Scheduling of Sport Tournaments using Constraint Programming

Anton Spanne

Thesis for a diploma work in Computer Science, 30 ECTS credits
Department of computer Science, Faculty of Science, Lund University

Examensarbete 30 hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds Universitet

Scheduling of Sport Tournaments using Constraint Programming

Abstract

This thesis investigates whether it is possible to construct a scheduling algorithm for large sport tournaments. These can easily contain several thousand matches that need to be scheduled to a large amount of different arenas. Usually the tournaments take place during a limited time period of a few days and the entire schedule has to be constructed just a few days prior to the tournament.

The common structure of many tournaments along with some basic rules is used to divide the entire problem into several Constraint Satisfaction Problems (CSPs) that can be solved independently. The CSPs are modeled with the Constraint Programming library JaCoP and with some additional constraints and incomplete search methods that where implemented during the thesis work.

While the large part of the constructed CSPs can be solved completely within a few minutes, some of them need to use incomplete search methods. A combination of Limited Discrepancy Search and Credit Search was successfully used when necessary. By creating a variable and a value ordering based on respectively the fail-first-principle and the succeed-first-principle, the results improved dramatically. Pareto optimality and the ε-method was used to handle multiple objectives.

The complete algorithm shows promising results, and is easily extendable with extra rules and objectives. Some examples of such extensions are presented in the thesis.

Schemaläggning av Idrottsturneringar med hjälp av Constraint Programmering

Sammanfattning

Det här examensarbetet undersöker huruvida det är möjligt att konstruera en schemaläggnings-algoritm för stora idrottsturneringar. En sådan kan innehålla flera tusen matcher som behöver schemaläggas i flertalet arenor. Vanligtvis varar dem under ett fåtal dagar och hela schemat måste konstrueras bara en kort tid innan turneringen inleds.

Den struktur flertalet turneringar har gemensamt samt några grundläggande regler har använts för att dela upp hela uppgiften i ett flertal *constraint satisfaction problems* som kan lösas oberoende av varandra. Dessa är modellerade i JaCoP samt med ytterligare constraints och icke-kompletta sökmetoder som implementerades i samband med examensarbetet.

Medan den största delen av problemen kan lösas fullständigt inom ett par minuter, behöver vissa av dem använda sig av icke-kompletta sökmetoder. En kombination av *limited discrepancy search* och *credit search* användes vid behov med gott resultat. Pareto-optimalitet och ε-metoden användes för att hantera flera samtida mål i sökningarna.

Den fullständiga algoritmen visar lovande resultat, och den är dessutom lätt utbyggbar med extra regler och mål. Exempel på sådana finns presenterade i arbetet.

Contents

List of Figures

Acknowledgements

My acknowledgements go to Krzysztof Kuchcinski and Ferenc Belic at the Department of Computer Science for their help and comments during this work. I am grateful to Krzysztof for giving me the opportunity to help his students as a lab assistant during his Constraint Programming course. Teaching and explaining is a very good way to learn and it helped me develop many of the ideas used in this thesis. Krzysztof has also developed JaCoP together with Radoslaw Szymanek.

Further acknowledgements go to Emme Adbo and Stefan Evertsson for sharing their tournament expertise with me.

1

Background

This thesis originated in the issue of managing very large sport tournaments. One mayor part of this is to construct a schedule for all the matches in the tournament. As the tournaments grow large, this becomes a complex problem as both the number of matches and the number of arenas that has to be used increases.

Since the number of teams participating in the tournament is unknown until just a few days prior to the first match, the available time to construct the schedule is limited. Manually created schedules also tend to have errors that can be hard to find. Correcting these, or changing the schedule during or close to the tournament is not appreciated by the teams, as they might be traveling to it from all over the world.

This thesis will investigate whether an algorithm using Constraint Programming can be constructed to handle such a scheduling process.

