Universiteit Utrecht
Department of Information and Computing Sciences

Master Thesis

Generic Scheduling of Sports Tournaments

Author:
Paul Staats
ICA-3404447

Supervisor:
Dr. J.A. Hoogeveen
Second supervisor:
Dr.ir. J.M. van den Akker

October 2014

Abstract

Tournament scheduling is well-researched. Recent literature focuses on specific tournaments, for instance the Brazilian football league or the Finnish major ice hockey league. All of these major competitions have specific settings, ranging from travel distances to television broadcasts. These settings increase the difficulty of the problem, but they also only focus on the specific tournament. We will do exactly the opposite, focusing on the basic problem and extending it with many smaller extensions, which will lead to a more general problem of trying to schedule a common tournament. The settings mostly focus on normal home-users, instead of the highest classes of sports, who will be able to create a tournament for for instance the local sports club or a tournament between streets.

To schedule these tournaments, we created a web-based tool. Using the tool it is possible to set the value of a lot of settings; the number of teams, the number of rounds, different types of tournaments etcetera. Five different types of tournaments are acknowledged: the competition with (weekly basketball league) and without (weekly chess league) time constraints, the common tournament (football tournament on one location), the single round tournament (checkers) and the multiple disciplinary tournament ("straattoernooi"). It can solve the given problems using two different techniques; ILP and CP. We will show how we implemented several of the settings and how they influence the problem. We will of course also show some of the results of the tool.

Contents

Chapter 1

Introduction

Sports are an important part of peoples lives. They tend to spend a part of their leisure time competing in sports. Some of these sports are done without any opponent (fitness, jogging, swimming), though even some of those can be played in a tournament. Some people actually say that competitive sports can only exist when it has some kind of competition [1]. Therefore, to make sports possible, tournaments and competitions are also really important. People want to compare how good they are in a sport. This started already with the first kind of tournaments in Greece [2] (and probably before that in Sumer [3]). These tournaments can be played on a single day (for instance checkers) or can last multiple weeks (for instance football). All these types of tournaments lead to more people competing in sports, because most people enjoy some form of competition, because trying to be the best motivates people.

The problem is always that it is hard to create such a tournament. A lot of people are involved, all having their own specific requests and they all want their request to be honored. So creating a tournament for a large number of teams and a lot of requests becomes a dire task, which will result in quite a suboptimal solution when done by hand.

There are a lot of constraints concerning a tournament, for instance the following: "We are limited in the number of rounds our tournament consists of and in the number of teams that can play at the same time. Which rounds can a team play in? Do we need breaks between games? Do we need a minimum number of games per round?"

These constraints can be divided in two different types: hard and soft constraints. Hard constraints are constraints we are not allowed to break; if we break any of them we will always have a schedule which will not be acceptable. Soft constraints are constraints we are allowed to break, but the fewer constraints we break, the better a schedule is.

In this thesis we will automatically create a schedule for a sports tournament, where we will use the computational power of the computer to create a schedule which will fit the constraints given. The constraints can be set by a user using the program created. The resulting program is web-based, which makes it accessible for everyone without having to install anything. There is no computing science knowledge expected, the program works on the input by the user alone and will return a schedule which will fit the input by the user as good as possible.

To accommodate all the possibilities different sports and their peculiarities, multiple types of tournaments are recognized. Depending on the type of tournament, the user can set different constraints. Not all of the constraints are useful in every type of tournament. The resulting schedule can then be used by the user to accommodate their own sports tournament!

1.1 Previous work

Scheduling itself is a really important research subject in the field of computer science and within scheduling, sports scheduling is an important subject.

We start with a short introduction about the basic problem we are trying to solve. We will use the explanation from Drexl and Knust [4]:

> The league consists of $2n$ teams, each team has to play against each other team exactly $l \geq 1$ times. In order to schedule these $\binom{2n}{2}l = n(2n-1)l$ games, $(2n-1)l$ rounds are available, where each team has to play one game in each round. Thus, for each round $t = 1, ..., (2n-1)l$ one has to determine which teams $i, j \in \{1, ..., 2n\}$ play against each other in this round and for each of these pairings $i-j$, whether it is played in the home stadium of team i (home game for i) or in the home stadium of team j (away game for i).

It has to be noted though that the $(2n-1)l$ rounds limitation is not always the case. We can have some additional room in our tournament to schedule the games in. The same paper points that out and calls the tournaments time-constrained (with the limitation) or time-relaxed (without the limitation). We also use both in this thesis, since it is quite common to have a time-relaxed tournament.

The number of games teams play can differ, most competitions use $l = 1$ (Single round robin, everyone plays each other once) or $l = 2$ (Double round robin, everyone plays each other twice), but there are also competitions which have a triple round robin [5] or even more [6].

The number of teams can of course be even and odd and in the case when we have an odd number of teams, we normally solve that by introducing a bye team (a dummy team, the team playing this team will normally automatically win), which will make sure we always have got an even number of teams. This means we do not have to take care about making specific changes for the odd case, since the extra team will just solve it for us automatically.

An extension of the tournament problem is the traveling tournament problem. We will not implement this in our thesis, but quite some research has been done concerning that problem as well. The traveling tournament problem is a combination between the tournament problem and the traveling salesman problem. Some competitions have got large distances between stadiums. A team which has to move little, will have an advantage to teams who have to move large distances between games. Therefore the distance is taken into account when solving the tournament problem.

We will shortly explain which methods have been used in the past to solve the tournament scheduling problem and how they work. We start with graph-coloring, which was one of the first methods to solve the tournament scheduling problem. After graph-coloring a lot of solution approaches have been tried. This includes decomposition of the problem, Integer Linear Programming, tabu search, constraint programming, heuristic searches and others. I will give a short introduction in the aforementioned approaches to solve the problem.

1.1.1 Graph-coloring

Graph coloring is a technique where we create a graph and color the edges with a set of colors. Every node can only be attached to each color once and the colors represent the solution. In our case, the nodes are the teams, the edges are the matches and every color represents a round. If we are able to color all the edges using the colors (rounds) provided, we are able to find a solution to our scheduling problem.

Graph coloring was one of the first techniques to solve the problem of tournament scheduling, for instance used by de Werra [7, 8]. It is an obvious choice, since we want to know which game is played in which round. When we are able to color our graph (which is made up from all the teams, where lines are games between them) with the number of colors (rounds) provided, we have found a viable solution.

1.1.2 Decomposition

Another basic algorithmic approach to problems like the tournament scheduling problem is decomposition. We take the problem, try to divide it into smaller subproblems and solve these. Using the solutions to the smaller subproblems, we merge these sub solutions to find the solution to the total problem.

For a tournament scheduling problem, this could be done by dividing the problem in two parts. The next example is from Trick [9]. The first part is to schedule the teams, ignoring any home-away requirements. The schedule just consists of teams playing each other in a given round, but we do not include who plays at home or away. Since that is also important to include in any tournament (we do not want a team to play every game at home), the second step will only consist of finding out who plays at home and who plays away.

Nemhauser and Trick [10] use a tree-step decomposition approach. They start with generating patterns, which are orders in which teams play their games. For instance HAHBAAH stands for home, away, home, bye, away, away and home. The second step consists of assigning matches to the pattern sets, while still ignoring the teams. Then they will assign teams to the patterns, while using their specific preferences for having a certain round at home for instance.

Decomposition itself is a helpful technique, but we will still have to solve the sub parts of the problem. That is why decomposition is always used with another technique, so we will see it quite often with the techniques we will address later.

1.1.3 Integer Linear Programming

Another technique which is widely used for (sport) scheduling problems is Integer Linear Programming (ILP). ILP uses variables, which are defined when describing the problem. We generally want to decide for these variables whether they are true (1) or false (0). For instance whether team A plays team B in round C ($A\_B\_C = \{0, 1\}$).

To solve problems, nearly always an ILP-solver is used, which is optimized to solve the variables and constraints posed on them. When we select $A\_B\_C = 1$, we know that we will never be able to play any other game with A or B in round C. So these are blocked, because of the choice made. We will have to formulate the constraints on the problem (a team can only play once in a round). We give these constraints to a solver and then we will, in return, get which of the variables are true.

Briskhorn and Drexl [11] start with a basic approach which tackles the single round robin (every team plays each other exactly once) version of the tournament scheduling problem and then they pose more constraints on the problem. This ranges from team constraints to third party constraints and fairness constraints. They use a set of real-life examples to show how to formulate constraints you can pose on an ILP-formulation.

Already in the early 90's Chaudhuri, Walker and Mitchell [12] did work on how to formulate constraints for ILP-problems. They give a very in-depth and theoretical base about formulation ILP-constraints for scheduling. Della Croce and Oliveri [13] used ILP to find a schedule for the Italian Football League. It is a double round robin tournament created using a combination of the decomposition approach and ILP. Duran et al [14] created a nice paper about the Chilean competition, where a lot of constraints will be implemented for all of the parties involved in the Chilean competition. They use ILP to solve the problem, adding formulations for all of the constraints which will have to be taken care of.

1.1.4 Constraint programming

Constraint programming (CP) is based on constraints, stating how variables influence each other. All of the variables in our problem have finite domains and we want to select a value from their finite domain for every of our variables, in such a way that we do not violate any constraints. We use propagators, which are functions, to make the domains smaller and to make the constraints stronger. A propagator will always make us get closer to a solution, since we will never make the domains bigger again.

We will not directly find a solution all the time, so we will have to branch in our search in order to find a solution. We will use backtracking to make sure we search all the branches if necessary. It is a technique which is used a lot on scheduling problems and it is also used on tournament scheduling problems.

Henz [15] uses CP to solve part of the problem, since he first uses decomposition to create smaller subproblems. It uses the same decomposition we saw with Nemhauser and Trick [10] in the decomposition section in combination with CP.

Regin [16] uses CP on another problem, closely attached to the tournament scheduling problem. He wants to minimize the breaks (rounds in which a team does not play) in his resulting schedule and uses CP to find schedules with a minimum number of breaks. Larson [17] uses CP to find a schedule to the Swedish top handball league, the eliteserien. The eliteserien consist of single round robins (everyone plays each other once) and double round robins (everyone plays each other twice: once at home, once away). Then there are additional constraints concerning the teams and the tournament in total.

1.1.5 Local search

Heuristic searches is the last type we will discuss. It is used widely in all sorts of problems and it is also used in tournament scheduling problems. There are a lot of techniques and we will only discuss one of them. Some techniques work better on certain types of problems, so it is also important to review the technique used. We will discuss tabu search.

The idea behind these local-search-based heuristics is that they will generally be able to find an acceptable solution in a lower amount of time. They normally start by trying to find a viable solution and then try to increase the quality of this solution by making smart decisions. More information can be found in many books, including [18] and [19].

1.1.5.1 Tabu search

The tabu search technique ([20, 21]) has been used as one of the primary techniques to solve the scheduling of sports tournaments. Hamiez and Hao [22] formulate the basic sports league scheduling problem as a CSP (constraint satisfaction problem). They start with an accepted solution and then try to find the best solution possible using tabu search.They then use a neighborhood based algorithm, evaluating how good solutions found are and try to find the best solution to continue the search from.

Gaspero and Schaerf [23] use tabu search to solve the traveling tournament problem. They use extensive calculations to make sure good neighbors are found compared to the original problem.

Ribeiro and Urrutia [24] used tabu search to compute a solution to the mirrored traveling tournament problem: this is the traveling tournament problem where the schedule will be mirrored when we scheduled half of the games. They make a solution, which is done using decomposition and then try to find a better solution using tabu search. They have got a couple of different ways to compute neighborhoods and use all of them to find the best neighbor.

1.1.6 Real-life examples

A lot of real-life examples can be found about the tournament scheduling problem. Because nearly every tournament has its own specific characteristics, there have been multiple papers about soccer [13], cricket [25], basketball [26], baseball [27], tennis [28], ice hockey [29] etcetera. Many of these papers have got characteristics concerning that sport, the amount of time there is to show the sport on TV or specific tournament constraints.

Most of the research done so far about tournament scheduling, especially the last decade, has been upon the scheduling of tournaments for the highest competitions. Since there is a lot of money involved in those parts of sports, there are a lot of things which will have to be taken into account. Some teams may have a preference to play on a certain day (for instance Friday) or we may have constraints concerning which teams play at home since a sponsor does not want two teams to play at home at the same day. There are a lot of examples, like the baseball league [27], the national soccer league in Austria and Germany [30] or the national soccer league in Italy [13] and many more.

This thesis will be focused on the practical aspects of scheduling, focusing more on a typical tournament for the lower divisions, because of this some of the information written before does not apply. For instance we do not have to worry about traveling times [31] or take into account whether we do not break any TV games [26] or sponsor agreements [32]. We will, however, apply some of the techniques mentioned before. There are just different things considering problem definitions and constraints.

1.2 Goal

This thesis is about scheduling a tournament aimed at regular users, considering constraints which are found in scheduling a basic tournament. We consider different types of tournaments, since we want to be as generic as possible, but we will leave out a lot of the more specific constraints. Those constraints could for instance be the constraints about considering the traveling distance or the order in which teams play against each other. They will not be taken into account, since they are much too specific for the generic problem concerning sports tournament. The result is a program which can be used by anyone, through the web, to create a tournament.

The last part is actually where this thesis differs from the previous work. The previous work all attacked a specified problem and tried to solve that. This thesis focuses more on the general cases and we depend on the user to give us the information about a tournament. The settings are maximum number of games per round, minimum number of games per round, unavailabilities etcetera. This can probably be used to solve some of the problems from the previous work, but also to problems which are quite different from those solved in the previous work.

1.3 Complexity

Our problem of course consists of a lot of settings and they all influence the complexity of the problem. There are actually quite some subparts of the problem which are not NP-complete, but there are also a lot of cases where the problem becomes NP-complete. It is easy to see that the problem is in NP, because we are able to check in polynomial time that a given solution does not violate any of our constraints. We will need all of the teams to play the desired number of games and not violate any of the settings on our tournament.

Because there are a lot of subproblems, some NP-complete and some are not, we will not look into all of the NP-complete problems and prove they are. We will show some results from the literature concerning the NP-completeness of the problem. Schaerf [33] shows that the problem becomes NP-complete when we have some settings concerning how the matches against the top teams (”top matches”) should be divided over the total schedule. Itai et al [34] proved that ”restricted” bipartite graph matching is NP-complete. When we have restrictions on our arcs in our bipartite matching, the problem becomes NP-complete.

De Werra [35] has written about scheduling classes, teachers and periods. We want to create a schedule where these are assigned to each other in such a way we decide which teacher gives which class in which period. When we have unavailabilities (a certain teacher or class is unavailable in a certain period) the problem becomes NP-complete. He points to Even et al [36] for the proof of the NP-completeness.

We can also have breaks between our games, which means teams can’t play multiple rounds in a row. When we have breaks, the problem also becomes NP-complete, because the rounds influence each other. Even deciding for a certain team whether we will be able to schedule the remaining games can still be scheduled is already NP-complete (dr. H.L. Bodlaender, personal communication).

