we found an improved solution for Early 2.

6.10 Second Hybridization with Simulated Annealing

For this approach we chose 16 configurations of SA from the previous Section 6.9 and let the hybridization algorithm randomly choose one of those configurations. We chose the top 5 configurations from Table 6.17, the top 4 from Table 6.19 and the top 7 from Table 6.21, in order to get to feasibility and improve feasible solutions. In this experiment we tested the algorithm with 16 threads for 24 hours. The results can be seen in Table 6.23.

The first column shows the average objective value over all threads that found a solution to that instance and column *neigh* shows the number of examined neighbors. Column

Table 6.23: Hybridization with 16 threads and the best configurations from the SA tuning. avg: the average objective value over all threads that found a solution, neigh: the number of neighbors examined by the SA, config: the configuration that found the best solution, status: how that solution was found, best: the objective value of the best solution, t[s]: time to best solution, F_{ks}: factor of our best solution to the best known solution, F to [39] and [51]: factor of our best solution to the solutions of [39] and [51].

	avg	neigh	config	status	best	t_{best} [s]	F_{ks}	F to [39]	F to [51]
Early1	1459.14	119824555.38	700_0.98_10_1.05_false_true	SA	1098	70292	3.03	2.60	1.77
Early2	599.00	38233262.62	700_0.98_100000_1.05_true_true	SA	599	63146	4.13	1.88	1.62
Early3	3519.62	30476488.06	700_0.98_10_1.03_false_true	impr	1302	85654	1.31	1.22	1.07
Early4	-	69688455.00	-	-	-	-	-	-	-
Early5	-	15570032.81	-	-	-	-	-	-	-
Early6	5605.00	21810504.56	-	CP	5432	2630	1.63	1.38	1.16
Early7	-	17032941.00	-	-	-	-	-	-	-
Early8	4617.75	24092877.12	700_0.98_10_1.05_true_true	impr	2837	55269	2.70	2.70	1.79
Early9	1957.50	44385678.56	700_0.98_10000_1.05_true_false	impr	453	52447	8.09	3.43	2.20
Early10	-	14885097.69	-	-	-	-	-	-	-
Early11	-	16374088.38	-	-	-	-	-	-	-
Early12	2081.67	60414727.50	-	CP	2020	3468	6.41	2.00	1.79
Early13	773.67	36798657.62	1200_0.98_100000_1.03_false_false	SA	483	11287	3.99	2.79	1.30
Early14	1608.81	75274727.75	1200_0.98_10_1.05_true_true	impr	105	48447	26.25	1.67	4.38
Early15	6893.20	21883620.88	-	CP	6534	357	2.21	1.84	1.37

config shows the configuration that was used for SA for the best run. The column *status* tells us how the solution was found: SA means we found a feasible solution only in the SA phase, impr means we found a feasible solution with CP and improved it with SA, CP means we found a solution with CP and did not further improve it with SA. We can find all three cases in the table, which shows that the hybridization is preferable to using only one component. Column *best* shows the best found solution, *t [s]* shows the time in seconds when the best solution was found, F_{ks} shows the ratio to the best known solution (e.g., 1.0 corresponds to the same quality, 2.0 would mean our approach returns an objective value twice as bad), *F to [39]* shows the ratio to the best found solution by Rosati et al. who used a SA-only approach and *F to [51]* shows the ratio to the best found solution by van Doornmalen et al. who used a hybrid approach combining mixed integer programming and local search in a fix-and-optimize fashion.

As we can see, we are not able to compete with the best known solutions and the paper of Rosati et al. [39]; however, we came closer to the results of van Doornmalen et al. [51]. On the other hand, they managed to find solutions to instances Early 7 and Early 11, where we did not find solutions.

Exemplarily, we show the improvements over the iterations of Early 3 and Early 13 in Figure 6.4. For Early 3, we can see that all of the 16 threads found a solution but only eight of them were improved by SA. For Early 13, we can see that CP did not find a feasible solution and SA found six of them.

In previous experiments, to choose a move for the SA we used a probability based on the number of possible moves in that neighborhood. Therefore, SwapHomes was chosen far less than, for example, PartialSwapRounds since the prior has far less possible moves.

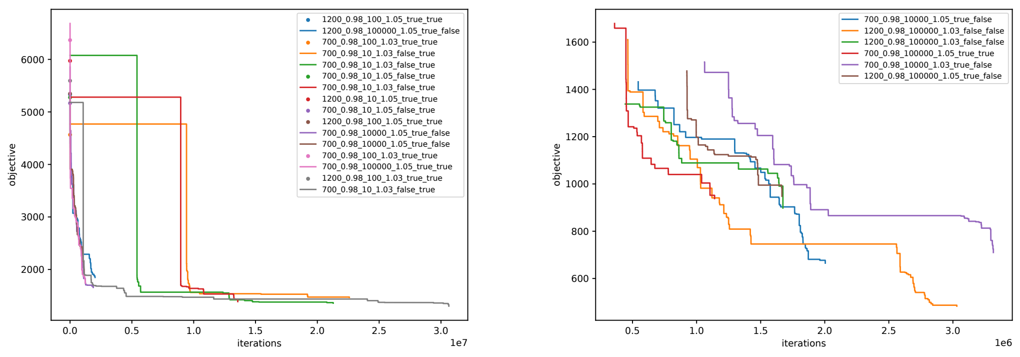

Figure 6.4: Objective values over the iterations of the second hybridization. Left: Early 3, right: Early 13.

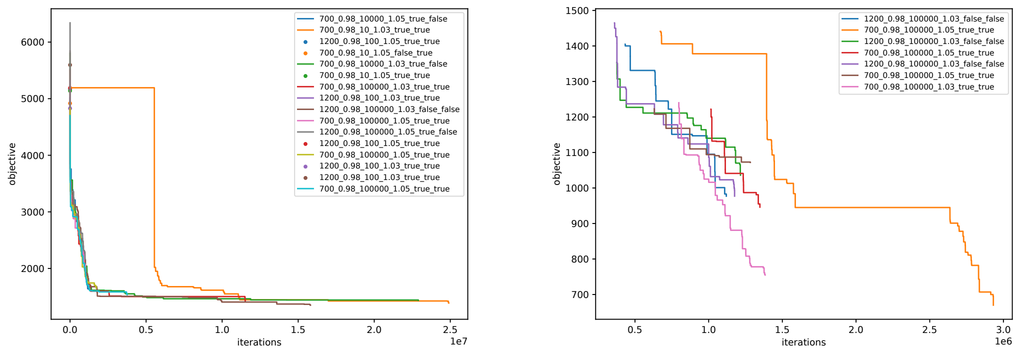

Figure 6.5: Objective values over the iterations of the hybridization with equal neighborhood probability. Left: Early 3, right: Early 13.