1.1 Problem area and previous work

There are many types of sport tournaments with very different demands. Some of the most publicly known are the national series of different sports. Most of these games are fairly simple in the sense that there are only a small number of teams or individuals competing, which results in a limited number of events to schedule.

There exists several algorithms that can handle such scheduling problems. Some of them, such as [15] and [9], use Constraint Programming and others use other techniques [8]. It also exists a standardized test problem called "sport tournament scheduling[1]. The main differences between the problem defined there and the tournaments that this thesis will deal with, are the size and the time span. While the time span of the standardized problem is several weeks, with

[1] http://www.csplib.org/prob/prob026/

one match for every team each week, all the matches in the kind of tournaments this thesis will consider will be played in a few days. This means that it will be a completely different scheduling problem. The currently best algorithm for the standardized problem can handle 50 teams and about 1000 matches [9], while some large tournaments contain up to 1600 teams and 4500 matches[2]. As the algorithm should be fast enough to be used for real, which means running for days or even hours is out of the question, a completely fresh start is needed.

There actually exist one product, called Cup Assist, that is built to schedule this kind of tournaments[3]. It is using a technique called Mixed Integer Quadratic Programming to solve the scheduling problem. A similar approach to forest treatment is described in [17]. It is currently used on tournaments with up to 800 matches with good results. As it is hard to compare the algorithms without comparing the whole products, including user interface, Cup Assist is simply mentioned here to allow the interested reader to test and evaluate it.

1.2 Tournaments

A tournament is divided into several categories. Each category contains teams with a certain age and/or a specific gender. The purpose of the tournament (other than having fun) is to create a ranking between the teams in each category. This ranking could be more or less complete, but at least a winner should be appointed. The simplest way of doing this is with a playoff. This is a tree structure, where the teams are eliminated whenever they loose a match until only a winner remains. One such playoff would be created for each category and at the end there will be one winner from every playoff.

There are at least two important problems when playoffs are used. The first one is that most teams pay (travel, accommodation and entry fees) to be able to play in the tournament. The risk of loosing their first match would simply deter many teams from taking part in the tournament. The other problem is the validity of the final ranking. In the ideal case, the two best teams should meet in the last match, but there is no way to make sure that that will happen.

The solution to these problems is to establish a ranking between the teams by playing some matches before the playoff. The teams are guarantied a certain amount of matches, and it will be possible to create a playoff where the best teams, according to the new ranking, do not meet until the end of the playoff. The matches are created by dividing the teams into groups. Within each group all the teams play one game against every other team in that group. Currently the algorithm that will be shown in the following chapters will only consider these group matches. The construction of a scheduler for the playoffs is slightly simpler, since the ordering of the matches are more strict which reduces the search space.

[2]Gothia Cup: http://gothiacup.se/

[3]Cup Assist: http://www.cupassist.com/

Otherwise, much of the ideas can be directly transferred from scheduling the group matches.

The schedule that is created has to follow some basic rules. These are the most basal need for a scheduling algorithm . It should also be possible to extend the constructed algorithm with additional rules without too much work. The first two basic rules are

- Two matches cannot be played at the same time and place
- A team cannot play two matches at the same time (often extended to that a team cannot play two consecutive matches)

The first rule is just a basic scheduling rule, no two events can be scheduled at the same place and time. The second rule does make sure that a team is allowed to rest for at least one match between two of its matches. Other than the first two basic rules, some additional rule exist that should be taken into consideration.

- All matches from one division that are scheduled during a single day should be played in the same arena
- All the matches of a category might be restricted to a subset of all arenas
- Arenas might have varying opening hours

If a teams matches where located at different arenas during the same day, the team would have to travel between the arenas. This is not fair and it would cause a lot of disorder as some of the teams are travelling to the tournaments from far away and do not have sufficient local knowledge. There are exceptions to this rule, and they will be discussed and tested as an extension to the basic algorithm.