1.4 Outline

Chapter 2 provides a problem description, which of the elements we will implement in the thesis and how we formalize them to make sure that they are indeed implementable. This starts with the general problem description and leads through soft-constraints, tournament types and a short summary of how to work with knock-out parts of the tournament. Chapter 3 is about the ILP-approach to this problem, how we implemented the subproblems in an ILP setting and how everything works with the PHP-based website. Chapter

4 is about the implementation using CP, where we will describe the specific adaptations to solve this problem using CP. The implementation is fully in PHP and does not use any separate tools. Chapter 5 is about solving parts of the problem using polynomial algorithms. Chapter 6 is about a max flow implementation of checking whether we can still find a solution and the way it works, including an alternative way to implement checking whether we can find a solution. Chapter 7 contains some information about the implementation, Chapter 8 has some experiments, Chapter 9 is the conclusion and Chapter 10 yields some possible future work.

Chapter 2

Problem definition

2.1 Problem formulation

Our goal in the end is to find a schedule. In our problem we have a limited set of teams ($T_1 ... T_x$) and a limited set of rounds ($R_1 ... R_y$). To formalize what a schedule is, we will first have to formalize what a match is and what a game is.

2.1.1 Match

A Match M is played by 2 teams. These teams will play against each other, where the team first in the line will always be the team that plays at home (if applicable). No round has been set yet, just the teams who will play a certain match.

$$Match\ M = (T1, T2)$$

2.1.2 Game

A game is a match combined with a round. In some tournaments, we will also need a time in a certain round, but we capture this in the round (for instance round 1.1). So a game still just consists of a match and a round, so we do not have to depend on the fact whether we have a time or not.

$$Game\ G = (M, R)$$

2.1.3 Schedule

A schedule is viable, when we don't violate the hard constraints imposed on the problem. These hard constraints can be for instance that we do not want a team to play twice in a certain round. The resulting schedule consists of a combination of a match and a round (eg; a game). So we can define (iff no violations on hard constraints):

$$Schedule\ S = [G]$$

The best schedule is the one where we violate the least number of soft constraints. These soft constraints are constraints posed on our problem, but we are allowed to break them. If we do so, we will decrease the quality of our solution and therefore we will have different scores for different solutions. Nearly every problem has multiple solutions, so we would like to find the best solution. The soft constraints therefore do not influence whether our found solution is viable. They can only be used to score the found solution thus far.

2.2 Settings

There are several settings that can be set on the tournaments. We will explain in short what every setting means and how it influences the scheduled tournament.

2.2.1 Limited number of games per round

We can have a maximum or minimum number of games per round. They are both integers, ranging from 1 to ∞. These limits reflect for instance the maximum number of fields available to play the games on (soccer) or the fact that we want to have at least some games per round to keep it interesting for people who come to watch the games.

2.2.2 Teams

We have at least a set of teams in our tournament. There might even be multiple sets of teams, where we will create the tournament in such a way that teams will only play their own sub-tournament.

2.2.3 Breaks

We can have breaks in our tournament, which means that a team can not play without a B rounds break. This means a team can play no more than one game in $B+1$ subsequent rounds. The setting 'breaks' is an integer; how long the minimum break is. This setting is useful for tournaments where rounds follow each other up in a rapid succession.

This makes the tournament harder to schedule, since we will need to take into account what happens in other rounds, besides the fact which teams are already scheduled against which teams. This is often combined with a minimum constraint on the number of games in a round, because otherwise it could happen that a schedule only has games in rounds 1, 3, 5..... That would mean there would always be an empty round and generally that is not a solution which is liked. Teams tend to like to watch games played by other teams, so the combination of a break with a minimum games per round constrain is used to solve the problem of a broken schedule.

2.3 Number of games

This section is about deciding how many games we should schedule in our tournament. Depending on the type of tournament we are scheduling, we will only have a certain subset of the following types. This all depends on the specific tournament, the number of teams available and the number of rounds we have to schedule. There are many types out there, but we will only give a subset. The remaining types are not interesting to our thesis and are therefore left out.

2.3.1 Single Round Robin (SRR)

Single round robin is the tournament where you play everyone once. This is normally done when you want to play a normal competition with only a limited number of weeks. You can't play everyone twice, but you do want people to play against everyone an equal number of times. This is then of course only fair if everyone plays each team once.

We will have to take into account that there might be home/away matches, because they probably will influence how well teams do. If a certain team plays nearly all of its games at home, it has a huge advantage. Normally it is said that the maximum difference is one. Either one more at home or one more away.

The number of games we play in SRR with n teams is $n-1$ for every team. The total number of games is therefore $\frac{n(n-1)}{2}$, since every game will have two opponents playing against each other. There is no limit on the maximum number of rounds generally. The minimum number of rounds is $n-1$, though it can be larger when we have a maximum number of games per round. If we have $n-1$ rounds, we will have a Dense Single Round Robin.

2.3.2 Dense Single Round Robin (DSRR)

If the SRR is dense, that means that we actually just play $n-1$ rounds to play the $n-1$ matches per team. So the schedule is totally filled, every team plays every round. The only exception is if there is an odd number of teams, where we just add a 'bye' team. If we then fill all the games in n rounds we still call the tournament schedule dense, because there is no way we could have made it more dense.

The denser a schedule is, the harder it is to schedule. If a tournament is dense, we know that there is no room for any breaks or unavailabilities (unless there is a bye team). This makes it harder to schedule compared to the SRR.

2.3.3 Bipartite Single Round Robin (BSRR)

This tournament form is related to SRR. The difference is that we split the teams in two groups and every group only plays against the other group. So lets say we have group X and Y, then every team in X plays against every team in Y. But teams within X can not play against any team in X and neither do teams in Y play against any team in Y.

The number of games we play in BSRR isn't known in advance, since the size of X and Y will not have to be equal. But if we assume they are, the number of games each team plays is $n/2$. That also leads to the maximum number of games we can play with this type of tournament, namely $(n/2)^2$. When we shift the teams between the groups, we also end up with the smallest number, being $n-1$, when we only have one team in group X and the rest of the teams in group Y.

2.3.4 Double Round Robin (DRR)

When we schedule a SRR, you could argue we only play half a tournament. A DRR is a full tournament, a tournament where every team plays each other team twice, once at home and once away. We see this type of tournament in every major sporting competition and it is also used in a lot of tournaments where you have some kind of sub competition. Because this will lead to $2(n-1)$ games per team, this will lead to additional games to schedule. For 10 teams, this already leads to 18 games per team, 180 games in total. The total number of games for this kind of tournament is always double the number of a SRR, so it is $n(n-1)$. Therefore the sets of teams which play in such kind of competition is generally small or the number of rounds is large.

We will not have to take care about home and/or away games, since each team plays each kind once against each opponent. So every teams plays an equal number of home and away games and (different to SRR) we will not have to take it into account when scheduling a tournament. This tournament therefore also always leads to one of the fairest schedules, since every team plays every opponent at home and away. Teams do not have advantages concerning weaker or stronger teams at home or away. But we can have an advantage by the order teams play against their opponents, the home-away patterns.

2.3.5 Dense Double Round Robin (DDRR)

The same as what the dense property says for the single round robin, the dense property holds if we play the $2(n-1)$ matches in $2(n-1)$ rounds. This is actually the case with the Dutch Eredivisie, but it is also used in a lot more large sports competitions. Since this kind of tournament always leads to a full schedule, it is used when we want to make sure that every team plays every week. It leads to the fairest schedule, since there are also no advantages concerning breaks before important matches etcetera, which could give an advantage when a DRR is not dense.

2.3.6 Mirrored double Round Robin

The idea behind the mirrored schedule is that there are combinations of 2 days, where the teams playing each other are the same. The only difference is the fact that the home teams play away and vice versa. They are generally mirrored in the middle round, so round $R/2$. That way you play the same teams in the opposite order (or the same order). It can be easier to model it this way, since we just have to find half the schedule (which is a SRR) and then switch around all the games. This is in fact always also possible with the other DRR tournaments, but when we do have a mirrored schedule, we end up with a mirrored DRR. It is used in papers, because of the fact that it can make the computation easier and sometimes is a condition for a certain tournament.

2.3.7 Triple round robin (TRR)

This type is quite clear without explanation. A team will play each opponent three times, once at home and twice away or twice at home and once away. We can continue this way and even quadruple round robin tournaments exists [6], but the idea is always the same.

2.3.8 Fixed number of matches per team

This is a version I have not been able to find a name for. It is the general case where we do not want the number of teams to influence the number of games a team will play. We set it in advance, every team will play x games. Depending on this setting, we know whether it is smaller than a single round robin, between a single and a double robin or bigger than a double round robin. We could of course continue that way, since a triple round robin exists as well.

For instance when we have n teams and we want to schedule $1.4n$ games, we will have to schedule a game against every opponent once, but also have to schedule some additional games to reach that number. If we only have to schedule $0.7n$ games, we can skip $0.3n$ games, so the team will only meet a subset of its opponents. None of the teams will every play more than the fixed number of matches per team.

2.3.9 Tournament types

In this thesis we give the user the possibility to fill in which type of tournament he wants to schedule. By choosing one of the options, we know how many games to schedule to make sure the tournament is acceptable. For the user this means he can select either 'Single Round Robin', 'Double Round Robin' or 'Amount x'.

When the latter is selected, he can then enter how many games he wants to schedule. This way we give the user full power to create the tournament expected. Both the SRR and DRR can be created using the right number of games

We did, however, implement SRR and DRR in the end, since for most users it is much more logical to select either a SRR or a DDR. There are normally sets of teams who all have to play each other once or twice and not a fixed number of games. So those are selectable and the code then makes sure the right number of games is scheduled.

Dependent on the number of games we have to create the schedule for, we can get a dense schedule. This is only the case if the user selects the right number of rounds and games per round. It is not in the thesis literally, but it is possible to create such a dense tournament by using the right input.

The mirrored double round robin and the bipartite single round robin have not been implemented. It was decided to take a subset of the tournament and since these are quite specific tournament types, these were not implemented. It would not pose any difference to the other types of tournaments though, we would just have to generate a little different schedule. There are some situations where you want these types of tournaments on a professional level, but because this tool is for amateur tournaments we did not include these tournaments.

2.4 Different types of tournaments

In the previous section, we ended up with a rough problem description, but the problem is larger because we acknowledged multiple types of tournaments.

For this thesis project, we worked with 4 different types of tournaments. Since the differences within the tournaments also change the constraints we can set on teams, the thesis almost splits in 4 different parts, with each tournament being one of the parts. There will be some overlap of course, since they are all sports tournaments. We will discuss the differences and equalities and explain how the different types change the way we solve the problems.

In the literature we found that most information concerns the general tournament: which team do we schedule in which round. Here we do something different, because that applies only to the competitions and common tournaments. The single-round tournaments and multiple disciplinary tournaments will nearly always not make a full round, meaning everyone will play each other less than once. This means the bulk of the work is not on finding out who plays who in which round, but also which opponents we want to play. The single-round is even more separate from the rest, since it will focus on scheduling the rounds separately depending on scores.

We will follow up with a short introduction about the different types of tournaments and their specialties and then we will explain in Chapter 3 and Chapter 4 how we solve these specialties.

2.4.1 Competition

This is just the general case of a competition, where we have the most constraints. These constraints reach from 'team x can not play in round y' to 'team x can not play before 12:00' and everything like those. The competition consists of multiple weeks and is the only one which does so. The dates are specific and set in advance, so they do not influence our problem, we just acknowledge rounds. This type of tournament has unavailabilities, we can use that to set when a team can not play.

Our end result is a schedule where we play a specific number of games against the other teams, for instance a single round robin. We want to know who plays who in which round and who plays at home.

In fact we acknowledge two types of tournaments which are subparts of this type. We have one type which has time constraints, because matches are played during different times within a round, and one type which has no time constraints. The first one is used generally for teams playing football etcetera, where we have a limitation on the number of games at the same time and where teams play their games during a full day.

The second one is for games like checkers when players play a competition. They meet at a club house at a set night and play a team-game (teams with for instance 4 players, most points win). There are fixed times per week at which the matches are played and they can not play multiple times per week. Their games take place on different days and times, but one week represents a round.

2.4.1.1 The competition with time-constraints

In this case we have to schedule a normal competition, where we have constraints concerning time and place. For instance certain teams can not play before (or after) a certain time and no more than x teams can play at the same time. This means we should not only generate the matches in a certain round, but also at which time they will be played.

The rounds will be extended to hold this time as part of their round. An unavailability means that we will not be able to play at all in a certain round, while a 'team y can not play before 12:00' means team y can not play in a part of that round.

This type of tournament is for instance used for soccer and baseball.

2.4.1.2 The competition without time-constraints

This type of competition is one where all the games will be played at the same time. So there are no time-constraints concerning the teams or the games. A round is generally just a certain week (this is our assumed standard period), where a team can only play once a week. Depending on whether it is a home or away-game, they might play on different days, but since there are 'week-rounds', that does not influence the resulting schedule.

This tournament has no time-constraints, but there do exist limitations on the number of teams that can play in a certain round. We may not want all the teams to play in every round, since we can only accommodate a certain number of games at the same time. The same with a minimum number of games, since we might need a certain number of games per week to make sure that we have a nice schedule for people who want to come watch the games.

This type of tournament is for instance used for checkers and chess.

2.4.2 A common tournament

This type most of the time consists of a single round robin (every team plays every other team once) with lots of small subcompetitions which only have to play each other. Because these tournaments most of the time last only one day, it is not common that they play more than a single round robin. Otherwise you would meet the same teams twice in a really short time period, so it generally is just a SRR. This type of tournament with a single round robin could end with a knockout part, but we will discuss that more extensively later in section 2.6.

It could also be a partial single round robin, when we have a large set of teams who can all play each other. We then just want to schedule a certain number of games, independent of the number of teams who join our tournament.

In such a tournament we already know the time each round will be played, how long the games will take and what the different subgroups are (if that applies). These preconditions make it close to the original scheduling problem which was discussed in the literature section in chapter 1, where we cited the problem description by Drexl[4]. When the settings are right, we end up with that original problem.

This type of tournament is for instance used for soccer and volleyball. It also applies when a lot of teams come together to play a tournament at one location to compete against each other on a set date. Everyone is at that location and everyone is available at every round. Thus no unavailabilities exist, though we could have some settings concerning the minimum and maximum number of games at the same time, since there might be a limit on the number of fields available (or referees available) to play all the games.

2.4.3 Single round tournament

This type is used a lot in chess (or similar kinds of sports) tournaments. You come together with a lot of players and play a certain number of games. These games normally all happen on the same day, though there are also tournaments which last a weekend. The idea is that you always play a person who scored close to your own score. That way people who play well will play other people who played well and in the end everyone played against opponents close to the strength of the player. The highest scoring player after the set number of rounds is the player who wins the tournament. Every player will always play in each round, so the tournament is always densely scheduled.

Since everybody plays each round, there is a problem when we have an odd number of teams. We fix this by adding a bye, which means someone walks away winning the match without playing anyone. The objective function of this type of tournament exists of minimizing the difference in score. So we want to let the teams with the scores closest to each other play against each other.

This type of tournament normally exists of a partial single round robin, nobody will play each other twice. We also have a home-away game advantage, for instance having white at chess. We will have to take that into account, since having the home game will lead to an advantage and we do not want someone to get this advantage all the time.