For our next experiment we changed the probability of the neighborhoods to be equal, so each of the five neighborhoods is equally likely to be chosen. As the experiments for Early had better average objective values and the SA found more improvements, we continued with experiments for Middle and Late with the same configuration for the neighborhood choice as well.

In Table 6.24, we can see the best results of those experiments. The same columns are used as in Table 6.23, except that for Middle and Late instances there is no comparison to the paper of van Doornmalen et al. [51] as they only included results for Early instances in their paper. Unfortunately, we can again see that we cannot compete with the best known solutions, and also still do not find a solution to Early 7 and 11 where van Doornmalen et al. [51] did find a solution.

Moreover, we can see that for 13 instances we only found a solution in the CP phase and did not improve these solutions in the SA phase. Looking further into this problem, we see that the acceptance rate of the SA process is high enough; it starts around 0.9 and

Table 6.24: Hybridization with 16 threads and the best configurations from the SA tuning with equal neighborhood probabilities. avg: the average objective value over all threads that found a solution, neigh: the number of neighbors examined by the SA, config: the configuration that found the best solution, status: how that solution was found, best: the objective value of the best solution, t[s]: time to best solution, F_{ks}: factor of our best solution to the best known solution, F to [39] and [51]: factor of our best solution to the solutions of [39] and [51].

	avg	neigh	config	status	best	t_{best} [s]	F_{ks}	F to [39]	F to [51]
Early1	1419.33	57457972.31	700_0.98_10000_1.03_true_false	SA	1129	76691	3.12	2.67	1.82
Early2	644.67	36407895.38	700_0.98_10000_1.03_true_false	SA	592	17296	4.08	1.86	1.60
Early3	3140.25	30636369.56	1200_0.98_100000_1.03_false_false	impr	1351	43873	1.36	1.26	1.11
Early4	-	66683884.69	-	-	-	-	-	-	-
Early5	-	14735000.69	-	-	-	-	-	-	-
Early6	5634.00	16436580.31	-	CP	5634	1877	1.69	1.43	1.20
Early7	-	15929553.12	-	-	-	-	-	-	-
Early8	4133.44	38785941.69	700_0.98_10000_1.05_true_false	impr	2674	80195	2.54	2.54	1.68
Early9	2268.75	32531785.44	700_0.98_10000_1.03_true_false	impr	652	48128	11.64	4.94	3.17
Early10	-	14624669.25	-	-	-	-	-	-	-
Early11	-	13705015.94	-	-	-	-	-	-	-
Early12	2032.50	51054620.44	-	CP	2000	768	6.35	1.98	1.77
Early13	919.00	30044495.81	700_0.98_100000_1.05_true_true	SA	670	10226	5.54	3.87	1.80
Early14	1577.19	52448378.38	700_0.98_10_1.05_true_true	impr	165	50708	41.25	2.62	6.88
Early15	6907.00	11796721.31	-	CP	6631	384	2.24	1.86	1.39
Middle1	-	24919025.50	-	-	-	-	-	-	-
Middle2	-	10783365.50	-	-	-	-	-	-	-
Middle3	-	12369352.56	-	-	-	-	-	-	-
Middle4	362.19	97702384.50	-	CP	317	35	45.29	19.81	-
Middle5	2706.00	56683288.75	1200_0.98_100000_1.05_true_false	impr	1049	7098	3.76	2.06	-
Middle6	3514.17	40094887.81	-	CP	3070	156	2.74	1.80	-
Middle7	9445.56	70309652.25	-	CP	7923	268	4.44	3.60	-
Middle8	1280.56	49839021.94	1200_0.98_100000_1.03_false_false	impr	416	20592	3.22	3.06	-
Middle9	3439.06	52874080.69	-	CP	3145	1665	7.58	4.91	-
Middle10	-	32994274.56	-	-	-	-	-	-	-
Middle11	4388.00	12058887.19	-	CP	4388	1442	1.79	1.63	-
Middle12	3956.94	32093070.25	700_0.98_10000_1.03_true_false	impr	1919	3618	3.20	2.02	-
Middle13	8794.56	40924701.56	-	CP	7984	526	31.68	22.06	-
Middle14	3695.62	13332198.00	700_0.98_100_1.03_true_true	SA	3138	63094	2.75	2.68	-
Middle15	2710.31	45757405.00	1200_0.98_100_1.03_true_true	impr	1316	74209	2.71	1.34	-
Late1	3403.69	38372349.25	-	CP	3171	1290	1.65	1.57	-
Late2	-	12488717.94	-	-	-	-	-	-	-
Late3	4585.25	67577741.12	700_0.98_10_1.03_false_true	impr	3264	39964	1.38	1.33	-
Late4	1152.25	200359451.75	-	CP	649	10	649.00	649.00	-
Late5	-	18442098.56	-	-	-	-	-	-	-
Late6	-	67661874.06	-	-	-	-	-	-	-
Late7	3773.43	41141110.00	700_0.98_100000_1.03_true_true	SA	2762	24564	1.77	1.56	-
Late8	3380.00	82007068.88	700_0.98_100000_1.05_true_true	impr	1550	17762	1.66	1.55	-
Late9	2338.00	27042668.62	700_0.98_10000_1.03_true_false	impr	1319	85866	2.65	1.84	-
Late10	-	23063277.25	-	-	-	-	-	-	-
Late11	-	37888390.75	-	-	-	-	-	-	-
Late12	-	28207700.75	-	-	-	-	-	-	-
Late13	10534.00	44590868.06	-	CP	8618	2232	4.74	4.61	-
Late14	3859.94	21229173.75	-	CP	3599	46	3.07	2.99	-
Late15	765.94	89765669.94	700_0.98_10_1.03_true_true	impr	180	82106	180.00	3.00	-

then goes down to about 0.6. At that point the next reheat happens, which brings the acceptance rate back up again. However, the process only accepts infeasible solutions as it does not find any feasible ones. The problem might be that the weight is too low to bring the search towards a feasible solution. However, we already tried using higher weights in our first hybridization experiments; there we had the problem that the overall acceptance rate tended towards zero.