Since the categories are divided by age, some differences between them apply. There might for example be a mayor arena where the oldest teams should play if possible. In handball, the older teams use wax on their balls to make them easier to catch. This is not allowed in all arenas as the floors get sticky. There might also be arenas which are too small or do not have the correct lines on the floor, or perhaps goals which do not have a regulated size. Younger teams are often scheduled to such arenas, and common tournament rules might prohibit older teams from using them. This calls for the second rule, where a category can be restricted to a subset of all arenas.

Other than the rules, some measures of how good a completed schedule is has to be established. The two most basic are given here, but some additional examples of measures are given in Chapter 3.

- Minimize the total waiting time for all teams
- Minimize the required opening hours for each arena

1.3 Outline

The purpose of this thesis is to construct a basic algorithm that can schedule large sport tournaments in a reasonable amount of time. To make the algorithm usable, a reasonable amount of time is a matter of minutes, which differs from the computer science use of reasonable that can mean before the universe ends or before my thesis has to be completed.

There will also be some focus on the structure of the algorithm. Is it easy for a user to understand and follow the process and is it easy to construct a user friendly interface to it? As the thesis work only covers the actual algorithm and not the implementation of such a user interface, ideas are presented and discussed, but no finished examples are shown.

The next chapter will focus on Constraint Programming, to make it possible for the reader to understand chapter 3, where the algorithm is discussed, even if you are not a Constraint Programmer. Chapter 3 will then describe the algorithm and some extensions to it that which been implemented. At last, the reached results are discussed in chapter 4 together with some ideas for improvements and future work.

2

Constraint Programming

> "Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming; the user states the problem, the computer solves it"
>
> Eugene C. Freuder, 1997

This chapter is meant to give a brief overview of Constraint Programming and how it is used. It will only cover those areas that are deemed necessary to be able to understand the rest of this thesis. Much more information for the interested reader can be found in [14], [13] and [2]. The latter is recommended if the topic is scheduling of different kinds.

Constraint Programming approaches the problem of assigning values to a set of variables. The available values for each variable are specified in one domain for each variable. The variables are also related to constraints, where each constraint can be related to one or more variables. These constraints limit the possible combinations of values that can be assigned to their related variables. This leads to the classical definition of a Constraint Satisfaction Problem (CSP).

A CSP is a 3-tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where

$\mathcal{X}$ is a finite set of variables, $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$,
$\mathcal{D}$ is s set of the corresponding domains, $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ and
$\mathcal{C}$ is a set of constraints.

A solution s to P is found when all variables in X have had a value from their domain assigned to them such that all the constraints in C are consistent.

Example 2.1. We construct a CSP with the following variables, domains and constraints:

$\mathcal{X}$= $\{X, Y, Z\}$
$\mathcal{D}$= $\{\{0, \ldots, 2\}, \{0, \ldots, 2\}, \{0, \ldots, 2\}\}$
$\mathcal{C} = \{X > Y, Z > X - Y\}$

There are only two value combinations form the given domains that satisfy both the constraints. Hence the CSP has the following solutions:

$s_1 = \{1, 0, 2\}$
$s_2 = \{2, 1, 2\}$

2.1 Java Constraint Programming Library

There exist many different toolboxes, written in a multitude of languages, for modelling and solving constraint related problems. This thesis work does exclusively use the Java toolbox JaCoP[1] to evaluate and implement the described algorithms.

As it is written in Java, it looses some efficiency against the really fast solvers that are written in C++ and the like. At the same time it is Open-Source and very easy to extend. This makes it a good tool for testing new ideas. Since it is Open-Source, working with it also allows you to learn all the inner workings of the solvers (or even implement your own). Nothing has to be explained through magic.

The latest currently stable version of JaCoP is version 2.3. That is also the version that is used throughout this thesis work.