In this type of tournament we will have to use a really different type of scheduling, since the rounds will have to be scheduled 'live'. We schedule them when we know all the scores for the previous round(s) and use them to again minimize the difference. This is the biggest difference with the other tournaments, which will all be scheduled fully in advance.

This type of tournament is used for chess, checkers and certain types of card games, for instance bridge. These are mostly games where it is important to play against people who are equally strong and where a lot of players of near equal strength play. We also do not know for every team how strong they are, so we can not make sub competitions easily. Thereby these are all games where there is a lot of difference between players having a weak or strong day, accordingly that will influence who you play in such a tournament.

2.4.4 Multiple disciplinary tournament

This is a type of tournament you will generally see in a more recreational setting. This could for instance be an activity for a group of people. In The Netherlands the 'straatfeesten' or 'straattoernooien' are quite well known (there is actually even a government fund to organize them). In these 'straatfeesten' we have a group of people living in the same street who want to compete in multiple sports against each other (or other streets), but everything is in a recreational setting. The teams will have to compete against each other, playing each sport exactly once, but never twice against the same opponent.

We have therefore an extra set of sports, S, where we will have to make sure that we play them all. The sports all take the same time (or there is an additional waiting time for some games), so all sports have the same amount of time for a certain round. There are no unavailabilities in this type, since all players are (once again) at the same location. So we will just have to take into account that we deliver a viable schedule concerning the sports. The number of sports available is the factor which decides the maximum of games per round.

There are no time constraints etc, but we will have to assign the sports as an extra. We could also need breaks, since we probably do not want everyone to play each round. Especially since it is a recreational tournament, the players probably want a break to go to the other field and to fill in the scores at a central desk (and to have a lot of fun, it still is recreational!).

2.5 Soft constraints

We might have soft constraints in our problem. Soft constraints are constraints we want to fulfill, but if it is not possible to do so, we will violate them. Violating them will mean we will have to pay an extra penalty, since we did break one of the constraints. If there is a solution where we do not break this constraint, that solution should be preferred over the solution found which violates the constraint.

Depending on the tournament, there can be multiple soft constraints. All of these will have a certain penalty, which in total makes it possible to find a score for a certain tournament. With this score we can decide which of the tournaments breaks fewer constraints and accordingly decide which tournament is better.

For instance it is possible to make the number of rounds a soft constraint. We have 10 rounds and the user says that that is actually a soft constraint. We search for a solution and find that we are not able to place all the games within the 10 round constraint. So the games will have to go into a separate round, round 11. It is always better to find a solution where we break the constraint as little as possible, so placing a game in round 12 would lead to a higher penalty compared to placing a game in round 11. This way we make sure that we do not introduce a gap at the end of the tournament, where we will play half of the games in round 11 and half in 12, while we could have played all of the games in round 11.

There are more options a user can make soft, dependent on the type of tournament selected. We will list them in the following chapters, where we will also explain how we handle them. You could for instance think about the number of games a team should play.

2.6 Finishing with a knock-out

After a group part of the tournament, many tournaments will end with a knock-out part. This is for instance happening at every world cup of football, but also at a lot of other different tournaments. This means that scheduling a common tournament will not be enough, because the tournament is not finished yet. We have winners in several sub competitions, but we do not have the general winner for the total tournament yet.

A knock-out is a type of tournament where the winner continues his way on to the next round, while the losing team is removed from the competition. It is impossible to set any information about the knock-out part, whether teams are unavailable etcetera, since we do not know which team will be involved in the knock-out part just yet. The option whether or not to end with a knock-out is available for both the common tournament and the single-round tournament.

For the common tournaments, we will need to know how many teams will have to join the knock-out. This could be just a certain number of spots, but it is also quite normal to have the best 'second places' in all of the sub tournaments to fill up the knock-out.

For the single-round tournament, there are no sub tournaments, thus you can only set how many players/teams should join the knockout. That number of highest ranked players will then join the knockout part.

Special about the knockout part is the fact that we will always need 2^x players. Otherwise we can not play the knockout part. For the single round tournament, this means you can select whether you want to add 2, 4, 8 or 16 teams to the knockout part. The knock-out schemes are predefined, thus they give you no further options. Team 1 will play team x, team 2 will play team $x - 1$ etc. In Figure 2.1 you can see an example of a knock-out tournament when we have 8 teams, ranked on their score so far (1 the highest, 8 the lowest).

For the common tournaments, we will also need 2^x players. This is of course a harder constraint to fulfill there, since we have to take out x players from each sub tournament etcetera. If we have (for instance) 5 sub tournaments, we will never be able to play a knockout part (since neither 5, nor 10, nor 15 is a valid entrance to a knock-out tournament). To make sure that that is possible, it is possible to fill up with the knock-out with the best of the teams not selected yet.

For instance, assume we want to have an 8-team knockout part with 5 sub tournaments. The teams with the highest score from every sub tournament will be added. We will then still need 3 more teams, which will be filled up by the three best scoring teams who came in second in their specific sub tournament. We will then take all the scores

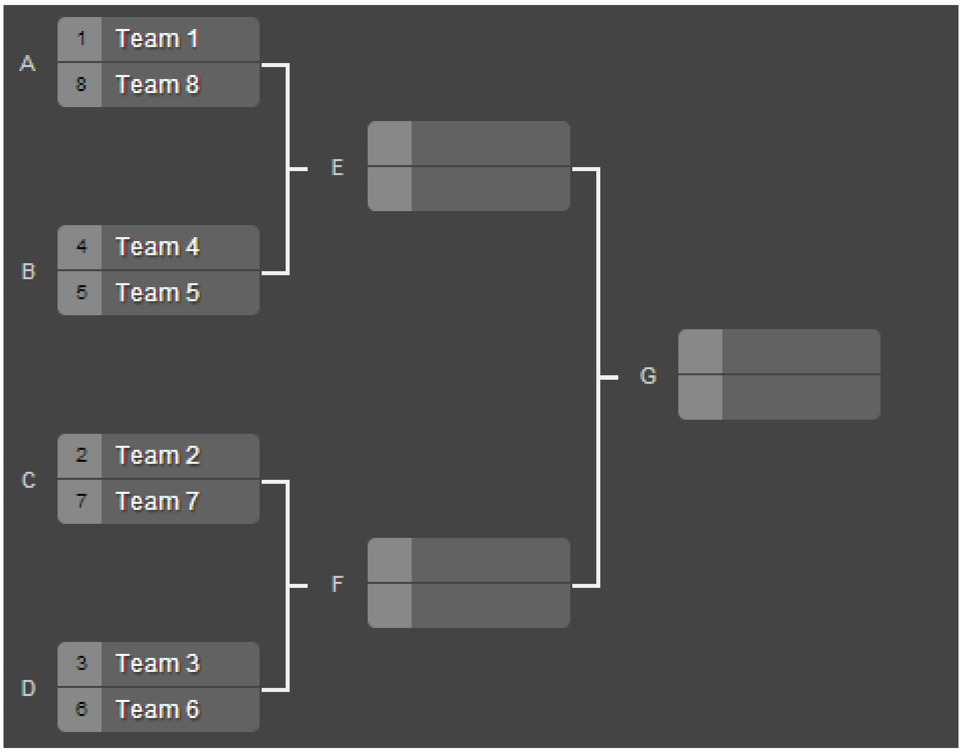


FIGURE 2.1: An 8-team knock-out tournament

by every team and let the best play the worst etcetera. When we have a schedule where we do meet the requirements (for instance 4 sub tournaments) we will just have fixed schedules, independent of the scores. The winner from sub tournament 1 will play the runner up from sub tournament 2 and vice versa.

As said, it is impossible to add team dependant constraints to the knockout part. This is because the order and which teams they play are important in the knockout. So the games will have to be played in that fashion and the user should set the time and date manually.

The user can also choose whether he wants to add a third place playoff. If he chooses to have such a playoff, he can choose from which round on he wants the contestants to join the playoff. Most of the times this is from the semi-final (the two losers will play the 'bronze game'), but sometimes this can be played from earlier on. Judo is a nice example which has tournaments with a third place play-off.

Chapter 3

Integer Linear Programming

This chapter consists of an Integer Linear Programming (ILP) approach to solve this problem. For solving this problem, we use CPLEX [37], though any ILP solver will be able to solve the problem. The given formulations are all formal, though we will also supply some small examples for clarification. We will go through all the settings and how we will solve the sport scheduling problem using ILP.

3.1 ILP formulation

We will split the ILP formulation in two parts. This is done, because these two parts require a different type of formulation and therefore we need to acknowledge both. In the first type we have to find out in which round a match will be played. In the second type we will have to find out which opponent a team gets in a certain round.

We will continue with a small introduction of both and then show the ILP-formulations for both types of tournaments and how the additional settings influence the formulation. We then need to tie some loose ends to finish up the tournaments (address problems concerning the multi disciplinary tournament and the time-constraint competition) and then we end up with a viable schedule.

3.1.1 Finding a round

The first formulation is used in the common tournament, the competitions and the multi disciplinary tournament. We have got a fixed set of matches, which is the total set of matches teams will have to play. We want to assign a round to every match. The result is then a schedule where we assigned a round to every game in the set.

We want to have a full schedule before the tournament starts and only a knock-out part (optional, Chapter 2.6) is not filled in entirely in advance.

3.1.2 Finding an opponent

The second type of tournament is for the single round tournament. The round is given, since we will schedule every round separately and after the previous round, so we just have to decide which teams play against each other in the given round. We do not want to find a full schedule here, which will also most of the time be impossible, because decisions we have made in the previous rounds might block possibilities to find a full schedule. We schedule the rounds one after another, where we will have to await the ending of the previous round before we can schedule the next round.

This type of problem is seen with chess, checkers etc. We have a large set of players and we want them to play against players with equal strength.

3.2 Finding the round for a match

This is the basic formulation for this type of ILP-problem. There are a couple of settings which can be set by the users, so these settings will be explained in separate sections below. These settings are all also formulated as a constraint to the ILP-problem we have. It depends on the user which of the settings are used, because the user selects them, though some of them will not be available for some types of tournaments (for instance the breaks in competitions).

$$\sum_{T2 \in T \setminus T1} (G_{T1,T2,r} + G_{T2,T1,r}) <= 1 \qquad \forall T1 \in T, \forall r \in R$$

We never want a team to play more than once a round. Therefore we sum all of the variables for a certain team and a certain round and make sure that the sum of all the variables together is lower or equal to one. If it would exceed the one, we know that the schedule will fail, since no team can play two games at the same time.

$$G_{T1,T2,r} \in \{0, 1\} \qquad \forall T1 \in T, \forall T2 \in T, T1 \neq T2, \forall r \in R$$

We of course also want the variables to be binary, since they are the decision variables whether or not a game is played in a certain round.

Depending on the type of tournament and the setting, we will extend the ILP-formulation. We will continue with the specific settings and formulate the constraints they pose on the problem to complete the ILP-formulation. We do not even need an objective yet, since the schedule we find is already viable and the best there is. Later on in this chapter we will also explain which of the settings can influence the objective.

3.2.1 Fixed number of games

We will need to know which number of games a user wants the team to play. We do this by giving him certain options and then formulate separate constraints concerning the number of games he decided to play.

The maximum number of games possible is the number of games in a double round robin. Even with the specific number of games, we will not go over the number of games we can schedule using a DRR. When we would go over this number, we will introduce problems concerning the formulation as is (since we will have to decide for a certain match in which two rounds it will be played). This was therefore left out. It can still be done by first creating a double round robin tournament and then the remaining games you want to schedule as user.

We will continue with the different options the user has considering the number of games in the tournament he wants to schedule and which ILP-formulation makes sure we get that kind of tournament. Remember these are all extensions to the ILP-formulation above.

3.2.1.1 Single round robin

To implement a single round robin, we will have to change the constraints a little bit to make sure that we only play every match once. So what we do is make sure that for every game, only the home or away game is selected. The result is the following formulation:

$$\sum_{r=1}^{R}(G_{T1,T2,r} + G_{T2,T1,r}) = 1 \qquad \forall T1 \in T, \forall T2 \in T, T1 \neq T2$$

There is however the problem of the home and away games. If we only implemented the above constraint, we could end up with a schedule where one team could play all of the games at home or away. We do not want that as a viable result, and we have to make sure that the teams get an equal number of games at home and away. When we have an odd number of teams, we have an even number of opponents and should end up with an even number of home and away games. When we have an even number of teams, thus an odd number of opponents, we can only have a difference of one between the home and away games. The formulation to make sure that we do not exceed this amount is the following:

$$\sum_{r=1}^{R}\sum_{T2 \in T \setminus T1} G_{T2,T1,r} \leq \lceil (T-1)/2 \rceil \qquad \forall T1 \in T$$
$$\sum_{r=1}^{R}\sum_{T2 \in T \setminus T1} G_{T1,T2,r} \leq \lceil (T-1)/2 \rceil \qquad \forall T1 \in T$$

3.2.1.2 Double round robin

We will have to play every game once. So we will generate all the matches as variables and we will just have to schedule all of them. We do not have any form of home/away games which we will have to take into account, since we always play an equal amount of them. So the only thing we will have to do is make sure every game is scheduled. The resulting formulation fulfills that:

$$\sum_{r=1}^{R} G_{T1,T2,r} = 1 \qquad \forall T1 \in T, \forall T2 \in T, T1 \neq T2$$

We do not take the home/away pattern into account in this formulation. This is addressed in multiple other papers and could be an extension to the way we find a solution at this moment.

3.2.1.3 Specific number of matches

This setting is the hardest when the number of games is unknown when we are going to schedule. Recall that the user inputs a number x, which is the number of games we will have to schedule. Since we do know the number of teams and the number of games we need to schedule, we will have to do some calculations to make sure that we schedule the right amount. We feed these numbers to CPLEX and the calculations are done in PHP.

- Less games than a SRR
  We schedule every game at most once, making sure every team has the right number of games.
- Equal to the number of games in a SRR
  We just schedule it like it is a SRR.
- More games than a SRR, but less than a DRR
  We schedule every game at least once, at most twice, making sure every team has the right number of games.
- Equal to the number of games in a DRR
  We just schedule it like it is a DRR.