Again, we show exemplarily the improvements over the iterations of Early 3 and Early 13 in Figure 6.5. As we can see in comparison to the plots of the previous experiment with a neighborhood probability according to the number of possible moves, we improved more CP solutions for Early 3 and found one more solution for Early 13 with SA.

Table 6.25: Comparison of CP runs with hybridization runs. Best solutions among our approaches are shown in bold.

	CP	TTSA	tuned	best known
Early1	-	-	**1129**	362
Early2	-	-	**592**	145
Early3	2673	**1239**	1351	992
Early4	-	-	-	507
Early5	-	-	-	3127
Early6	-	5700	**5634**	3325
Early7	-	-	-	4763
Early8	3455	2902	**2674**	1051
Early9	2763	**443**	652	56
Early10	-	-	-	3400
Early11	-	-	-	4426
Early12	**1680**	1900	2000	315
Early13	-	-	**670**	121
Early14	4250	**105**	165	4
Early15	6572	**6543**	6631	2955
Middle1	-	-	-	5177
Middle2	-	-	-	7381
Middle3	-	-	-	9542
Middle4	**225**	295	317	7
Middle5	2490	**775**	1049	279
Middle6	**2560**	3110	3070	1120
Middle7	7985	8157	**7923**	1783
Middle8	1203	**374**	416	129
Middle9	**3105**	3125	3145	415
Middle10	-	-	-	1250
Middle11	-	-	**4388**	2446
Middle12	3229	**1431**	1919	599
Middle13	**7084**	7442	7984	252
Middle14	-	-	**3138**	1140
Middle15	6774	1338	**1316**	485
Late1	**3062**	3160	3171	1922
Late2	-	-	-	5400
Late3	6859	**3249**	3264	2369
Late4	**0**	1004	649	0
Late5	-	-	-	1923
Late6	-	-	-	923
Late7	-	-	**2762**	1558
Late8	3101	**142**	1550	934
Late9	2824	**1149**	1319	498
Late10	-	-	-	1945
Late11	-	-	-	202
Late12	-	-	-	3428
Late13	10900	10540	**8618**	1820
Late14	**3520**	3717	3599	1174
Late15	3845	**115**	180	0

6.11 Final Results

In this section, we will provide a side by side comparison of our best MiniZinc standalone results (without our random implementation for Chuffed), the hybridization with the untuned TTSA and the latest hybridization results for each instance. The experiments ran for 24 hours. The results can be seen in Table 6.25.

Of the 45 instances we found solutions for 30 of them with the latest hybridization. With the MiniZinc configurations we only found solutions for 23 of the instances; with the TTSA hybridization for 24 of the instances. There are 22 instances for which hybridization beat the MiniZinc approach and eight instances where it did not. In these cases, where MiniZinc was better, the gap to the hybridization solution is rather small, except for Late 4 where MiniZinc found the best known solution, which is 0 and can therefore not be improved. The TTSA hybridization is especially good in improving CP results, whereas the latest hybridization excels in finding solutions to prior infeasible instances in the CP phase.

To sum up, hybridization is to prefer when comparing it to MiniZinc standalone runs as it finds more solutions and also better solutions on average. Ideally, full CP runs would be made part of the portfolio in the multi-start hybrids since there are instances where SA was not able to improve initial solutions. Therefore, further work should deal with, possibly automated, tuning of SA with potentially feature-based parameters to be effective on all ITC instances with their varying properties, as done by Rosati et al. [39]. In particular, starting from initial solutions satisfying the phased constraint and only accepting phased-retaining neighbors could be more effective for the phased instances.

CHAPTER 7

Conclusions & Future Work

In this thesis, we investigated a hybrid approach to tackle sports league scheduling based on constraint programming (CP) with MiniZinc and simulated annealing (SA). As challenging representative problems, we considered the Traveling Tournament Problem (TTP) and the recent International Timetabling Competition 2021 (ITC2021). The former was introduced by Easton et al. [10] in 2001 and combines a difficult feasibility and optimality aspect in a rather simple yet $\mathcal{NP}$-hard problem formulation while the latter, introduced by Van Bulck et al. [50] in 2021, aims towards modeling messy real-world problem instances with their plethora of hard and soft constraints.

We started by covering related state-of-the-art works on heuristic approaches to these problems. In our methodology we described foundations for combinatorial and constraint optimization, the CP paradigm, and improvement metaheuristics with the focus on SA. Afterwards, we defined the ITC2021, describing its constraints and objective function. Then we explained our solution approaches regarding CP, including the automated translation of the ITC instance files into MiniZinc models, an adaption of the simulated annealing approach from Anagnostopoulos et al. [1] to the TTP, and a combination of both in a parallel multi-start hybridization. Finally, we presented our thorough computational results.

In the computational study, we compared different configurations for solving the MiniZinc CP models using four backend solvers, two MiniZinc versions, various variable choice heuristics as well as value choice heuristics, and the impact of global constraints and restarting mechanisms. The Chuffed solver with multi-starting and a custom randomization turned out to be the most effective for finding feasible solutions quickly, while for standalone CP optimization long-runs also Gecode and Google OR-Tools provided best results over our approaches on the 45 tested ITC2021 instances. Given the highly constrained nature of this problem, we expected the CP approach to work better but it could only find a feasible solution to slightly more than half of the instances.