Example 2.2. Example 2.1 as it would look in JaCoP v2.3:

```
Store store = new Store();

// Variables:
Variable X = new Variable(store,0,2);
Variable Y = new Variable(store,0,2);
Variable Z = new Variable(store,0,2);

Variable tmp = new Variable(store,0,4);

// Constraints:
store.impose(new XltY(Y, X));
store.impose(new XplusYeqZ(Z, Y, tmp));
store.impose(new XgtY(tmp, X));
```

Note that the $Z > X - Y$ constraint is created with two constraints, $Z + Y = tmp$ and $tmp > X$. The result is no different, but JaCoP has not got any constraint implementation that can handle that relation by itself.

[1] http://www.jacop.eu

2.2 Labeling

Assigning the values to the variables is handled by a search or labeling algorithm. The simplest one would just assign values and then use the constraints to make sure that no inconsistencies exist with the current assignments. The algorithm would choose one variable at a time, assigning it a value from its domain. If a inconsistency is found, the algorithm simply tries out another value. When all values of a variable has been tried, the search backtracks one step to the previous variable.

The labeling process can be organized into a tree structure as in figure 2.1 where the CSP from example 2.1 is traversed. At each node in the tree a variable is selected. In the example the variable X is selected first. It is assigned 0 from its domain. Then, the variable Y is selected, and the consistency checks of the constraint $X > Y$ will fail for all the values in Y's domain. This is shown with crosses in the figure. Since all values in Y's domain have been tried and caused inconsistencies, the search has to backtrack one step. Back at the root, another value from X's domain is tried. The search continues in this manner until all combinations of value-assignments have been tried.

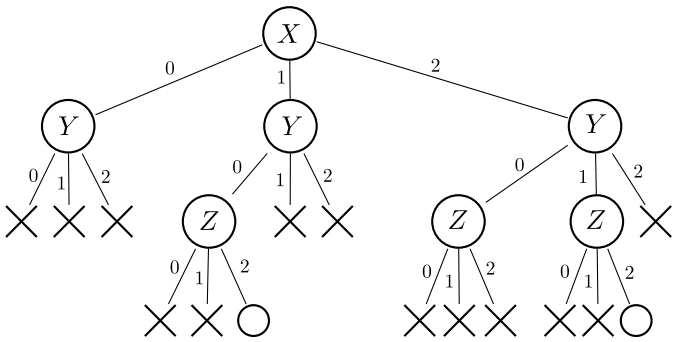


Figure 2.1: *A simple search tree traversing the variables from example 2.1.*

A solution is found when all variables have been assigned values. In the figure, this is shown with a small circle. If a single feasible solution is enough, the search stops. If all solutions should be found, the search backtracks and continues to find the remaining solutions. Most of the times though, a single solution is enough. But that solution is often required to be optimal in some sense. This can be accomplished with a extra cost-variable that should be either minimized or maximized.

When such a variable is used, every time a solution is found, the cost is recorded and the cost-variable is limited to be smaller (or larger) than the cost of the found solution. This means that the cost-variable and its related constraints can be used to prune the search tree. All solutions will not be found, but if the

search is allowed to finish and the whole search tree is explored, at least the best solution will be found.

To evaluate labeling as a method to solve CSPs, it is interesting to to discuss its computational complexity. Considering a simple CSP with n variables which all have domains with less than m values, the worst case would force the algorithm through m^n nodes to find all solutions.

Many CP decision problems are NP-complete. The task of finding a solution to such a CSP takes exponential time. However, if a potential solution is given, it only takes a polynomial amount of time to validate the correctness of it. If a decision problem that is NP-complete is reformulated into a optimization problem with a cost-variable, it will be of NP-hard time complexity. Verifying if a given solution is actually the optimal solution would in that case take exponential time.

Since it is not (currently) possible to reduce the NP-complete or NP-hard problems into more well behaved problems with nicer complexities, the course of action is now to try to reduce the domains of all variables. This will reduce the size of the search tree, even if it will still be a NP-hard or NP-complete problem.

With many variables, the exponential complexity will take its right, even with efficient pruning of the search tree. It will not be possible to examine the whole search tree since it will be extremely large. In such a situation, incomplete search methods are used.