When we end up with the first or third case, this means we will have to make sure that we limit the search on the given number X. This is done by summing over all the opponents and rounds and see whether we then end up with enough games to fulfill that limit:

$$\sum_{T2 \in T \setminus T1} \sum_{r=1}^{R} (G_{T1,T2,r} + G_{T2,T1,r}) = X \qquad \forall T1 \in T$$

We then still have to make sure that we do not exceed the maximum number of games which are given implicitly by the number of games the user selected to be scheduled:

$$\sum_{r=1}^{R} (G_{T1,T2,r} + G_{T2,T1,r}) <= 1 \qquad \forall T1 \in T, \forall T2 \in T, T1 \neq T2$$

The same check will have to happen for the third case, when we fall between the SRR and the DRR:

$$1 <= \sum_{r=1}^{R} (G_{T1,T2,r} + G_{T2,T1,r}) <= 2 \qquad \forall T1 \in T, \forall T2 \in T, T1 \neq T2$$

For all of the games we will still have to address whether we have a home/away game pattern which is fair to the teams. We use the same formulation we used with the SRR, where the only difference is that we replace the $T - 1$ by X:

$$\sum_{r=1}^{R} \sum_{T2 \in T \setminus T1} G_{T2,T1,r} \leq \lceil X/2 \rceil \qquad \forall T1 \in T$$
$$\sum_{r=1}^{R} \sum_{T2 \in T \setminus T1} G_{T1,T2,r} \leq \lceil X/2 \rceil \qquad \forall T1 \in T$$

3.2.2 Breaks

In the common tournament it is possible to select whether you want to have a break between games. This means two games played by one team will be separated by a break, with a length of B. This is a setting for all the teams, a general setting. To make sure that the ILP solves this accordingly, we use the next formulation: For every team sum all the games in $B + 1$ subsequent rounds (depending on the length of the breaks) and for every combination we can find of $B + 1$ subsequent rounds, this can not exceed 1. Hence, a certain team can only play one game in $B + 1$ subsequent rounds.

$$\sum_{T2 \in T \setminus T1} \sum_{br=0}^{B} (G_{T1,T2,(r+br)} + G_{T2,T1,(r+br)}) <= 1 \qquad \forall T1 \in T, \forall r \in [R]$$

This constraint replaces the original constraint concerning the fact that we should play at most once per round, because this already limits it to once per $B + 1$ rounds.

3.2.3 Minimum or maximum number of games

In the common tournament, the competition and the multi disciplinary tournament (though it is hidden by the number of sports entered) we can set a minimum and/or a maximum number of games per round. This is modeled by summing all the games for a certain round and then check whether it is bigger than the minimum and smaller than the maximum.

$$\sum_{T1 \in T} \sum_{T2 \in T \setminus T1} G_{T1,T2,r} >= minimum \qquad \forall r \in [R]$$

$$\sum_{T1 \in T} \sum_{T2 \in T \setminus T1} G_{T1,T2,r} <= maximum \qquad \forall r \in [R]$$

3.2.4 Unavailabilities

An unavailability is a team which can not play in a certain round. It is a setting which is always team-specific, so we will have to implement a constraint for every unavailability in U, the set of unavailabilities.

The set of unavailabilities consists of two values; a team and a round. We will have to make sure that the given team can never be scheduled in the round given by the unavailability. We do this by making it unavailable for a certain team to play a game at home and make it unavailable to play a game away:

$$G_{T1,T2,r} = 0 \qquad \forall (T1, r) \in U, \forall T2 \in T, T1 \neq T2$$

$$G_{T2,T1,r} = 0 \qquad \forall (T1, r) \in U, \forall T2 \in T, T1 \neq T2$$

3.2.5 Soft constraints

To implement soft constraints, we add variables to the right side of the objective, to make the found solution worse. We also need some additional information implemented in our constraints, since they now can be violated. So we will give new formulations when the user selected a soft constraint on a certain part of the problem. The user has the following two types of soft constraints to select

- Number of rounds
- Number of games a team should play (only for 'Amount x', a SRR or DRR can never have less games)

3.2.5.1 Number of rounds

This soft constraints gives us the opportunity to use more rounds than the user selected. We will add two additional rounds and give additional punishment when we violate the rounds set by the user. In practice this means that we will add two variables for every match, one for each round. We will also make the set of rounds two rounds larger, since scheduling in those rounds can now lead to a viable solution. The soft constraints will not be violated unless necessary.

In this case we will therefore have to give a punishment when we schedule a game in one of the extra rounds (R1, R2). We add the following as an extra cost to the objective function:

$$\sum_{T1 \in T} \sum_{T2 \in T \setminus T1} H_r \cdot G_{T1,T2,r} \qquad r \in (R1, R2)$$

The higher the round, the higher the punishment is when we use that round. The two rounds are set as a standard in this case, just because it seemed like a fair amount, but this could have been any amount (even an amount given by the user). We already added the extra variables to our problem, by extending the rounds by two. We end up with a general scheduling problem, with just two extra rounds, when we schedule anything in those rounds it will reduce the quality of our solution.

3.2.5.2 Number of games

We can also let the user select a lower number of games for the teams as a soft constraint. We will only lower them all equally, thus everyone will play one game less than the user put in. If this is the case, we should punish the problem for selecting less games. We will have to change the games which can be removed from the schedule, by making sure that a viable solution can be found when there are less games than scheduled by the user.

So we change the constraint which limited on the X, to become:

$$\sum_{T2 \in T \setminus T1} \sum_{r=1}^{R} (G_{T1,T2,r} + G_{T2,T1,r}) = ((X - 1) + Y) \qquad \forall T1 \in T$$

The Y is now changeable, but when we set $Y = 0$, we will have to pay a big penalty. The Y can not become bigger than 1 (otherwise we will have to play more games than the user put in probably). It is the same for every constraint, that way we make sure that every team still plays the same number of games. So we get an extra constraint concerning the Y:

$$Y \in \{0, 1\}$$

We also need to pay a huge penalty when we select less games, since it is a huge change to the tournament. So we just want it to be acceptable when we can not find a solution for the tournament with the given X rounds. We add the following part to the objective function, where G could be any huge number. By selecting a huge G we make sure that we will only select if we have no other choice. Since we will not fulfill the input by the user the best way we can, we want to be certain not to do this if it is not necessary.

$$-(G \cdot (Y - 1))$$

The above function is the addition we will have to do to the objective function to include the punishment when we select less rounds. When we select $Y = 1$, no penalty is involved, when we select $Y = 0$ we will have an additional punishment of G.

3.3 Finding an opponent for a given round

This is the other part of the problem we are trying to solve. This leads to a one-round schedule, where everyone should play once (except when there is an odd number, where one team will have a 'bye'). So instead of searching whether we are able to schedule a certain combination of (team, team, round) we will have to decide which teams play each other.

3.3.1 Additional rules

This calls for a different kind of solving. Of course we still solve it using ILP, but instead of selecting a round for every match, we select a team. We do not schedule a full tournament, but will have to take into account the scores teams got previously.

It actually matters which team we select as our opponent:

- Make sure teams get a fair home-away pattern
- Let the team play other teams who scored alike

Of course there are also a couple of rules we will have to make sure we follow:

- Never play someone twice
- Make sure every team is scheduled only once per round
- We need to schedule exactly one match per team. It is not acceptable to have teams idle who could have played a match
- Minimize the total cost
- The home/away pattern must be fair, we can not let someone play all of its games at home or away

So in fact we end up with an assignment problem, where the focus is different compared to the round-solving type of tournament. It is much easier to find an acceptable solution, because there is a smaller number of constraints involved, but there is a bigger difference between the solutions in quality. Quality is the most important here, because we do not want to give someone an advantage by playing players who scored less points.

3.3.2 ILP-formulation

This asks for a different kind of ILP-formulation, where we do not have any constraints considering rounds anymore. Unavailabilities, breaks, maximum and minimum games, number of games all do not exist. On the other hand we have an addition to the soft constraints, where we will have to take into account how many points teams scored. Matches between teams who scored a similar number of points will get a small (or no) penalty, while teams with a high difference will get a high penalty. This way we force it to a solution where teams with nearly the same number of points scored so far will play each other.

We do this by introducing a new set of parameters: S_T1_T2. This set of parameters holds the specific score for every game, where S_T1_T2 will not have to be equal to S_T2_T1, since there might be a difference because of the number of games they played at home or away. The scores are computed by taking the difference between the scores of the combination of teams. The teams who already played each other will not be selected (we will emphasize on this later, in section 3.3.4) and therefore we will not compute those scores. The difference between the scores should be as low as possible for the total tournament, that is the best solution we can achieve. We assume we have a specific round R here.

minimize $\sum_{T1 \in T} \sum_{T2 \in T \setminus T1} S_{T1,T2} \cdot G_{T1,T2,R}$

subject to

$$\sum_{T2 \in T \setminus T1} (G_{T1,T2,R} + G_{T2,T1,R}) = 1 \qquad \forall T1 \in T$$

$$G_{T1,T2,r} \in \{0,1\} \qquad \forall T1 \in T, \forall T2 \in T, T1 \neq T2$$

Our solution is only viable if we play as many games as possible. This is because playing as many games as possible is a precondition for this type of tournament. So that is why we now use a 1 in the constraint to check whether we play exactly once. Every team will play every game, where we will add a bye team when we have too little teams.

3.3.3 Removing half of the variables

When we have the above formulation, we can remove half of the parameters. We can do this because we will only play each opponent once (e.g. there will never be a return-game) and we will minimize depending on the score. We will only have to take into account one of the variables per game. This is the variable with the lowest score, which will depend on the number of times someone played at home or away. The variable with the higher score can therefore be left out (or set to zero).

3.3.4 Played games

You can consider these played games as unavailabilities. We will never play another team twice. So we can block the teams who already played each other (both the home and away games between those teams, since this type of tournament means that we can only play each team once in the total tournament).

We have a set of games which have already been played: PG (played games). This is a set containing all two teams who played each other. $PG = [(T1, T2)]$, the round we are scheduling is called R.

$$G(T1, T2, R) = 0 \qquad \forall (T1, T2) \in PG$$

$$G(T2, T1, R) = 0 \qquad \forall (T1, T2) \in PG$$

The difference between these previously played game versus the unavailabilities in the other type are two sided: this setting exists because of the rules set by the tournament, while the unavailabilities are added by the user and this setting is for a specific game, while the unavailabilities block an entire round for a certain team.

3.3.5 Home/away patterns

When we schedule the tournament one round after another it is really important to handle home-away patterns. Since such kind of tournament is mainly used for chess or checkers, this can give a huge advantage [38] [1].

So we will have to make sure that people who already played many home games will get an away game and the other way around. We never want any player to have an advantage which will exceed two to the average. This could still lead to a difference, but when we set the difference to one, we will end up with a schedule which will stay too fixed. So instead we give a penalty when someone who already had one more home game than away a penalty. When a player has had two more home or away games, we will have to block the other home or away games, since the schedule will then become too much of an advantage to them. The teams with such an advantage will be put into a separate set: CA (constrained away) and CH (constrained home). The round is left out, since it is a fixed value:

[1]Accessed 3-6-2014: white has got a 37.44% lead compared to 27.48% for black

$$G(T1, T2) = 0 \qquad \forall T1 \in CH, \forall T2 \in T, T1 \neq T2$$

$$G(T2, T1) = 0 \qquad \forall T1 \in CA, \forall T2 \in T, T1 \neq T2$$

We will also have to change the scores (which were in the parameter S_T1_T2) to add a penalty for playing an additional home or away game. So we will decrease the score for a certain game S_T1_T2 if $T1$ played an extra home game. This will then be used in the removal of the variables and only the best scoring variable will be kept in the calculation.

This also means that we will never have schedules where people play many games at home or away after each other. The maximum number of games one team can play at home/away after each other is three, namely when he first equalizes the number of home and away games and then gets two home games.

The maximum number of home or away matches in a row is four. This is the case when we have (for instance) the following pattern (H is home, A is away): $HHAAAA$. The seventh match will always be a home game and there will be a penalty for the sixth game, which we also had to play away, while he already had an extra away game.

3.3.6 Byes

In our problem we can have byes, when we have an odd number of teams. If the number of teams is even we always want to schedule $T/2$ games per round. But if it is odd, we add one bye. The rule is that a team can never have more than one bye in a whole tournament. Therefore there is an additional variable (you could call it an extra team), which will make sure that that never happens.

We can use the fact that we can only schedule one game against every opponent once very well. Because we can just include the bye as an extra team and the ILP will take care of the rest. It will make sure that a team only has one bye and prevent it from having multiple ones.

A bye is always a win, the team gets the points you would get when you win a game. We have a small incentive to make sure that a bye is scheduled in the lower teams in a certain tournament. Otherwise we might end up with a bye for the highest scoring team so far in the last round, which will therefore always win the tournament without a doubt. That is of course a situation we do not want to have, so we will give the bye an incentive to play against the lower-scoring teams, by lowering the scores when playing a game against a bye when we schedule the tournament. We also block the bye for the top twenty-five percent of the tournament, to make sure that never happens.

If there is a bye in a certain tournament, the user will be asked to give in the points the team who plays the bye gets. For football (which is generally not the case with this type of tournament) it would be three points, for checkers it is one point. It counts as a home-game, since winning the game without playing already has an advantage. You could also say that the bye 'team' always plays away.

3.4 Finishing loose ends

We still have some loose ends we will have to visit before we can finish the ILP chapter. These loose ends concern the time-constrained tournament and the multiple disciplinary tournament. These two tournaments have a unique thing, which should be fixed after we get the scheduled tournament back. For the time-constrained tournament that is the time, for the multiple disciplinary tournament it is the different kinds of sports.

3.4.1 Time-constrained tournament

For the time-constrained tournament, we will have to split the rounds in sub rounds. This means a round is not (for instance) round 7, but instead round 7.1. These sub rounds correspond to times on the day. The day is the parent round, so in this case round 7 would be a specific date. With these sub rounds, we can block certain specific times for a team. So if you say a team can not play before 12:00, he might miss sub rounds 1, 2 and 3. This of course means we will need a schedule for the day, representing at what time the sub rounds start and at which they end. This schedule could be generated by information given by the user, but it was decided not to do so. It does not change the problem a whole lot and therefore it was decided to use a fixed day schedule.

We will then have to change some settings concerning these sub rounds, where we block rounds 7.1, 7.2 and 7.3 for example. The resulting schedule will then also be returned with specific times, so the resulting tournament will also have times when games should be played.

3.4.2 Multiple disciplinary tournament

For the multiple disciplinary tournament we still need to fix some information concerning the different kinds of sports. We end up with a schedule where every team plays a SRR. This is just an assumption we made when we created this type, because in our opinion this is the case with these multiple disciplinary tournaments. The type of tournament demands that everyone should play each other once and every team should play each sport once. So we will need to do some extra work to make sure we let every team play every sports once.

We do this by adding an extra part to our variables. This means our decision variables become $G(T1, T2, R, S)$, where S is the sport. We then have additional constraints to make sure we only play each sport once.

$$\sum_{T2 \in T \setminus T1} \sum_{r \in R} (G_{T1,T2,r,s} + G_{T2,T1,r,s}) = 1 \qquad \forall T1 \in T, \forall s \in S$$

When we sum all of the games played by a certain team, he can only play every sport once.

We will of course also have to change all of the constraints we had before, since we will now also have to take into account the sports. So all of our variables will become $G(T1, T2, R, S)$ and we will have to add a sum at the beginning in order to check them for all the sports.

Chapter 4

Constraint programming

In this chapter we will implement a constraint programming (CP) approach to solve this problem. What follows in the first subsection is a small introduction to constraint programming, then our approaches and some information about the propagators we used.

4.1 Introduction

In this section we will give a short introduction into constraint programming and how solving using CP works. We will also shortly explain why we decided to use CP, besides ILP, for solving the tournament scheduling problem. We will have to solve a so-called *Constraint Satisfaction Problem (CSP)*. We use the definition about what a CSP is from the "Handbook of constraint programming" [39].

A CSP is defined by a triplet of variables:

- X: A finite set of variables
- D: A finite domain; one for each variable
- C: A finite set of constraints between the variables

For a standard sport scheduling problem:

- X: The matches we will have to schedule
- D: The rounds a match can be scheduled in
- C: Constraints on the matches (teams only play once per round etc.)