Turning to the improvement heuristic, we reimplemented Traveling Tournament Simulated Annealing (TTSA) [1] in the Julia programming language and tested it in the fast cooling regime on TTP benchmark instances with which we could confirm results from the literature. This implementation and the suggested parameters served as a basis for the ITC2021, for which we implemented the corresponding hard and soft constraints, penalization of infeasible solutions and algorithmic improvements. In a second step, we tuned and measured the impact of the various parameters of the adapted SA for ITC2021, for example, the initial temperature, penalization of infeasible solutions, the cooling schedule, neighborhood probabilities, acceptance rate monitoring, and more.

In the end, we combined both parts in a parallel hybrid approach, which uses multiple threads to run a randomized CP solver to find an initial solution, which is subsequently handed over to SA to improve it further with a sampled configuration from the previously tuned portfolio. This allowed us to find feasible solutions for two thirds of the instances and frequently improve initial CP solutions while sometimes the CP full-runs provided better solutions. Finding an effective parameter set over all ITC2021 instances for the SA remained a challenge for us, therefore, we were not able to compete with the best known solutions nor with the work of Rosati et al. [39] who used a three-stage SA approach (second place in the ITC2021). We were a bit closer to the results of van Doornmalen et al. [51] who used a hybrid approach combing mixed integer programming and local search intertwined in a fix and optimize fashion, which seems to be the more promising way to go over CP.

Whether it is more beneficial to use more time to get to a feasible solution in the CP phase or if it is better to spend more time in a local search phase possibly starting with an infeasible solution cannot be definitely answered as it depends in our case on the specific instance. We dealt with instances where we only found a solution with CP and with instances where we only found a solution in the SA phase. Given the rather disappointing performance of CP and the success of SA as demonstrated in the literature, we believe it is beneficial to spend more time in the improvement phase and use CP to quickly generate a random infeasible or partly feasible (e.g., adhering to phased constraints) solution while retaining this partial feasibility in the subsequent improvement by suitable neighborhood structures, see, e.g., Van Hentenryck and Vergados [52].

Future work should include automated tuning, e.g., using irace [28], and studying extensions/improvements for the five, somewhat canonical, TTP neighborhood structures as proposed by Van Hentenryck and Vergados [52] or by Langford [27]. Another interesting direction is to make use of cooperative search for a more effective paralleliziation, e.g., to use Population-Based SA as proposed by Van Hentenryck and Vergados [53] for the TTP, where elite solutions are shared at synchronization points to restart threads which are stuck with worse solutions. Moreover, the search throughput should be increased by performing incremental evaluation, which we implemented for the TTP but omitted for the ITC2021 due the complexity of the constraints. Another improvement could be to choose from a portfolio of configurations for the CP phase from which the algorithm picks one at random as we do it for the SA part.

APPENDIX A

Additional CP Results

In this appendix we show result tables that we deemed to be too exhaustive for the main part of the thesis. The Tables A.1 to A.12 show experiments with MiniZinc on the Early instances comparing the use of global constraints and avoiding them. The Tables A.13 and A.14 show the results of an experiment using Chuffed and $first\_fail$ where we dropped the break constraints, which appeared to be the hardest constraints to fulfill in preliminary experiments, to get to more "near-feasible" solutions than with the break constraints enabled. Thereby, we convert the used instances to ITC2021-NSC-NBC instances. If we compare the Table A.13 to its equivalent with the break constraints Table A.7, we can see the number of solutions was brought up from six to ten, however, of course the solutions found are not feasible anymore.

The columns of the tables show the following: $t[s]$ is the time to first solution, n^{nd} is the number of examined nodes, n^{pd} is the maximum depth of the search tree, n^{fl} is the number of fails, n^{re} is the number of restarts, n^{pr} is the number of propagations, n^{ps} is the number of propagators and $\bar{t}^{\mathrm{nd}}[\mu s]$ is the used time per node in milliseconds. Whenever there is no time given, then there was no solution found for that instance with the particular configuration. If other columns are left blank, the solver did not provide information for it.

Table A.1: Gecode globals first-fail

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$
ITC2021Early1	-	500317016	209	250158419	0	114844957598	3123	173	-	520847196	207	260423509	0	110490017139	3123	166
ITC2021Early2	-	183455859	205	91727856	0	96031980437	40513	471	-	142775683	182	71387763	0	122678596250	40513	605
ITC2021Early3	-	300814678	121	150407293	0	53657003698	4076	287	0.1	221	210	10	0	54422	4076	243
ITC2021Early4	-	178657896	193	89328866	0	54960067727	15734	484	-	174874582	186	87437212	0	54204981543	15734	494
ITC2021Early5	-	322799604	241	161399702	0	123112079792	55140	268	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	132711143	257	66355456	0	41557073415	44828	651	-	365514712	263	182757245	0	96599246021	44828	236
ITC2021Early8	0.1	338	336	1	0	69055	2868	295	0.1	338	337	0	0	68798	2868	315
ITC2021Early9	0.1	318	316	1	0	70410	3453	337	0.1	326	318	7	0	72800	3453	340
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early13	-	140744358	334	70372028	0	101689619346	49148	614	-	148616864	337	74308280	0	121683337352	49148	581
ITC2021Early14	0.2	476	450	18	0	108180	3605	398	0.2	456	447	8	0	100978	3605	387
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.2: Gecode globals input-order

	input-order-indomain-median								input-order-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$
ITC2021Early1	-	513556382	188	256778115	0	114627920637	3123	168	-	510389135	187	255194490	0	111248291423	3123	169
ITC2021Early2	-	88852091	133	44425992	0	25365387647	40513	972	-	156431450	178	78215649	0	151601728179	40513	552
ITC2021Early3	-	387832176	158	193916029	0	77283941684	4076	223	121.0	1468793	192	734309	0	215154741	4076	82
ITC2021Early4	-	249415159	220	124707487	0	47584354754	15734	346	-	97637156	104	48818532	0	27077196847	15734	885
ITC2021Early5	-	-	-	-	-	-	-	-	-	52547120	154	26273499	0	17333831231	55140	1644
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	126772134	230	63385969	0	32271743159	44828	682	-	84220375	196	42110111	0	47411885395	44828	1026
ITC2021Early8	-	284159914	216	142079861	0	45890169982	2868	304	59.7	211627	317	105656	0	28464721	2868	282
ITC2021Early9	0.1	317	307	8	0	71043	3453	269	-	322448293	197	161224059	0	30705138491	3453	268
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early13	-	167685508	318	83842612	0	51414561029	49148	515	-	107421912	251	53710847	0	38386822823	49148	804
ITC2021Early14	-	513049616	238	256524701	0	134428940904	3605	168	-	202315104	245	101157454	0	38377889183	3605	427
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.3: Gecode non-globals first-fail