2.3 Constraint Propagation

One way of reducing the number of nodes that has to be examined in the search tree is to apply the constraint logic before a value is assigned to the variable. Instead of using the constraints to validate nodes in the search tree, they can be used to prune the tree, removing values that are inconsistent with the current assignments. This is called constraint propagation and is widely used in in different constraint programming tools including JaCoP.

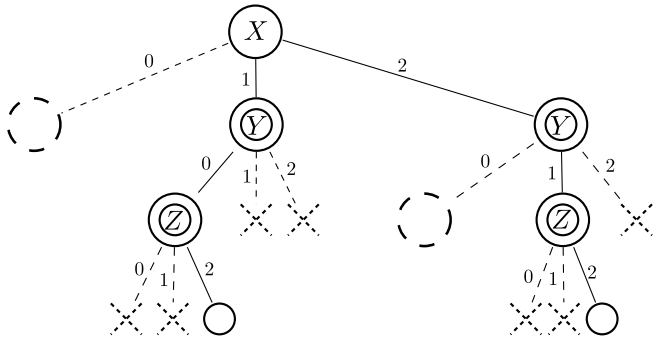


Figure 2.2: *The resulting search tree traversing example 2.1 when constraint propagation is used.*

Using propagation techniques with the CSP in example 2.1 would yield the search tree in figure 2.2. The dashed arcs and nodes in the figure are paths of the tree that can be pruned by one of the constraints and do not have to be searched. As an example, the value 0 can be removed from X's domain even before any labeling occurs. This is because $X > Y$, and Y cannot be smaller than 0. Hence $X > 0$ and therefore 0 can be removed from X's domain.

The nodes with small inner circles are nodes from where it is possible to find a solution using only constraint propagation. In the current example, the labeling algorithm only has to assign a value to X (either 1 or 2). The constraints can then deduce which values Y and Z have to have to avoid inconsistencies.

2.4 Constraints

There exist a lot of different constraints that are widely used with CPSs. The purpose of this brief survey is to give an introduction to the constraints that are used in the scheduling algorithm. Out of all the constraints shown here, only MinSquare had to be constructed for this thesis, the rest of them are already implemented in JaCoP. Some of the constraints are fairly simple and require no elaborate explanation. These constraints are shown in table 2.1. The propagation techniques they employ will not be discussed here, but the interested reader can find more information about general propagation in [14], and the propagators for specific JaCoP constraints can be found in the JaCoP source code.

Table 2.1: *Some common constraints that are used in this thesis*

Min$(\{X_1, ..., X_n\}, Min)$	$Min = \min_{i=1}^{n} X_i$
Max$(\{X_1, ..., X_n\}, Max)$	$Max = \max_{i=1}^{n} X_i$
Sum$(\{X_1, ..., X_n\}, Sum)$	$Sum = \sum_{i=1}^{n} X_i$
XplusYleqZ(X, Y, Z)	$Z > X + Y$
In$(\mathcal{D}, X)$	$X \in \mathcal{D}$
Element$(Index, \{X_1, ..., X_n\}, Value)$	$Value = X_{Index}$
Reified(C, X) $X :: \{0, 1\}$	C.consistency() $\Leftrightarrow X = 1$

The In and the Reified constraint are special in the sence that they do not only have variables as input. The In constraint restricts the supplied variable X's domain to only contain values from the given domain $\mathcal{D}$. The Reified constraint takes another constraint C as input together with a binary variable X. If C is consistent, then X has to be 1 or the Reified constraint will not be consistent. If C is not consistent, then X has to be 0 for the Reified constraint to be consistent.

Tasks

The constraints in the following sections all use the task concept. A task is something that needs to be scheduled. This could be anything from a lecture in a school to a computer operation that should be scheduled to a processor. Each task also has a resource demand which specifies the amount of resources it occupies. A general task that uses resources can be defined as the 4-tuple

$$task = \langle T, R, D, C \rangle$$

where T is its start time, R is the resource it uses, D is its duration and C is the resource demand. It can be viewed as a rectangle in a 2-dimensional space where one dimension is time and the other resources.