In order to find a solution to a CSP, we will have to make sure that we satisfy the following constraints:

- Assign for each variable one value out of the domain for the variable
- None of the constraints is violated

We decided to use CP to solve the sport scheduling problem, since in our instance of the problem we have a lot of additional settings which are constraints on our problem. For example, when we have a minimum number of games, we have an additional constraint on our original problem, namely that we should not have less than the minimum number of games in a round. We can do this for every setting and therefore it is convenient to use CP, where the constraints are used as a guideline to solve the problem.

4.1.1 Propagators

Propagators are the backbone of solving the CSP using CP. They are functions which change the domains and the constraints in order to find a solution, without removing any possible solution in the process. In our sport scheduling problem, for example when we schedule a certain match, we will have to make sure both teams in that match should not play any other game in the same round. So we have a propagator which will remove that round from every variable containing one of the teams.

Propagators are functions which can influence the CSP in two different ways:

- By removing a subset of the domain for a certain variable
  This can be applied when we know for sure that a certain value in the domain can never lead to a viable solution.
- By enforcing the constraints
  The stronger the constraints are, the more information we can draw from them. So the more we are able to enforce them, the better it is for finding a solution.

The idea of solving a CSP using CP is that we assign a value to every variable. The value must be in the domain of the variable and when we have assigned a value to each variable, we should not violate any of the constraints. So it is clear why taking a subset of the domain will bring us closer to a solution. We will never increase the size of the domains, so the domains will always be smaller or equal after the propagator. When the domains become smaller, the possible values for a variable decrease and therefore the problem becomes easier to solve.

Enforcing the constraints is something which is really important to keep in mind. We start with no assignment for any variable and the more values we assign, the more information we get about our solution. When we know certain values for sure, we can change our constraints by filling in the values. For instance; there is a problem where we want to assign a certain value x (in our case round x) at most y times (when we have at most y games in a certain round). After we assigned value x once, we can only assign it $y-1$ times to the remainder of the variables. So using a propagator, we actually enforce the constraint $SUM(variables[x]) < Y$. These changes will affect our probability to decrease the size of the domains in the end. After we assigned the value x y times, we know that none of the remaining variables can have value x. Thus we can remove x from the domain of the remaining (still not assigned) variables and remove this constraint from our problem.

4.1.2 Solving a CSP using CP

The result after solving a CSP using CP is thus an assignment for each variable, but the question is how we reach such an assignment. With CP we solve the problem by using the constraints we have. The constraints give us information about variables which influence each other and how they do that. Some problems which are generally solved quite well by CP can be solved without (or with very little) 'guessing'.

With guessing we mean that we sometimes get stuck trying to find an assignment for every variable. None of the variables will have either an empty domain (when we can stop the search, because we will never find a viable solution) or a domain with one value (when we know that we will have to assign that value) and it is impossible to eliminate any more values from the domains. We will then have to branch on the values in the domain of one variable. We will do this by making branches for every value in the domain of a certain variable. We will then pick a branch and continue the search in that branch, as if we were able to select the value for the variable. If we have checked the whole branch and are sure we will not be able to find a solution anymore, we will go back to this point and continue the search in another branch. This is called backtracking and in this way we are able to continue the search and know that we will find the solution if it exists, when we give the algorithm enough time.

Of course there might be multiple solutions to the problem, depending on the problem. For instance for a sudoku there might just be one, while for a scheduling problem there might be thousands (or even more). The search space could grow enormous (again; that depends on the problem). What we do when we reach a solution depends on the problem description. If we let the algorithm search on, we will eventually find all the solutions. Generally for this kind of problem we have to find a single solution, since we want a solution to our problem and we do not care about all the solutions. There are some examples where we do, when problems always have a lot of solutions. It is not hard to find a single solution, but it is much harder to find all the solutions (or count how many there are).

We will use forward checking to solve the CSP. Forward checking is a technique where we will assign a value and directly evaluate whether we can still find a viable solution given the already known values and the new known value. The earlier we are able to establish that a branch will never lead to a viable solution, the earlier we can start the backtracking. This can be done in different ways, though the general way is when any of our variables ends up with an empty domain. If that is the case, it means that there will never be an assignment for every variable in that branch and we can cut it. A nice example is the n-queens problem [40].

The problem becomes a kind of search-tree, since we will have several branches and we will have to save which branches we will still have to search etcetera. This means we will have to decide which kind of tree-walking algorithm we will use. Depending on the problem we might use a breadth-first search or a depth-first search. If we want to find any solution, a depth-first search will generally be faster. If we want to find all the solutions, we could also use a breadth-first-search, since we will need to search every branch anyway.

Given the fact that this is a tree and we will have to search the branches, we also need some sort of backtracking. Which branches are still open and which branch should we work on next? There are a lot of techniques that can be implemented here. Do we want to backtrack to the previous branch or do we want to backtrack to a branch with a higher possibility to find a solution? The problem with this kind of solving is that it is hard to score how likely we are to find a solution. This of course depends on the problem, but for our problem it is hard to do so. We do not implement such kind of scoring the branches, the general way to do this is by just backtracking to the previous branch and try to find a solution for that branch.

4.1.3 Soft constraints

When making this constraint programming solution, we know that we will eventually find an acceptable solution if it exists, but we do not know whether it is the best solution. When we have soft constraints or try to implement the tournament with a single round, there might be better solutions, where we violate less constraints and thus find a better solution (one which fits the input from the user better).

In the end we decided in our implementation to use the CP itself to find a better solution, by implementing the score. The idea behind the score is that we will keep on searching, but can now limit branches which will never give us a better solution than the solution found previously.

To make this even stronger, we add the score we get minimally from the remainder of the rounds. We don't search the tree and find the lowest value, but just go through every game and add up the lowest value. This way we will be able to backtrack as early as possible.

It can then be used as a backtracking rule and therefore the search will speed up and we will (hopefully) be able to find the optimal solution (given the constraints). In the end we will have to stop the search at a certain point. We give it a certain amount of time (which can be set in the code) and we will return the best solution found so far.

Of course we could have used other techniques as well. After we found a viable solution, we know that we can most of the time switch around games in that schedule. For instance we can switch full rounds if none of the teams is blocked from playing in the other round. This means we actually get a whole different kind of search, namely a search to find a local optimum after we found a viable solution. There are a lot of techniques we could use here (for instance simulated annealing) to find a local optimum from our viable solution.

In the end we decided to go with the constraint programming approach, since it seemed more logical. We are already solving the problem using CP and we will just have to let the search continue, trying to find a better solution. We will backtrack the games we used to find a solution and then use it to find the best solution in our time-window. When we exceed the time-limit, we will stop the calculation and return the best result found so far.

4.1.4 Backtrack rules

In a CP-solver it is important to stop searching branches when we reach some point where we know that no more solution exists. The basic backtrack rule is to quit when the domain of any of the variables becomes empty. When the domain is empty, we can keep on trying to find a solution in that branch, but we already know we will never be able to find one where we can assign a value to that variable.

In CP we will never add values to our domains after we started the search, unless we backtrack. If we would add values to the domains, we would not know whether a choice we made previously would still be the only choice we have for the assignment of the values in the end.

Problems can have additional backtrack rules, when we already know that we will never be able to fulfill all the constraints in our problem, no matter what we use as values. This needs some problem and solution analysis, where it depends a lot on the problem whether or not we will be able to introduce additional rules. Putting time in finding these inconsistencies is most of the time worth it, since we will be able to quit the search earlier.

We will, however, have to take into account that analyzing the problem will also cost extra time. We will need to weigh off whether the extra checks are worth the time invested, since the best check is of course to see whether we can find a schedule, but that would take the same time as just finding the schedule. So it is important to check whether some validations are worth their time when we execute them.

4.2 Two types of problems

Once again we split the problem into two subproblems: one where we assign rounds to matches and one where we assign teams to other teams to play in a in advance specified round. When we assign rounds to matches, we will schedule the whole tournament in advance, while we will only schedule a certain round when we need to assign teams to each other. This is necessary because we are simply trying to solve another problem and therefore we will have to split them. For instance in the first our variables are matches, in the second the variables are teams. More information about why we need the split and what the different subproblems are has been explained in Chapter 3.2.

4.3 Finding the round for a match

Recall that we will have to assign rounds to matches in this problem. The resulting schedule will consist of all the matches that will be played and a round for each match.

We will start by formalizing the problem as a problem instance and then explain the propagators used. Recall that the problem instance consists of three variables. We will have to give these variables meaning in order to be able to solve the problem using CP.

The variables consist of all the matches we will have to schedule. The domain for these variables consists of the rounds. Depending on unavailabilities (and other influences), the domain can be different for each match. If there are no influences (like unavailabilities), the domain is the same for every match.

We then of course still have the constraints, which go from the basic constraint 'a team can only play once per round' to minimum and maximum number of games. We will explain later how they influence the solving. There are roughly two groups which help us to get to the right solution: propagators and backtrack rules. Both will help us to enforce the rest of the constraints and to make sure we find the right solution we are looking for.

4.3.1 Propagators

Below we will define the propagators. The propagators will most of the time change the domains, though some of them will also change the constraints on the problem. These updated constraints may then again be used in the backtracking rules.

4.3.1.1 Match scheduled

When we schedule a certain match, the domains of the other matches will have to be changed. Of course for the scheduled match, the domain will be emptied, with exception of the round we scheduled it in. We change the following domains accordingly ($M_{x,y}$ is a match, with x and y as the teams and we schedule it in round z):

$\forall f \in T, f \neq y : D_{x,f} \leftarrow D_{x,f}/z$

We remove the round z from the domain of every match where x plays at home, unless it is the match we just scheduled.

$\forall f \in T, f \neq x : D_{f,y} \leftarrow D_{f,y}/z$

We remove the round z from the domain of every game where y plays away, unless it is the game we just scheduled.

$\forall f \in T : D_{f,x} \leftarrow D_{f,x}/z$

We remove the round z from the domain of every match where x plays away.

$\forall f \in T : D_{y,f} \leftarrow D_{y,f}/z$

We remove the round z from the domain of every match where y plays at home.

The domains are sets of course, so we might try to remove round z while it is already not in there anymore, but that does not matter.

4.3.1.2 Match scheduled with breaks

When we have a one-round break in our tournament, the above result is a little different. Instead of removing only round z, rounds $z-1$ and $z+1$ are both removed as well. So, when we schedule match $M_{x,y}$ in round z again, we end up with:

$\forall f \in T, f \neq y : D_{x,f} \leftarrow D_{x,f}/\{z-1, z, z+1\}$

$\forall f \in T : D_{f,x} \leftarrow D_{f,x}/\{z-1, z, z+1\}$

$\forall f \in T : D_{y,f} \leftarrow D_{y,f}/\{z-1, z, z+1\}$

$\forall f \in T, f \neq x : D_{f,y} \leftarrow D_{f,y}/\{z-1, z, z+1\}$

When we have a longer break, we will just have to remove more values from our domains the same way. So when we have a 2 round break, it becomes removing rounds $z-2$, $z-1$, z, $z+1$, $z+2$.

4.3.1.3 Symmetry removal

When we schedule a certain game, we remove all the rounds which are symmetric and then save the state after the removal of the rounds. This way we will be able to skip some rounds which will never lead to a better solution than the solution we already had.

$D_{x,y} \leftarrow D_{x,y}/\{symmetric\, z\}$

This is a propagator which actually influences our current state instead of our future state. It will make sure that we will not try to find a better solution in a branch, while we already know that no better solution will exist in that branch. We will emphasize more about the symmetry later in a separate section, 4.3.3.

4.3.1.4 Constraint changes - maximum

When we schedule a match in round R, we know that there is an extra match in that round. With this information the constraints can be strengthened. We will need an additional variable SM. SM just represents the scheduled matches so far. The constraint is normally:

$$\sum_{x \in T, y \in T, x \neq y, M_{x,y} \notin SM} G_{x,y,R} <= maximum$$

After we scheduled three games in a certain round, the following constraint holds.:

$$\sum_{x \in T, y \in T, x \neq y, M_{x,y} \notin SM} G_{x,y,R} <= maximum - 3$$

The same update on the constraints can be done to the minimum number of games we need to schedule, but how that works is obvious. When the right hand side of the constraints becomes 0, this means we ended up with a round where no more games can be scheduled. That brings us to the last propagator.

4.3.1.5 Maximum number of games reached

When we reach the maximum number of games in a certain round R, we know that none of the games may ever be scheduled in that specific round. So we can remove that round from every domain in our search algorithm. SM is once more the set with the already scheduled matches.

$\forall x \in T, \forall y \in T, x \neq y, M_{x,y} \notin SM : D_{x,y} \leftarrow D_{x,y}/R$

It is interesting to note that even though we implement the maximum number of games as a propagator, the minimum number of games is a backtrack rule. This is because we will use the information from the maximum number of games to make the domains smaller and use the information from the minimum amount to block certain partial solutions which will never lead to a viable solution. So this is a nice example of how the propagators and backtrack rules work with each other.

4.3.2 Backtrack rules

In this section we will explain which backtrack rules we apply to solve this CSP instance. There are a lot more backtrack rules compared to basic problems. As we already explained, it is important to find a rule to backtrack as fast as possible when we have a CSP instance requiring a lot of branching.

4.3.2.1 Empty domain

The search on this specific problem can be stopped when we reach an empty domain. Just as with any form of CSP problem. When we reach an empty domain, we know that we will never be able to schedule a tournament within that branch.

4.3.2.2 Score/soft constraints

The score is also a backtrack rule, as we can stop the search in a specific branch when the score exceeds a score found previously in another branch of the problem. The score was handled more extensively in section 4.1.3.

4.3.2.3 Tournament impossible to fill

The third way we can backtrack in this problem is by checking whether we can still schedule the remainder of the games for a certain team in the tournament we are trying to schedule. This can be done in two different ways; using a max flow and using a simple count. The max flow will be discussed in chapter 6.

The count is an easier way to find out whether it is impossible for us to schedule the remainder of the rounds. If a team can play his remaining games in x different rounds and only y ($y < x$) rounds are still open to him, we can stop the search. We can only schedule one game per round and thus we will never be able to find a solution, since he stall has to play more games than rounds are free to him. Therefore we can stop the search and backtrack to the previous branch.

4.3.2.4 Minimum round constraints

The last rule is when we have a minimum rounds constraint. We might schedule in such a way that we can never fulfill that constraint. Therefore we will have to stop searching for a solution when we end up with too little games to fill up all the rest of the schedule. If this happens, we will stop the search and start backtracking.

So there are quite a lot of backtracking rules which are specific for this problem. The earlier we are able to use the backtracking, the faster (and thus better) the algorithm works.

4.3.3 Removing symmetry

With scheduling and timetabling problems we can expect problems to arise concerning the symmetry in the solutions. The problem is that we may already know in advance that setting a certain round for a certain game, will indefinitely lead to the same solutions. Because we are here just scheduling rounds for sports, we can sometimes find another viable solution by switching total rounds.

When we have decided which rounds are symmetric for a certain match, it does not matter which of the selected rounds we assign to the match. Since they are all symmetric, we should only try one of them and if we are not able to find a solution with that assignment, we will not find it with any of the symmetric rounds. That is because all the solutions will be symmetric and can be reached by switching rounds and/or multiple games.