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$
ITC2021Early1	-	703816744	208	351908285	0	124939835070	37373	123	-	699929410	207	349964617	0	121245568434	37373	123
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	-	104178265	120	52089087	0	36591144912	35142	829	0.1	221	213	7	0	149074	35142	578
ITC2021Early4	-	96164473	199	48082155	0	79414748160	62213	898	-	85301115	179	42650482	0	84016636031	62213	1013
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	0.2	338	336	1	0	217866	54098	718	0.2	338	337	0	0	215841	54098	710
ITC2021Early9	0.2	322	316	3	0	219551	54681	680	-	321888971	255	160944375	0	106079386900	54681	268
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	0.3	472	450	16	0	338538	72701	548	0.5	456	447	8	0	326477	72701	990
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.4: Gecode non-globals input-order

	input-order-indomain-median								input-order-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$
ITC2021Early1	-	723220224	189	361610033	0	127828704865	37373	119	-	695764678	190	347882264	0	120243021236	37373	124
ITC2021Early2	-	140945467	134	70472681	0	50463773583	69505	613	-	-	-	-	-	-	-	-
ITC2021Early3	-	269952299	158	134976088	0	143830810793	35142	320	166.0	1230580	190	615203	0	657342085	35142	135
ITC2021Early4	-	220372506	219	110186164	0	100315778659	62213	392	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	-	216579593	216	108289699	0	98373687862	54098	399	57.3	225855	330	112770	0	75173845	54098	254
ITC2021Early9	0.2	362	310	30	0	227921	54681	660	-	239708277	205	119854050	0	73765782274	54681	360
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

In Figure A.1, MiniZinc solution times with Gecode and restarting using the model listed in A.1 (or corresponding relaxations) are shown. We can see that feasible DRR (plain, phased, and mirrored) and TTP (plain, phased, and mirrored) solutions can be created for up to 40 teams within at most a couple of seconds, therefore, their creation is deemed not to be a bottleneck for approaches with dominant improvement phases.

Table A.5: Chuffed globals first-fail

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	235707021	226	202102454	0	32504787741	4778	367	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	0.2	5441	221	218	0	452037	3608	42	0.2	2698	239	90	0	231328	3608	70
ITC2021Early4	-	161028679	242	80056634	0	32449069664	6206	537	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	46257.3	33455678	223	18410567	0	9088055437	5582	1383
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	0.2	7721	362	654	0	413006	4032	29	0.1	2102	356	127	0	120965	4032	44
ITC2021Early9	0.9	17599	356	685	0	1170242	3947	51	0.3	5502	344	520	0	457132	3947	61
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	530.3	749173	283	172809	0	202769395	7069	708	-	-	-	-	-	-	-	-
ITC2021Early13	-	275289348	330	183256850	0	36119367844	7559	314	-	-	-	-	-	-	-	-
ITC2021Early14	1.5	22264	447	2272	0	1822932	4764	66	0.3	4383	427	233	0	436647	4764	62
ITC2021Early15	786.1	4192089	313	189186	0	419482424	6140	188	-	-	-	-	-	-	-	-

Table A.6: Chuffed globals input-order

	input-order-indomain-median								input-order-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	230384628	198	206488776	0	23549808466	4778	375	-	213541889	179	200143471	0	33908283864	4778	405
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	0.3	3746	209	246	0	472476	3608	85	0.3	3906	212	312	0	391897	3608	73
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	0.1	2385	354	144	0	216174	4032	46	2.2	19249	353	16277	0	3389293	4032	114
ITC2021Early9	0.1	3718	337	122	0	271637	3947	28	0.4	6711	324	966	0	656428	3947	66
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	129.9	139024	265	44081	0	66105718	7069	934	121.7	193044	226	86747	0	75327996	7069	630
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	0.5	13110	428	1931	0	2176424	4764	37	-	176306515	203	175612817	0	68625853746	4764	490
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.7: Chuffed non-globals first-fail

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	0.6	3076	184	170	0	530915	17992	190	0.2	930	193	58	0	190221	17992	211
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	19011.9	8391410	197	6064800	0	4097466277	26372	2266
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	0.6	5101	336	528	0	502384	24822	116	0.1	338	330	7	0	71271	24822	266
ITC2021Early9	1.3	9099	320	448	0	1069088	24737	148	0.2	733	306	119	0	136453	24737	228
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	638.7	581158	257	157875	0	241348345	35929	1099	47414.1	12066999	255	10684590	0	10814793070	35929	3929
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	2.7	15341	425	921	0	1937221	33624	173	0.9	3190	414	147	0	703093	33624	267
ITC2021Early15	266.0	1017217	307	51223	0	175701289	35000	262	221.5	564206	289	31691	0	114829599	35000	393

Table A.8: Chuffed non-globals input-order

	input-order-indomain-median								input-order-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	0.3	1306	178	115	0	336574	17992	250	0.2	672	171	101	0	221124	17992	312
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	0.1	636	328	17	0	145685	24822	225	5.6	14870	330	14149	0	4333877	24822	375
ITC2021Early9	0.1	441	311	9	0	108872	24737	252	1.1	4973	296	1153	0	1224776	24737	218
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	212.3	117430	237	62038	0	99172300	35929	1808	106.9	90419	212	46746	0	54674084	35929	1182
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	2.6	10660	411	1501	0	2409046	33624	248	-	-	-	-	-	-	-	-
ITC2021Early15	724.4	433575	274	202453	0	180710042	35000	1671	-	-	-	-	-	-	-	-

Table A.9: or-tools globals first-fail

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	57.1	-	-	11	-	1920471	-	-	46.1	-	-	23	-	1921463	-	-
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	141.4	-	-	7	-	2711732	-	-	128.6	-	-	6	-	2708632	-	-
ITC2021Early9	135.5	-	-	5	-	2765276	-	-	226.4	-	-	27	-	2821368	-	-
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	282.8	-	-	12	-	4138418	-	-	419.8	-	-	24	-	4190497	-	-
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.10: or-tools globals input-order

	input-order-indomain-median								input-order-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	79.0	-	-	19	-	1956472	-	-	44.3	-	-	35	-	1940989	-	-
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	96.7	-	-	5	-	2708994	-	-	95.8	-	-	16	-	2720923	-	-
ITC2021Early9	103.0	-	-	12	-	2780538	-	-	142.8	-	-	34	-	2825835	-	-
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	22340.1	-	-	52054	-	174160442	-	-	9171.5	-	-	7877	-	34263665	-	-
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	232.4	-	-	35	-	4190717	-	-	179.6	-	-	61	-	4142653	-	-
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.11: or-tools non-globals first-fail

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	64.9	-	-	16	-	11130725	-	-	29.9	-	-	23	-	11119755	-	-
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	126.5	-	-	7	-	10783514	-	-	109.1	-	-	6	-	10778754	-	-
ITC2021Early9	-	-	-	-	-	-	-	-	140.0	-	-	32	-	10702527	-	-
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	356.5	-	-	12	-	10300457	-	-	2443.6	-	-	94	-	11439976	-	-
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.12: or-tools non-globals input-order

	input-order-indomain-median								input-order-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\text{nd}}[\mu s]$
ITC2021Early1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early3	104.5	-	-	20	-	11150576	-	-	70.1	-	-	43	-	11153600	-	-
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early8	103.4	-	-	4	-	10791319	-	-	98.9	-	-	16	-	10793387	-	-
ITC2021Early9	97.1	-	-	12	-	10681057	-	-	382.9	-	-	63	-	10910456	-	-
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early12	26912.2	-	-	54133	-	249704696	-	-	13971.5	-	-	12674	-	58516858	-	-
ITC2021Early13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early14	277.2	-	-	73	-	10599246	-	-	115.6	-	-	60	-	10284991	-	-
ITC2021Early15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.13: Chuffed non-globals on ITC2021-NSC-NBC instances (without soft constraints and break constraints)

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$
ITC2021Early1	0.3	2685	261	247	0	307668	17780	123	0.1	261	246	7	0	50831	17780	218
ITC2021Early2	-	-	-	-	-	-	-	-	43.1	45200	150	33102	0	21928139	18687	955
ITC2021Early3	0.3	2740	210	152	0	449992	17830	95	0.2	947	212	57	0	188486	17830	207
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	2541.1	2258904	204	966566	0	899457118	26372	1125
ITC2021Early7	796.1	2468631	233	167664	0	397278374	25186	323	109.4	340068	220	19965	0	68007238	25186	322
ITC2021Early8	0.6	5132	368	527	0	501344	24494	109	0.1	364	356	7	0	70581	24494	258
ITC2021Early9	1.3	9040	338	441	0	1049802	24525	147	0.2	759	332	119	0	135636	24525	208
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	113.9	296415	290	19040	0	50592949	35020	384	2038.5	1096595	280	520793	0	515611524	35020	1859
ITC2021Early12	25243.8	5920814	257	5072383	0	5591658913	35739	4264	4371.5	1288649	255	950949	0	1218744257	35739	3392
ITC2021Early13	5.1	19624	345	832	0	2565258	34199	258	0.6	2700	325	689	0	753779	34199	238
ITC2021Early14	2.7	15367	451	921	0	1933666	33460	174	0.8	3165	433	143	0	701352	33460	264
ITC2021Early15	394.6	1017217	307	51223	0	175701191	35000	388	231.6	564206	289	31691	0	114829600	35000	411

Table A.14: Chuffed globals on ITC2021-NSC-NBC instances (without soft constraints and break constraints)

	first-fail-indomain-median								first-fail-indomain-min							
	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$	$t[s]$	n^{nd}	n^{pd}	n^{fl}	n^{re}	n^{pr}	n^{ps}	$\bar{t}^{\mathrm{nd}}[\mu s]$
ITC2021Early1	0.3	5555	279	367	0	353593	3396	46	0.1	2765	270	108	0	133631	3396	38
ITC2021Early2	-	213212954	170	100013256	0	39401360166	4303	405	100.3	76564	167	63695	0	39998196	4303	1309
ITC2021Early3	0.3	4852	250	210	0	376591	3446	61	0.2	2851	253	98	0	229115	3446	64
ITC2021Early4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early6	-	-	-	-	-	-	-	-	47355.2	33455678	223	18410567	0	9088055437	5582	1415
ITC2021Early7	734.6	3654947	240	207310	0	413026474	4396	201	840.9	2387740	232	358458	0	463069371	4396	352
ITC2021Early8	0.3	7205	384	669	0	376324	3704	42	0.1	2319	371	145	0	117592	3704	44
ITC2021Early9	0.8	13846	361	580	0	893180	3735	55	0.4	5459	359	527	0	441953	3735	69
ITC2021Early10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ITC2021Early11	233.9	601880	312	93139	0	111368516	6160	389	-	-	-	-	-	-	-	-
ITC2021Early12	7230.9	2555104	283	1831879	0	1891312351	6879	2830	163.4	440026	266	114681	0	138269376	6879	371
ITC2021Early13	2.6	22163	371	1005	0	1805854	5339	116	56.3	117619	317	56894	0	24756363	5339	479
ITC2021Early14	1.4	22459	463	2282	0	1834918	4600	63	0.3	4103	438	235	0	403080	4600	77
ITC2021Early15	919.1	4192089	313	189186	0	419482424	6140	219	-	-	-	-	-	-	-	-