This leads to a huge improvement in solutions we will not have to search, because we already know that they will never lead us to a solution which will differ from the previously found solutions. Especially without periods of unavailability, all the rounds in the first game we schedule are symmetric. If we are unable to find a solution when we schedule it in round 1, we will never find a solution for the problem and thus can stop the search.

So we remove the symmetry by making lists of equivalent rounds and by just searching one of them. In this way we can in the end crop a large part of the search-tree. We could miss out on solutions, but we do not want to find the maximum number of solutions, but just the best. We will always find the optimal solution this way (if we keep on searching long enough).

There are a couple of things which can break the symmetry removal we implemented. This means the rounds which were previously symmetric are not symmetric anymore and therefore we will have to search all of them anyway. We continue with the things that block the removal of symmetry.

4.3.3.1 Unavailabilities

Because of unavailabilities, selecting a certain game can lead to the fact that no other game can be scheduled in that round anymore. We can not just simply switch two rounds, since the unavailability will block certain matches for that round. Two rounds are not symmetric if they don't have the same unavailabilities. If they do have the same unavailabilities, the same games can be played in the rounds and therefore they are completely equivalent and symmetric. We can then remove the other round for the domain we will still have to search in.

We can decide whether we still have symmetry in two rounds in the following way:
If in two rounds, the exact same games can still be scheduled, these two rounds are symmetric.

4.3.3.2 Other scheduled games

When we have other games scheduled, the rounds are of course no longer symmetric. The number of games that we can schedule could differ and the games we can schedule in that round differ. But two rounds could still be symmetric, even though games where scheduled. For instance when one round has a game where team 1 plays team 2 and the other round where team 2 plays team 1. Both can still schedule the same games and they are therefore symmetric.

4.3.3.3 Breaks

By introducing breaks, removing symmetry becomes considerably more complex. When we have breaks, two rounds influence each other. So the easy to grasp idea of 'when we can schedule the same games in a round' is lost. This occurs especially when combined with the unavailabilities or the other scheduled games. They influence each other greatly and it is hard to find a symmetric solution. Removing the symmetry is therefore also removed when the user decided to use breaks in the problem.

4.3.3.4 Soft constraints

The soft constraints also make it much harder to find symmetry in our solution. Because we might say that two rounds are exactly equivalent, though in one round there is a certain cost for scheduling the game there. This directly means they are not equivalent, because we do not know whether we will have to pay that penalty for any of our games. The rounds can not be switched around with no costs in the end. In fact this also arises the problem where games can influence each other, because of the penalties when we set certain games. So it is much harder to encounter which of the rounds are actually symmetric for a certain game, because none of the other games should have to pay a certain penalty for any of the two rounds. If they do have to pay a penalty (and the penalty is not equal), the rounds are not symmetric and thus we can not remove the symmetry for free.

So when any form of soft constraints is selected, the normal removal of symmetry is not used.

4.3.4 Different kinds of tournaments

The above explanation works well for the DRR, but when we have an unknown number of games or we do not know who plays at home and who plays away, we end up with problems. For a SRR we do know which games we have to play, we just do not know who we will play at home or away, this will be decided afterwards.

When we have a fixed number of matches it is even harder. We will just have to make sure that we schedule the subset of games. Normally we will have to find a value for each variable, so we will have problems because we will now just schedule a certain number of games. The basic backtrack rule of CP was: when a domain is empty; no solution can exist. If we can not use that basic rule (hence we have a fixed number of matches, some of the variables will not get a value), we will only be able to backtrack much later and therefore end up with a slow algorithm. If this is the case, we decided to let the ILP solve the problem.

4.4 Finding an opponent for a given round

In this section we will discuss the problem where we have to find an opponent for a given round. This is a matching problem, though when we get additional settings we might end up with a problem we can not fix as a matching problem. When we do have a simple matching problem, this means we can solve it exact. This section is about the CP implementation, the exact solution is discussed in chapter 5.

The CP implementation for this problem is quite different compared to the problem where we want to assign rounds to matches, since we have got a different kind of assignment problem. We will have to assign which team plays which, instead of assigning a round to a match. This means that instead of the variables and domains we just had, we also have different values for those.

The teams become the variables and the other teams become their domains. Recall that we will only play each opponent once in this type of tournament. So after we scheduled the first round, the domains will already be different between the teams, since we will not be able to play against teams we have played before.

We also of course have the other rules concerning this type of tournament, which can be found in chapter 3.3.1.

4.4.1 Selecting the teams

We will of course use CP to find the solution, but to get close to a good solution we will have to make sure that we use some kind of heuristic. If we do not, we will search a lot of branches who will never lead to a good solution. What we want to do is to try and make combinations of teams who have a low score and again backtrack if no solution is possible anymore (when the last two teams already played against each other).

Besides that we can get a score for the tournament found after we schedule all the games. We will use that score to stop searching in the branches which will never lead to a better solution. So that is also why it is important to use a heuristic to find the best score, because the closer we get, the more branches we will be able to cut. We implemented this as a separate function, so we can actually change this easily. We implemented the following heuristic:

> We select the highest scoring team we did not match yet and will have to find the best opponent to play against it. To get the opponent with the lowest score difference with this team, we search its domain and try out every combination of matches, away and home. The opponent with the closest score will then be returned as the opponent.

This is an easy heuristic, we could just as well have chosen the lowest team and work our way up from there. We choose one of the sides to start with, because we do not want to end up with the best scoring and worst scoring team in the end. If they are left we know that that score will be really bad (and it is probably an unacceptable solution, which will require a lot of backtracking). Therefore the decision was made to start at one of the sides and work our way to the other end.

4.4.2 Propagators

As the propagators for this type of tournament, we only have two propagators. We do not have a maximum number of games, though we do have a minimum. But just as with the other types of tournaments, we will handle the minimum number of games in the backtracking rules and not in the propagators.

4.4.2.1 Scheduling a match

After we schedule a match, we will have to make sure that the teams involved in that match will not play another match in that certain round. Since we only schedule one round, we will just have to make sure that they are removed from the domain of every team. For instance if we schedule the match $M_{T1,T2}$, the following happens:

$\forall T \in T, T \neq T1, T \neq T2 : D_T \leftarrow D_T / \{T1, T2\}$
$D_{T1} \leftarrow \{T2\}$
$D_{T2} \leftarrow \{T1\}$

4.4.2.2 Removing team from history

After we schedule a match, we will also have to make sure we save the state previously found. We remove the teams from each others domains and then save that as the state. When we return to the position we backtracked to before, we will make sure that we will select a team which is as close as possible, but we will never select a team we tried before to find a solution with. This way we make sure that we will only search every combination of teams maximum once.

This is actually just the generic way of removing information from the domain after we selected a certain branch. We have to keep in mind that we do remove the teams in an order which is specific to the domains given. This differs from the other type of tournament, where we just select the lowest round available and put it in that round. Here we use the heuristic to select the closest team and choose that branch.

4.4.3 Backtrack rules

In this section we will explain which backtrack rules we have in this CSP instance.

4.4.3.1 Empty domain

We again have the empty domain as a backtrack rule. For this type of tournament something like that can only happen after we scheduled a lot of games. These games will be blocked from our domains and therefore we might get empty domains.

It happens much less often compared to the other type of CP-solving, because there are a lot of options normally and there is only a small chance we will actually empty the whole domain, because of the fact that we have few propagators and few constraints. Though especially in later rounds, we get more information about matches which are already scheduled.

4.4.3.2 Score

The score is clearly the most important backtrack rule after we found an assignment of the variables. We will have a score for the total schedule and we can use that score to improve the search for a better schedule. When we have a certain score, we will check whether we can still get a lower score at a certain point in the tree.

We will save which team has got a minimum score as opponent for every team, that is the minimum we should add to get a full schedule. We will of course have to halve this result, since two teams will play against each other and therefore only one score for every two teams should be added. If this value is already over the minimum amount we found before, we know that we can stop searching.

4.4.3.3 Viability

Our solution can only be viable when we schedule as many games as possible. When we have an even number of T teams, we will need $T/2$ games in our scheduled round. When we have an odd number, we may have $(T-1)/2$ games, where the last team will have a bye (he will win, without playing anyone).

If we combine the formulas, we know that we need $\lfloor T/2 \rfloor$ games for a round to be viable. If we find a round with less matches as a solution, it is not viable. So we can use this in our advantage to calculate whether we need to backtrack or not.

We can check whether it is still possible to match every opponent against an opponent using a max flow. We can do this by matching all of our teams against another team. We create the max flow by setting a path from the source to every team, from every team to every team (he can still play, hence which teams are in its domain) and from every team to the sink.

We run the max flow and we must find twice the specific number of games we still have to schedule. We might find $T1vsT2$ and $T2vsT1$, so that is why we find twice the number. If we find less games, this means that we actually will never be able to schedule the round and we can stop the search.

4.4.4 Symmetry

In this problem there is actually little symmetry. We could possibly switch some of the games in the first rounds, but towards the end the amount of symmetry is reduced automatically, because scores will become different between players.

We did not implement removing the symmetry at all, because it does not matter when we try to find a best solution. We will find it, it could only be that we will search branches which will never lead to a better solution. But since the matches influence each other a lot (when a plays b, a can not play c), it means that we would need to do a lot of work to calculate whether we could remove symmetry in our problem.

Therefore symmetry is not removed in this subpart of the problem. The problem itself is also smaller, we only have to fill in every team once, against every opponent once. Our score-function will hopefully be able to make sure that it will stop searching as quickly as possible. We can not make any guarantees about that, because there are definitely partial problems where the score will not help us to cut a branch as soon as possible.

4.4.5 Home-away patterns

We will use the same ruling as with the ILP-formulation concerning the home-away patterns. We use their information to block certain teams from playing another game at home or away and we use the information before that already in the score. More information can be found in chapter 3.3.5.

4.5 Loose ends

We have two loose ends, but one of them has already been explained in Chapter 3. This is the time-constrained tournament, where we decided to have a fixed day schedule. More information can be found in Section 3.4.1.

The other loose end concerns the multiple disciplinary tournament: we still have to assign sports to the scheduled matches. We do this by changing our variables and not only by deciding in which round two teams play, but also which sport they play. We do this by adding a constraint we will have to check: every team can only play every sport once. This means we can also try to find inconsistencies concerning these sports.

So we keep an extra set of variables, which holds for every match which sports could still be assigned to it. If ever a set becomes empty before we scheduled all the matches, we can stop the search. After we assigned a certain sport, for both contenders we can remove that sport from all of the matches that team will still attend.

Chapter 5

Polynomial solutions

Some instances of the problem can be solved using polynomial solutions. We did not implement these, since CPLEX solves these instances really fast as well and they only work for specialized instances of the problem, where we no (or little) settings.

5.1 Bracelet method

When we need to schedule a SRR tournament without settings (unavailabilities, maximum number of games around, breaks), we can use the bracelet method. We put the teams in the order shown in Figure 5.1.

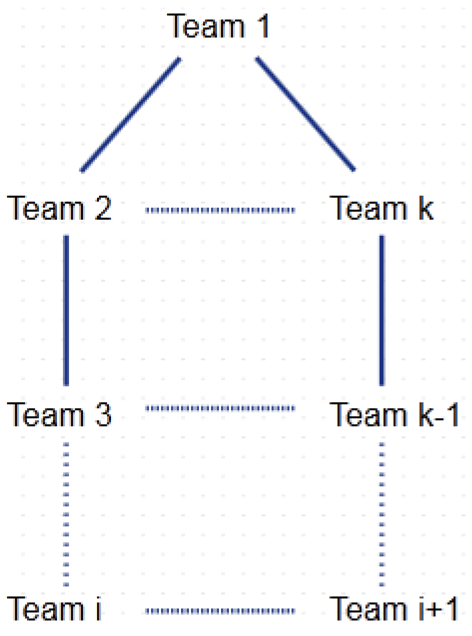


FIGURE 5.1: A bracelet tournament

We will then shift the teams to the next spot, where team 2 will go to the spot of team 3 and team k-1 will go to the spot of team k. We continue this way, until we have enough matches and have found a working schedule. This image is when we have an odd amount of teams (otherwise team 1 would not be at the top), when we have an even amount of teams, we will have to take one of the teams out of the bracelet. This extra team will play against the team on top of the bracelet.

5.2 Minimum cost maximum matching

The chess-type tournament is in fact just a matching problem. We need to match which teams play each other in a specific, already known, round. This means we should just match which teams play against each other. For every set of teams, we have a score (calculated from the scoring by both teams so far and how their home-away pattern is). We still need to schedule as many games as possible (which is always $\lfloor \#teams/2 \rfloor$). This score is used to calculate the lowest score for the total matching, which leads to a minimum cost maximum matching.

5.2.1 Graph building

To build the graph to calculate this matching, we start by creating a node for every team. We will have edges between the nodes which will represent the matches. Every match will have a certain cost which depends on the score both teams got playing the previous games in the tournament. Two teams will not play each other more than once, this means we will only have either the home or the away game. We will only add the arc with the lowest cost, we can afterwards check which team plays at home if the arc was selected to be played. When they both have the same cost, there is no difference and we will just select one of them. We will do this for every pair of teams, and add all the arcs between every two teams which have never played against each other. After we have added all of these, we will have a graph which represents the teams which can still play each other.

5.2.2 Solving the problem

The maximum matching problem has been solved by Edmonds [41]. It will find a maximal matching using the idea of augmenting paths. Augmenting paths are paths which start at an unmatched node, only visits matched nodes and end at an unmatched node. If that is the case, it means we could make the matching larger by taking all non-matched edges, remove all unmatched edges, and thereby improve the size of the matching. When no more augmenting path exists, this also means we found the maximal matching.

This is not the total algorithm, since we might have cycles in our problem. Normally they don't influence it, unless there are blossoms. Blossoms are cycles with $2k+1$ nodes where k edges belong to the maximal matching and the remaining node is part of an augmenting path. These blossoms can be contracted, all nodes will be contracted into one node. We will then continue the search for an augmenting path. If we find any augmenting path, one of two sides of the blossoms can be taken to add extra nodes to the matching.

Using the above algorithm we can solve the maximum matching problem, but we still have to make sure we find the minimum cost maximum matching. Lovasz [42] explains how this works in Chapter 9 of his book. He uses the above function, created by Edmonds, as a subroutine in a two-phased algorithm. The first phase is used to prepare the graphs so that they have a perfect matching. The second phase is used on the results from the first phase to find the minimum cost perfect matching using calls to the above function by Edmonds.

Chapter 6

Max flow

When we have a constraint programming approach, it is important to make sure we block branches from the search tree with no viable solutions as soon as possible. We can use max flows to check whether there still is such a viable solution. The idea was taken from a paper written by de Werra [35]. In his paper he uses a maximum weight matching to find whether for every element in his problem (in his example teachers, in our case teams) we can still find a solution. We use it in a slightly different matter, but the idea is the same.