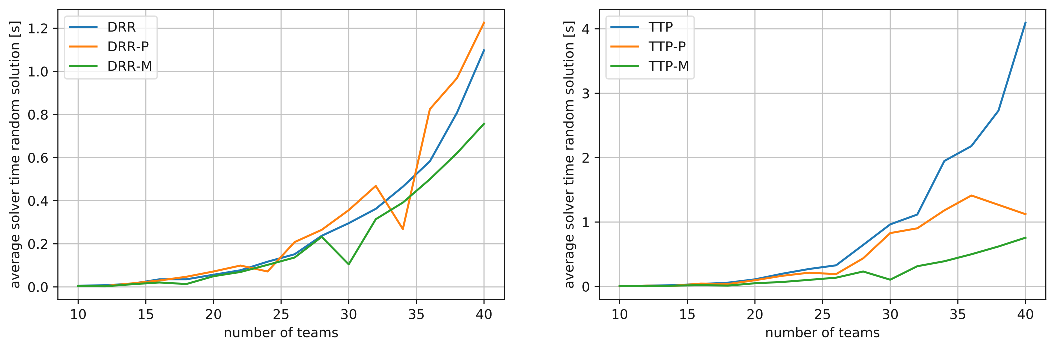

Figure A.1: Mean solving times for one solution over 100 solutions created with MiniZinc/Gecode runs to construct either feasible DRR schedules (left) or TTP schedules (right) from 10 up to 40 randomly, also with additional phased and mirrored constraints.

Listing A.1: MiniZinc model for Gecode with restarting for the mirrored TTP.

```
include "globals.mzn";
include "gecode.mzn";
include "alldifferent.mzn";
include "symmetric_all_different.mzn";
include "global_cardinality.mzn";

int: N;
int: R = 2*N-2;
int: R2 = N-1;
int: U = 3;
set of int: TEAMS = 1..N;
set of int: SLOTS = 1..R;
set of int: SLOTSNOREP = 1..R-1;
set of int: SLOTSATMOST = U+1..R;
set of int: SLOTS1 = 1..R2;
set of int: SLOTS2 = (R2 + 1)..R;

array[TEAMS,SLOTS] of var TEAMS: opponents_per_team_and_round;
array[TEAMS,SLOTS] of var {-1, 1}: venue_per_team_and_round;

% team does not play against itself
constraint forall (i in TEAMS, r in SLOTS) (opponents_per_team_and_round[i, r] != i);

% all games (opponent+venue) per team must be unique
constraint forall (i in TEAMS) (all_different([venue_per_team_and_round[i,r]*opponents_per_team_and_round[i,r] | r in SLOTS]));
% every team must play exactly twice against every other team over the whole schedule
%constraint forall (i in TEAMS) (global_cardinality([opponents_per_team_and_round[i,r] | r in SLOTS],
  [j | j in TEAMS where j != i], [2 | j in TEAMS where j != i]));
% phased: every team must play exactly once against every other team in each half of the schedule
constraint forall (i in TEAMS) (global_cardinality([opponents_per_team_and_round[i,r] | r in SLOTS1],
  [j | j in TEAMS where j != i], [1 | j in TEAMS where j != i]));
constraint forall (i in TEAMS) (global_cardinality([opponents_per_team_and_round[i,r] | r in SLOTS2],
  [j | j in TEAMS where j != i], [1 | j in TEAMS where j != i]));
% mirror (implies phased)
constraint forall (i in TEAMS, r in SLOTS1) (opponents_per_team_and_round[i, r] == opponents_per_team_and_round[i, r+N-1]);

% venue connection constraint
constraint forall (i in TEAMS, r in SLOTS) (venue_per_team_and_round[opponents_per_team_and_round[i,r],r]
  == -venue_per_team_and_round[i,r]);
% opponent connection constraint, symmetric_all_different works better with Gecode first_fail/indomain_random
constraint forall (r in SLOTS) (symmetric_all_different([opponents_per_team_and_round[i,r] | i in TEAMS]));
%constraint forall (i in TEAMS, r in SLOTS) (opponents_per_team_and_round[opponents_per_team_and_round[i,r],r] == i);

% no repeat constraint
constraint forall (i in TEAMS, r in SLOTSNOREP) (opponents_per_team_and_round[i, r] != opponents_per_team_and_round[i, r+1]);

% at most constraint, hardcoded for U=3
%constraint forall (i in TEAMS) (sliding_sum(-U, U, U+1, [ venue_per_team_and_round[i,r] | r in SLOTS ]));
constraint forall (i in TEAMS, r in SLOTSATMOST) ((venue_per_team_and_round[i, r-3] == 1 /\
  venue_per_team_and_round[i, r-2] == 1 /\ venue_per_team_and_round[i, r-1] == 1) -> (venue_per_team_and_round[i, r] == -1));
constraint forall (i in TEAMS, r in SLOTSATMOST) ((venue_per_team_and_round[i, r-3] == -1 /\
  venue_per_team_and_round[i, r-2] == -1 /\ venue_per_team_and_round[i, r-1] == -1) -> (venue_per_team_and_round[i, r] == 1));

% solving
solve
:: seq_search([
int_search(opponents_per_team_and_round, first_fail, indomain_random),
int_search(venue_per_team_and_round, first_fail, indomain_random)])
:: restart_constant(5000)
satisfy;

output [ format(3, 2, opponents_per_team_and_round[i,j]*venue_per_team_and_round[i,j]) ++
if j == R then "\n" else " " endif |
i in TEAMS, j in SLOTS
];
output ["\n"];
```

List of Figures

List of Algorithms

Bibliography

[1] Aris Anagnostopoulos, Laurent Michel, Pascal Van Hentenryck, and Yannis Vergados. A simulated annealing approach to the traveling tournament problem. *Journal of Scheduling*, 9(2):177–193, 2006.

[2] Ian Anderson. *Combinatorial designs and tournaments*, volume 6 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, 1997.

[3] Timo Berthold, Thorsten Koch, and Yuji Shinano. MILP. try. repeat. computing solutions to the ITC 2021 instances by repeated massive parallel MILP computations. In *13th International Conference on the Practice and Theory of Automated Timetabling*, 2021.

[4] Bob Bixby. The gurobi optimizer. *Transp. Research Part B*, 41(2):159–178, 2007.

[5] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[6] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed website. URL `https://github.com/chuffed/chuffed`.

[7] Dominique De Werra. Scheduling in sports. *Studies on Graphs and Discrete Programming*, 11:381–395, 1981.

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

[9] Guillermo Durán. Sports scheduling and other topics in sports analytics: a survey with special reference to latin america. *Top*, 29(1):125–155, 2021.