The problem we are trying to solve using CP will be solved easier when we have to backtracking as little as possible. What we want to do is to signal as soon as possible that no solution exists. We can cut off the branch there totally and therefore stop the search. The earlier we can do this, the better it is for the runtime of the algorithm. For this detection, we use a max flow. This max flow will detect whether we are still able to schedule the remaining games for a certain team T. If we are unable to schedule all the matches for any of the teams in our problem, this means we are unable to find a solutions for the problem in total.

The rest of this chapter is about how the implementation works, the results and why we decided to leave it out in the end.

6.1 Basic max flow implementation

The max flow implementation is done for every team separately. We do this by creating the source and the sink; we have two sets of variables between them: matches and rounds. We select all the matches the team ($T1$) still has to play and add them to the graph.

We then take the second set of variables, the rounds, and create a node for every round during which $T1$ is available. We then take the domains of the matches and create arcs between the match and the rounds in the domain of the match. We end with arcs from the source to each of the matches and from each of the rounds to the sink. All of the arcs have a capacity of 1. That is because we will never schedule more than one match per round, a team can play every game once and a team can only play once in a certain round.

Given the max flow instance, we just run a normal max flow algorithm over it. If we are not able to find a round for every match, this means that we will never be able to solve the schedule for that team. If that happens, we can stop the computation and backtrack. We do this for every team and if all of the teams succeed, we know that we can at least find a viable solution for every team separately and therefore possibly for the total tournament.

The max flow was implemented by checking it after each assignment of a variable. After we assign the value, we will check for every team whether we can still find a solution. If not, we can backtrack. After we checked the max flow implementation, we do not know whether this branch will yield an acceptable schedule. All we know is that we can not prove yet that all of the schedules in this branch will be unacceptable.

The max flow is quite slow. We will have to run a lot of max flows and every max flow takes quite a lot of time. So instead an alternative was implemented, we also need this alternative because we have a problem when we have to schedule it with a break. Teams can then not play multiple games in a row (depending on the length of the break). When we have breaks, the problem of deciding whether we will be able the remainder of the matches for a certain team can not be solved using a max flow. The problem to decide this is already NP-complete (dr. H.L. Bodlaender, personal communication), so we will need an alternative to make sure we can backtrack as early as possible.

6.2 An alternative to the max flow

Instead of using a max flow, an alternative was also implemented. This alternative is a check which roughly does the same as the max flow, it checks whether it is possible to schedule the remainder of the schedule for a certain team. The idea is that we check it heuristically, which will be faster, but also a little less strong. The strength we lose is so small compared to the win in time we make, that this alternative works better than a max flow solution.

The idea is that we make a distinct list of rounds a team can still be scheduled in. This distinct list is created by taking all the domains for the matches we still have to schedule for a certain team. When we combine the domains, we end up with a set of rounds a team can still be scheduled in. We then also count the number of matches we still have to schedule. When the number of matches exceeds the rounds, we can stop the search.

This can be done much faster, since we will only have to generate the information and perform the check. Combining the information is done really fast, because we already have to do some combinations when we do other checks. So the extra time it takes is really little.

We lose some expressive power when we have two matches that can only be scheduled in a certain round (or three in two rounds) and another match can still be scheduled in multiple other rounds. If this is the case, the max flow will detect it, while our alternative will not. On the bright side, this is not likely to happen when we look at our data. The domains are changed normally by assignments, which will either block a full round for every match of team, but it is unlikely that the domains will differ so much that we will miss it with our alternative.

6.3 Analysis

It is important to note how many times the max flow will be called. We will do these checks a lot, so a small improvement in speed for every run will already mean a large improvement in runtime for the total algorithm.

We solve the max flow using the general algorithm by Ford-Fulkerson [43]. This runs in $\mathcal{O}(E \cdot f)$ time, where E is the number of edges and f is the maximum flow. The latter will generally be small, that is the total number of games we will have to schedule for a team. E can be larger, but still only exists of the cardinality of the domains we still have left (and some extra for going to the source/sink). We could implement our problem using Edmonds-Karp [44], but since f is small, we do not win a lot in that way. To quote the reference [44]: "When the capacities are integral and the optimal flow value f is small, the running time of the Ford-Fulkerson method is good." [1].

The problem is therefore not in solving the max flow, but in the number of times we try to solve the max flow. If for instance we have a DRR with T teams (no backtracking), we already need $T \cdot T \cdot T - 1$ max flows. So we will already need $T^3 - T^2$ max flows without any backtracking. For an instance of T = 8, this means we already need 448 times the max flow algorithm, even though we do not backtrack at all!

[1]They use a little different format for the f, but it does not affect the idea behind the quote.

So when we have backtracking in problems, we will have to check even more max flows. As shown, the problems is therefore the number of max flows we will have to build and execute and not the running time of each max flow.

6.4 Comparing the running times

We will now show some results from our program, where we compare the running time in seconds with and without max flow and the number of max flow runs it actually tries. We also show the number of times the max flow check actually works better than the alternative.

While running the tests we first run the alternative check, so we can see how many times we would have been stopped by the max flow, which the alternative did not capture. This is shown in the table 6.1 in the column times. Column item count is the number of max flows we tried before we found the solution.

The last two have some unavailabilities set. You can see that these influence the times the max flow works better than the alternative.

Teams	Rounds	Max/round	No max flow	Max flow	Count	Times
3 groups with 6	23	4	0.37s	6.02s	1620	0
7	14	3	0.11s	1.00s	385	0
9	19	4	0.38s	3.11s	675	2
7	14	3	0.15s	1.19s	537	6

Table 6.1: Results

6.5 Leaving out the max flow

Given the results we can easily see that the max flow takes a large portion of the computation time. The longer it takes, the fewer assignments we can try. Of course it is a valid point that we have a faster algorithm when we find the blocked branch as soon as possible, but there is also a huge time difference. Even with the unavailabilities in there, which make the max flow work better, we still only use them a couple of times in favor of our heuristic solution. The score is quite bad, considering the scheduling is 10-20 times slower compared to the scheduling without max flow.

Most of the time we are able to schedule matches in multiple rounds and rounds are removed from all matches when a certain match is scheduled in that round. Therefore the list of rounds are quite alike. They will differ (especially with unavailabilities at the start), but they will stay quite close. With our alternative, we only check whether we have enough space to schedule the remaining games. Chances that a certain game can still be scheduled in many rounds and two others can only be scheduled in one round (when we would say with this check it can possibly still find a solution) are slim. So the alternative works very well on this type of data and is much faster than the max flow implementation.

So in the end we can search many more branches, while we will also look at some branches which might never yield a solution. With this technique, there is a large chance that we will find that out quickly and we will only have to search for some extra assignments, which will of course cost us. But in the long run we will benefit so much from the speed advantages, that it is worth removing the max flow and only using this check.

Chapter 7

Implementation

In this chapter we will give a short summary of implementation details. This will give the reader some insight in how things work together and why some choices were made.

7.1 Basic build

The tool itself was implemented using PHP. This is definitely not the most commonly used language in a thesis in general, but the idea behind the tool was to create a tool which worked on the web and was easy to use by people on-line. The front-end is created using HTML/CSS/Javascript/Jquery and the back-end uses Codeigniter (`www.codeigniter.com`) as a basis. This is just merely used to help us keep focus on the important things and we let the framework handle the basic things (routing to the right files etc.).

The database is simply a SQlite database, since it was the easiest to implement. The database itself is not really important in this thesis, since the focus lies on the calculation and the amount of data in the database is really small. It is just used to save the settings and to save the scores.

7.1.1 Tournaments

In this thesis we have seen different types of tournaments, which all have their specific settings. These tournaments are implemented using inheritance, a concept from object oriented programming (OOP). Depending on their settings and their specifics, functions are overloaded to help execute the correct tournament-specific code.

All tournaments have an array which indicates which modules should be shown to the user. Depending on the type of tournament, we can that way easily add a module, which is already finished, to a tournament when we deem it useful for that type of tournament. This leads to a nice extensible environment, where we can easily add settings to tournaments and the handling is all done by the (abstract) tournament superclass.

7.2 Settings

Right now the following modules are implemented:

- Basic settings: team-names, competition type and solver type
- Extended settings: maximum games per round, minimum games per round, type of tournament, breaks, number of games
- Unavailabilities: setting which teams can not play in which rounds
- Soft constraints: setting which constraint we are allowed to break, for a certain penalty

7.3 ILP implementation

In this section we will shortly emphasize some of the design choices in the program concerning the ILP implementation.

Since we have two different types of scheduling problems when we try to solve the problems using ILP, the variables we have to decide upon also differ.

7.3.1 Finding a round for a match

When we have to assign a round to a match, we have to decide whether the combination of a round and a match will be played. The variables we decide upon are in the following pattern: $X\_25\_53\_1$. The X is just to make the variables more readable, 25 and 53 would be the ids of two teams, while 1 is the round. If this variable would have a value of 1, it means 25 and 53 will play a game in round 1 and team 25 will play at home. So the formulation of each variable is of the form $X\_T1\_T2\_R$.

7.3.2 Finding an opponent for a given round

When we have to find an opponent for a given round, we will have to decide whether a certain match is played or not. The variable for this kind of ILP formulation would be X_25_53, thus X_T1_T2, since the round is already known in advance.

7.3.3 Calling CPLEX from PHP

To solve the ILP implementation, we use the tool CPLEX by IBM (`http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`). Since we want to incorporate the ILP solver in our existing program, which is written in PHP, we will need to make a call to CPLEX. CPLEX has the possibility to read from a file, then solve the given ILP and then return the result through XML.

We use the inheritance of the tournaments to generate the objective and the constraints. We will call specific functions which will generate the specific constraints we need in our tournament in order to find the right solution. We have two functions which are implemented in every type of tournament and using overloading we make sure that they return the right constraints. We then take the objective and the constraints and generate a file which holds the right format to call CPLEX to execute the code for us. The basic format can be found in Listing 7.1.

```
MINIMIZE
<objective function>

SUBJECT TO
<all constraints>

INTEGER
<all variables>

END
```

Listing 7.1: Standard format CPLEX

We then call a batch file from PHP, which we give the name of the file we need; for instance *tournament18.lp*. We copy that file to a file called *temp.lp* and then call a second batch file. That second batch file then runs CPLEX which reads the file from *temp.lp* and tries to solve the instance given in that file.

We let CPLEX then write the result to the file *temp.sol*, which we then read by the same batch which we used to call CPLEX to solve the problem. The batch file waits for CPLEX to finish and then it reads *temp.sol*. We then move *temp.sol* to *tournament18.sol* and read that file from PHP. PHP also waits for the batch file to finish computing the solution, so once PHP continues execution, we know that our solution is in the file tournament18.sol. We can then just continue with reading the XML and finding the integer values to our variables.

We then search for the variables in that file and read all of their values. The rest of the information is not interesting to us and is thus not processed. All of the values are then stored in an array and we end up with a list of games. Thus a schedule.

When CPLEX finds a solution, we know that the solution is viable. Otherwise CPLEX will throw an error and not write to a file. PHP then can not find the file and will therefore give an error message to the user ('Given the input, no solution exists.'). If we got a solution, we can continue to save the resulting array to the database, but that depends on the type of tournament. We gave CPLEX a time frame of 200 seconds, but this can just be adjusted when we call CPLEX.

7.4 CP implementation

In this section we will show some of the explicit details concerning the implementation of our CP-solver. The CP-solver differs a little from a normal CP-solver, since we implemented it specifically for this type of problem. We keep on searching until we are able to assign a value to every variable. In listing 7.2 there is (very short) pseudo-code which will show how it in basic works.

```
while (!all variables assigned) {

pick variable;
assign value;
save state;

execute propagators;

if (broken constraints) {
  backtrack;
}

}

return result;
```

Listing 7.2: short CP pseudo-code

7.4.1 Constraint formulation

We have two types of constraints: inequality constraints and set constraints. Set constraints are constraints set when usign the tool, like minimum/maximum number of games.

The inequality constraints are called that way, since one team can not play two games in the same round, otherwise we would end up with a schedule which is not viable. Because we generate the constraints ourselves, we end up with easy to manage constraints which we do not have to parse. The variables are written just as in the ILP, though the round is missing (as this is in the domain). The variable for a match between Team 1 and Team 2 for instance: X_1_2. So a constraint may be of the form X_1_2 $\neq$ X_1_3. These constraints are all just stored in objects, so we can easily work with these constraints.

We have these constraints between every set of matches, if one of the teams is equal between two matches. This way we make sure that we do not violate these constraints.

Set constraints are all handled in the CP-solver itself. It checks after assigning a value if we did not violate any of the constraints set previously. So we do not officially create constraints regarding them (like we formulate them in the ILP formulation), but instead we use the values given to calculate whether we did not violate any of the set constraints.

7.4.2 XDebug

The faster a CP implementation is able to calculate whether a certain branching is effective, the faster the algorithm will find a solution. In order to speed up the implementation, XDebug was used to make a map of how many times certain functions were called. Using this and a close code inspection, we were able to increase the number of assignments per second by 250%.

Chapter 8

Experiments

In this chapter we will show some of the results our tool delivers. We will start off with different types of tournaments and then we will experiment with the soft constraints.

8.1 No-time competitions, time-competitions and common tournaments

We took these three tournaments together, because the result we are trying to reach are quite the same. They just have a different set of settings, but we could rewrite each type of tournament in the other type. We start off with some simple tournaments, only specifying teams and rounds:

Teams	Rounds	Number of games	CP time (s)	ILP time (s)
4	6	12	0.02	0.69
8	14	56	0.16	0.74
12	35	132	1.02	0.88
16	61	240	14.56	1.53
16	65	240	3.34	1.58

Table 8.1: DRR, Maximum per round: 4

We see that the denser we try to find a schedule, the more time solving the problem using CP takes. This is of course logical, since it will have to backtrack a lot more in order to find a solution. The ILP solver on the other hand stays fast, which is nice to notice. There are a lot of (general ILP solving) speed improvements made in the ILP solver, which clearly shows in the results found. We do have a larger overhead, so CP is faster when we try to solve smaller instances.

8.2 Single round tournaments

The next type of tournament we experiment upon is the single round tournament. In this type of tournament we try to connect opponents until no opponents are left (unless the number of teams is odd).