[10] Kelly Easton, George Nemhauser, and Michael Trick. The traveling tournament problem description and benchmarks. In *International Conference on Principles and Practice of Constraint Programming*, pages 580–584. Springer, 2001.

[11] Matthias Elf, Michael Jünger, and Giovanni Rinaldi. Minimizing breaks by maximizing cuts. *Operations Research Letters*, 31(5):343–349, 2003.

[12] George H.G. Fonseca and Túlio A.M. Toffolo. A fix-and-optimize heuristic for the ITC2021 sports timetabling problem. In *13th International Conference on the Practice and Theory of Automated Timetabling*, 2021.

[13] Nikolaus Frohner, Bernhard Neumann, and Günther R. Raidl. A beam search approach to the traveling tournament problem. In Luís Paquete and Christine Zarges, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 67–82, Cham, 2020. Springer International Publishing.

[14] Nikolaus Frohner, Jan Gmys, Nouredine Melab, Günther R Raidl, and El-Ghazali Talbi. Parallel beam search for combinatorial optimization. In *51th International Conference on Parallel Processing Workshop*, ICPP Workshops '22. Association for Computing Machinery, 2022.

[15] Nikolaus Frohner, Bernhard Neumann, Giulio Pace, and Günther R Raidl. Approaching the traveling tournament problem with randomized beam search. *Evolutionary Computation Journal*, 2022. in press.

[16] Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[17] Ian P Gent, Christopher Jefferson, and Peter Nightingale. Complexity of n-queens completion. *Journal of Artificial Intelligence Research*, 59:815–848, 2017.

[18] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[19] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

[20] Zonghao Gu, Edward Rothberg, and Robert Bixby. Gurobi website. URL `https://www.gurobi.com/`.

[21] Eric J Hoffman, JC Loessi, and Robert C Moore. Constructions for the solution of the m queens problem. *Mathematics Magazine*, 42(2):66–72, 1969.

[22] Tiago Januario, Sebastián Urrutia, Celso C Ribeiro, and Dominique De Werra. Edge coloring: A natural model for sports scheduling. *European Journal of Operational Research*, 254(1):1–8, 2016.

[23] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[24] Graham Kendall, Sigrid Knust, Celso C Ribeiro, and Sebastián Urrutia. Scheduling in sports: An annotated bibliography. *Computers & Operations Research*, 37(1): 1–19, 2010.

[25] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

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

[27] Glenn Langford. An improved neighbourhood for the traveling tournament problem. *arXiv preprint arXiv:1007.0501*, 2010.

[28] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[29] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8 (1):99–118, 1977.

[30] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[31] Ibrahim H Osman and N Christofides. Simulated annealing and descent algorithms for capacitated clustering problem. *Imperial College, University of London, Research Report*, 1989.

[32] Ibrahim Hassan Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of operations research*, 41(4):421–451, 1993.

[33] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[34] Laurent Perron. Operations research and constraint programming at google. In *International Conference on Principles and Practice of Constraint Programming*, pages 2–2. Springer, 2011.

[35] Laurent Perron and Vincent Furnon. Or-tools. URL `https://developers.google.com/optimization/`.

[36] Antony E. Philips, Michael O'Sullivan, and Cameron Walker. An adaptive large neighbourhood search matheuristic for the ITC2021 sports timetabling competition. In *13th International Conference on the Practice and Theory of Automated Timetabling*, 2021.

[37] Rasmus V Rasmussen and Michael A Trick. A benders approach for the constrained minimum break problem. *European Journal of Operational Research*, 177(1):198–213, 2007.

[38] Jean-Charles Régin and Arnaud Malapert. Parallel constraint programming. In *Handbook of Parallel Constraint Reasoning*, pages 337–379. Springer, 2018.

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

[40] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[41] Peter J. Stuckey, Kim Marriott, and Guido Tack. Minizinc handbook, release 2.6.4, 2021. URL `https://www.minizinc.org/doc-2.6.4/en/index.html`.

[42] Elias Subba and Ole Jacob Lygre Stordal. Scheduling sports tournaments by mixed-integer linear programming and a cluster pattern approach: computational implementation using data from the international timetabling competition 2021. Master's thesis, 2021. URL `https://openaccess.nhh.no/nhh-xmlui/bitstream/handle/11250/2766790/masterthesis.pdf`.

[43] Daniil Sumin and Ivan Rodin. MILP based approaches for scheduling double round-robin tournaments. In *13th International Conference on the Practice and Theory of Automated Timetabling*, 2021.

[44] Guido Tack and Mikael Zayenz Lagerkvist. Gecode website. URL `https://www.gecode.org/`.

[45] Clemens Thielen and Stephan Westphal. Complexity of the traveling tournament problem. *Theoretical Computer Science*, 412(4-5):345–351, 2011.

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

[47] David C Uthus, Patricia J Riddle, and Hans W Guesgen. DFS* and the traveling tournament problem. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 279–293. Springer, 2009.

[48] David C Uthus, Patricia J Riddle, and Hans W Guesgen. Solving the traveling tournament problem with iterative-deepening A*. *Journal of Scheduling*, 15(5): 601–614, 2012.

[49] 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.

[50] David Van Bulck, Dries Goossens, Jeroen Belien, and Morteza Davari. The fifth international timetabling competition (ITC 2021): Sports timetabling. In *MathSport International 2021*, pages 117–122. University of Reading, 2021.

[51] Jasper van Doornmalen, Christopher Hojny, Roel Lambers, and Frits Spieksma. A hybrid model to find schedules for double round robin tournaments with side constraints. In *Proceedings of the 13th International Conference on the Practice and Theory of Automated Timetabling-PATAT 2021*, page 412, 2021.

[52] Pascal Van Hentenryck and Yannis Vergados. Traveling tournament scheduling: A systematic evaluation of simulated annealling. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, volume 3990 of *LNCS*, pages 228–243. Springer, 2006.

[53] Pascal Van Hentenryck and Yannis Vergados. Population-based simulated annealing for traveling tournaments. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, number 1, pages 267–262. MIT Press, 2007.

[54] Laurence A Wolsey. *Integer programming.* John Wiley & Sons, 2020.