Teams	CP time (s)	ILP time (s)
20	1.22	0.69
40	3.69	0.91
60	8.07	1.28
80	13.70	1.98
100	21.92	3.28

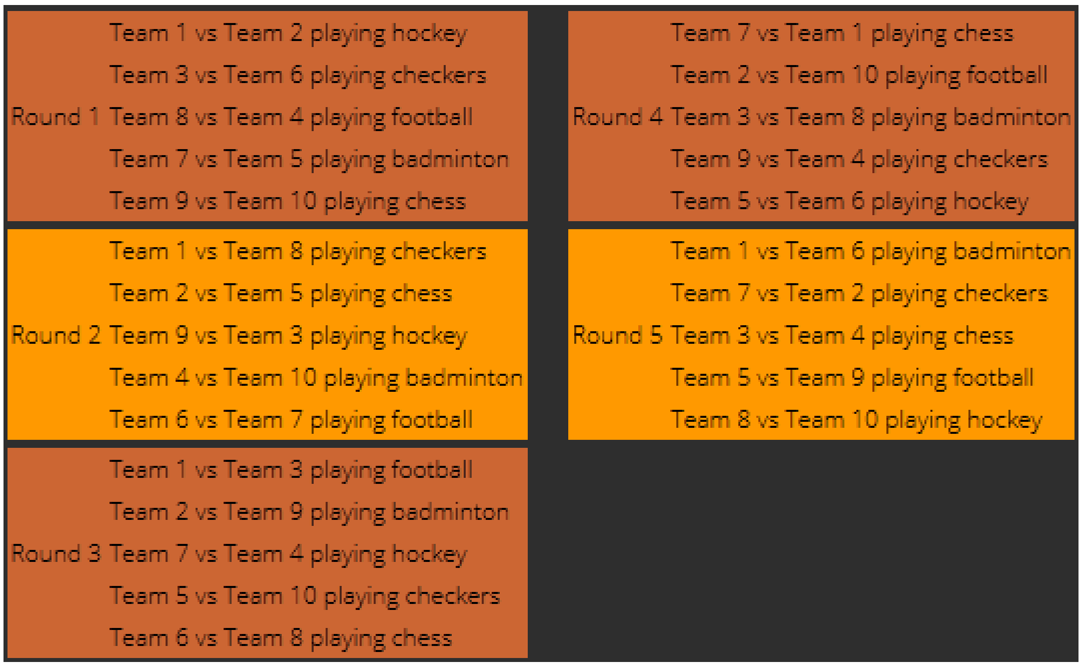
Table 8.2: Single round tournament

It is clearly shown that the ILP solver is much faster than the CP solver. Again this is probably because it is much better optimized. The results are quite good overall though, we can still find a solution for 40 teams (which in general would be a nice size for a chess tournament with 9 rounds).

As shown in the results, the ILP solver is in general quite a bit faster, but the CP solver is able to find solutions to the instances.

8.3 Multi-disciplinary tournaments

This is the last section where we test a different type of tournament. We enter a set of sports, which directly is the number of games a certain team will play. We will just use the ILP format here, since it is not that convenient to use CP for this type of decision problems; where we still have to decide which of the matches will be played. That means we will influence our result by selecting matches and that does not work well with the way we solve the problem using CP. Instead of showing the speed results (which compare to the results above) we will show the result by our tool. We have 10 teams, 5 sports and all teams will play each sport once, the result is in Figure 8.1.



	Team 1 vs Team 2 playing hockey		Team 7 vs Team 1 playing chess
	Team 3 vs Team 6 playing checkers		Team 2 vs Team 10 playing football
Round 1	Team 8 vs Team 4 playing football	Round 4	Team 3 vs Team 8 playing badminton
	Team 7 vs Team 5 playing badminton		Team 9 vs Team 4 playing checkers
	Team 9 vs Team 10 playing chess		Team 5 vs Team 6 playing hockey
	Team 1 vs Team 8 playing checkers		Team 1 vs Team 6 playing badminton
	Team 2 vs Team 5 playing chess		Team 7 vs Team 2 playing checkers
Round 2	Team 9 vs Team 3 playing hockey	Round 5	Team 3 vs Team 4 playing chess
	Team 4 vs Team 10 playing badminton		Team 5 vs Team 9 playing football
	Team 6 vs Team 7 playing football		Team 8 vs Team 10 playing hockey
	Team 1 vs Team 3 playing football		
	Team 2 vs Team 9 playing badminton		
Round 3	Team 7 vs Team 4 playing hockey		
	Team 5 vs Team 10 playing checkers		
	Team 6 vs Team 8 playing chess		

FIGURE 8.1: 10 teams, 5 sports, 5 rounds, multiple disciplinary tournament

To remark in the image: every team plays each sport once, every team plays each opponent maximum once and every sport is played maximum once per round. This is exactly what we expect from the multiple disciplinary tournament and therefor the resulting tournament is working perfectly as expected.

8.4 Soft constraints

In this chapter we will test whether the soft constraints work. We do this by selecting several cases which are on the boundary whether the soft constraint should be violated. We will then verify that the results are indeed correct.

We start with Table 8.3, where we will test whether the soft constraint concerning the rounds works. The idea behind this soft constraint is that it will give us two additional rounds if it is not able to schedule the games in the given number of rounds. When we have 8 teams, we will need the minimum of 14 rounds to create a DRR schedule. The results when we try to create a schedule with 11 to 14 rounds are shown in Table 8.3. As shown, the results are indeed correct. When we have 11 rounds, we can not find a schedule (because even with the extra 2, we will still not have enough rounds). When we have 12 or 13 rounds, we will use the soft constraint and punish our solution and when we have 14 rounds, it will just use the 14 rounds given.

Teams	Rounds	Violation	Time (s)
8	11	-	-
8	12	4 games in 13, 4 games in 14	0.63
8	13	4 games in 14	0.70
8	14	None	0.70

Table 8.3: ILP, Normal games, soft constraint: rounds, DRR, Maximum per round: 4

We continue with another test on the soft constraint concerning the rounds, in Table 8.4. We do this so we can compare the results to the results when we use the other soft constraint (the number of games, where we schedule less games). So in this case we have a set number of games, being 5, and we will check what the results are using that soft constraint.

Teams	Rounds	Violation	Time (s)
8	4	4 games in round 5	0.60
8	5	None	0.60
8	6	None	0.62

Table 8.4: ILP, Normal games, Soft constraint: rounds, number of games: 5, Maximum per round: 4

The result is that we have an extra set of games in round 5, when the number of games is too large for the given number of rounds. Since the constraint allows us to add additional rounds, this is expected behavior. When we use the other type of soft constraint, which allows us to use less games. The results of the exact same test, with only a changed soft constraint, can be found in Table 8.5.

Teams	Rounds	Violation	Time (s)
8	4	Just 4 games per team	0.57
8	5	None	0.57
8	6	None	0.60

Table 8.5: ILP, Normal games, Soft constraint: number of games, number of games: 5, Maximum per round: 4

We find that the other soft constraint, where we can make the number of games lower, reacts exactly the other way. We schedule less games, instead of using an extra round. This is the expected behavior and it works perfectly fine. It is able to find the results fast and reliable, while we do pay the penalty in the case with only 4 rounds, because we are not allowed to go past this number of rounds.

8.5 Breaks and unavailabilities

We still have breaks and unavailabilities, so we will continue with some results concerning these.

Teams	Rounds	Breaks	Time (s)
8	20	0	0.81
8	30	1	1.06
8	43	2	1.52
10	23	0	0.95
10	40	1	8.39
10	59	2	17.54

TABLE 8.6: ILP, Common tournament, Maximum per round: 4, Unavailabilities: Random

It can be seen from the Table 8.6 that breaks increase the difficulty of the problem. The rounds now influence each other and that directly means that solving the problem takes a lot more time.

Chapter 9

Conclusion

In this thesis we discussed the different things a normal (not professional) user would probably want in his tournament. How do these influence the normal tournament scheduling problem? We have different constraints, which can all be set by a user:

- Unavailabilities
- Minimum number of games
- Maximum number of games
- Different types of tournament
- Different kinds of tournaments (SRR, DRR, Amount x)

We recognize these types of tournaments, which all have their specialties:

- Competitions
  - No time-constraints
  - Time-constraints
- Common tournament
- Single-round tournament
- Multiple-disciplinary tournament

In the end we created a web-based tool, which makes it possible for users to put in their information and create a tournament they can use to schedule their games. All of their information is used in this scheduling and we return a schedule they can start to use. To solve this problem, we use two different types of solvers: CPLEX, with an ILP-approach, and PHP, with a constraint programming approach. The first one has huge expressive power and provides us the possibility to implement all the constraints we want easily and quickly. The latter one takes a lot more time, because of multiple reasons: PHP is interpreted instead of compiled, which of course means it is always slower to start with, and the solution is less optimized.

Chapter 10

Future work

There are still some things which could have been implemented, but were left out because they would take quite a lot of time to implement for only a little change in the thesis. The following is a list of these items, together with their chapters (when necessary) where more information can be found about them.

Knock-out (2.6)

We did explain how a knock-out part would and could end a scheduled tournament. This was not implemented in the end, since it is not a scheduling problem. No constraints can be set, since we do not know in advance who will be scheduled in which round of the knock-out part. Therefore the knock-out part would just be a fixed extension to the tournament and therefore no addition to the scheduling problem.

Day parts (3.4.1)

We did implement the time-constrained tournament. However, the time-constrained tournament still uses a fixed day-pattern (which will not change depending on user-input), where we would like it to be a pattern which is also created by the timestamps the user put in (if round X starts five minutes later, we can schedule more games). The pattern does not suffice when the time a match takes falls outside of the pattern. So we would like the pattern to be generated by the code instead of having it fixed.

Home-away patterns

We have created rules for the home-away patterns for the single-round tournament, but the schedules we found lack some common sense of home-away patterns with a DRR. We could start with all the home-games and then end with all the away-games. This is not a result we would like to see, so we should either fix these patterns or make sure that we do not use patterns which do not have a proper distribution over the rounds.

Clubs

We did not implement anything concerning clubs, groups of teams playing under the same name and using the same fields etcetera. It is probably a nice addition if we would have been able to set constraints for certain clubs, as well as divide our teams into clubs, so we can set for a specific club that only three games can be played at the same time at that club (at home). Right now this is not added, it would require extra input screens concerning the clubs and club-specific constraints.

More extensive tournament types (2.3.9)

We left out some tournament types, while we might want them for some more exotic tournaments. At this moment we did not implement the possibility to schedule a bipartite DRR, while we might want that for some types of tournaments by a user. So as an extension we could also implement these types of tournaments.

Bibliography

[1] Why we need competition in competitive sports. URL `http://www.essentialkids.com.au/younger-kids/kids-nutrition-and-fitness/why-we-need-competition-in-competitive-sports-20140401-35vee.html`.

[2] K. Toohey and A.J. Veal. *The Olympic Games: A social science perspective*. CABI, 2007.

[3] K. Blanchard. *The anthropology of sport: An introduction*. ABC-CLIO, 1995.

[4] A. Drexl and S. Knust. Sports league scheduling: Graph- and resource-based models. *Omega*, 35(5):465 – 471, 2007. ISSN 0305-0483.

[5] R.V. Rasmussen. Scheduling a triple round robin tournament for the best danish soccer league. *European Journal of Operational Research*, 185(2):795–810, 2008.

[6] J. Kyngas and K. Nurmi. Scheduling the finnish major ice hockey league. In *Computational Intelligence in Scheduling, 2009. CI-Sched'09. IEEE Symposium on*, pages 84–89. IEEE, 2009.

[7] D. de Werra. Geography, games and graphs. *Discrete Applied Mathematics*, 2(4): 327 – 337, 1980. ISSN 0166-218X.

[8] D. de Werra. Scheduling in sports. 59:381 – 395, 1981. ISSN 0304-0208.

[9] M.A. Trick. A schedule-then-break approach to sports timetabling. 2079:242–253, 2001. doi: 10.1007/3-540-44629-X_15.

[10] G. Nemhauser and M. Trick. Scheduling a major college basketball conference. *Oper. Res.*, 46(1):1–8, January 1998. ISSN 0030-364X. doi: 10.1287/opre.46.1.1.

[11] D. Briskorn and A. Drexl. {IP} models for round robin tournaments. *Computers & Operations Research*, 36(3):837 – 852, 2009. ISSN 0305-0548.

[12] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. Analyzing and exploiting the structure of the constraints in the ilp approach to the scheduling problem. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):456–471, 1994.

[13] F. Della Croce and D. Oliveri. Scheduling the italian football league: an ilp-based approach. *Computers & Operations Research*, 33(7):1963 – 1974, 2006. Special Issue: Operations Research in Sport.

[14] G. Durán, M.o Guajardo, J. Miranda, D. Sauré, S. Souyris, A. Weintraub, and R. Wolf. Scheduling the chilean soccer league by integer programming. *Interfaces*, 37(6):539–552, 2007.

[15] M. Henz. Constraint-based round robin tournament planning. In *ICLP*, pages 545–557. Citeseer, 1999.

[16] J. Régin. Minimization of the number of breaks in sports scheduling problems using constraint programming. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 57:115–130, 2001.

[17] Jeffrey Larson, Mikael Johansson, and Mats Carlsson. An integrated constraint programming approach to scheduling sports leagues with divisional and round-robin tournaments. 8451:144–158, 2014. doi: 10.1007/978-3-319-07046-9_11.

[18] I.H. Osman and J.P. Kelly. *Meta-heuristics: theory and applications*. Springer, 1996.

[19] S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, 1995.

[20] F. Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[21] F. Glover. Tabu search—part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[22] J. Hamiez and J. Hao. Solving the sports league scheduling problem with tabu search. In *Local Search for Planning and Scheduling*, pages 24–36. Springer, 2001.

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

[24] C.C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 179(3):775 – 787, 2007. ISSN 0377-2217.

[25] M. Wright. Timetabling county cricket fixtures using a form of tabu search. *Journal of the Operational Research Society*, pages 758–770, 1994.

[26] T. van Voorhis. Highly constrained college basketball scheduling. *The Journal of the Operational Research Society*, 53(6):pp. 603–609, 2002. ISSN 01605682.

[27] Pascal Hentenryck and Yannis Vergados. Traveling tournament scheduling: A systematic evaluation of simulated annealling. *Lecture Notes in Computer Science*, 3990:228–243, 2006.

[28] F. Della Croce, R. Tadei, and P.S. Asioli. Scheduling a round robin tennis tournamentunder courts and players availability constraints. *Annals of Operations Research*, 92(0):349–361, 1999. ISSN 0254-5330.

[29] F. Bonomo, A. Cardemil, G. Durán, J. Marenco, and D. Sabán. An application of the traveling tournament problem: The argentine volleyball league. *Interfaces*, 42 (3):245–259, 2012.

[30] T. Bartsch, A. Drexl, and S. Kröger. Scheduling the professional soccer leagues of austria and germany. *Computers & Operations Research*, 33(7):1907 – 1937, 2006. ISSN 0305-0548. Special Issue: Operations Research in Sport Special Issue: Operations Research in Sport.

[31] C.C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 179(3):775 – 787, 2007. ISSN 0377-2217.

[32] C.C. Ribeiro and S. Urrutia. Scheduling the brazilian soccer tournament with fairness and broadcast objectives. In *Practice and Theory of Automated Timetabling VI*, pages 147–157. Springer, 2007.

[33] A. Schaerf. Scheduling sport tournaments using constraint logic programming. *Constraints*, 4(1):43–65, 1999.

[34] S.L. Tanimoto, A. Itai, and M. Rodeh. Some matching problems for bipartite graphs. *Journal of the ACM (JACM)*, 25(4):517–525, 1978.

[35] D. de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19(2):151 – 162, 1985. ISSN 0377-2217.

[36] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 184–193, Oct 1975. doi: 10.1109/SFCS.1975.21.

[37] Cplex. URL `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[38] Chess stats. URL `http://www.chessgames.com/chessstats.html`.

[39] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[40] S.C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557 – 581, 1999. ISSN 0377-2217.

[41] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3): 449–467, 1965.

[42] L. Lovász and M.D. Plummer. Matching theory. *New York*, 1986.

[43] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2, page 651–659. MIT press Cambridge, 2001.

[44] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2, pages 660–663. MIT press Cambridge, 2001.