Cumulative

The cumulative constraint was created for scheduling problems in [1], where each task that should be scheduled uses resources. It makes sure that at any one time the total amount of resources used by all tasks running at that time does not exceed a given limit. The Cumulative constraint is used in the following way:

$$\text{Cumulative}(Tasks, Limit)$$

or as it is defined in JaCoP:

$$\text{Cumulative}(\{T_1, \ldots, T_n\}, \{D_1, \ldots, D_n\}, \{C_1, \ldots, C_n\}, Limit)$$

where n is the number of tasks, T_i is the start time for task i, D_i the duration and C_i the resource demand. $Limit$ specifies the resource limit at any given time. In the JaCoP implementation the limit has to be reached at least once in the schedule. Otherwise the cumulative constraint will not be satisfied. It can be noted that cumulative does not care about which resources the tasks use, it only cares about how much resources they use. The constraint can be formally defined as:

$$\forall t : \sum_{k:T_k \leq t \leq T_k + D_k} C_k \leq Limit$$

with the possible addition of:

$$\exists t : \sum_{k:T_k \leq t \leq T_k + D_k} C_k = Limit$$

The propagators the JaCoP implementation uses have quadratic time complexity. They are thoroughly explained in [2].

Diff2

The Diff2 constraint can be used in a similar way as the Cumulative constraint. The difference in use is that the Diff2 constraint also keeps track of which resource the task is scheduled to. It is used in the following way:

Diff2($Tasks$)

or as it is defined in JaCoP:

Diff2($\{\langle T_1, R_1, D_1, C_1\rangle, \ldots, \langle T_n, R_n, D_n, C_n\rangle\}$)

where n is the number of tasks, T_i and R_i are the starttime and the resource and D_i and C_i are the duration and the resource demand for task i. It can be formally defined as:

$$\forall i, j : i \neq j \ , T_i + D_i \leq T_j \vee T_j + D_j \leq T_i \vee R_i + C_i \leq R_j \vee R_j + C_j \leq R_i$$

Translated to text, it says that if you select two different tasks, there has to be one dimension, either time or resource, where the two tasks do not overlap. This has to hold for all combinations of tasks. The propagation technique used by the Diff2 constraint in JaCoP uses a method called Plane Sweep that is described in [6].

MinSquare

MinSquare calculates the squared sum of all resource uses for each domain value. Its purpose is to be used along with something similar to a Cumulative constraint. It is implemented for use in this thesis using the JaCoP framework. It is defined as follows,

MinSquare($Tasks, Sum$)

or as it is defined in the JaCoP extension,

$$\text{MinSquare}(\{T_1, \ldots, T_n\}, \{D_1, \ldots, D_n\}, \{C_1, \ldots, C_n\}, Sum)$$

where n is the number of tasks, T_i is the start time for task i, D_i the duration and C_i the resource demand. Sum is defined by the equation:

$$Sum = \sum_t \left(\sum_{k:T_k \leq t \leq T_k + D_k} C_k \right)^2$$

The propagation technique MinSquare uses, tries to calculate a minimal boundary on Sum's domain. This is done by first calculating the total resource demand for all fixed tasks for each time t. Then all the remaining tasks are added to the existing sums. This is done in a iterative manner where the most resource demanding task is added to the currently smallest time-sum. By ignoring the time domain of the tasks, it is ensured that no smaller solution can exist.

Currently, calculating a minimal boundary using the tasks is all that is implemented since it is the only propagation that is needed in this thesis. Other pruning would be possible, but it is hard to say if the extra time it would take to prune would pay off in increased labeling speed.

2.5 Search heuristics

Both constraint pruning and the discussed constraints are general improvements to prune the search tree. None of them use problem specific knowledge to improve the labeling process. In figure 2.3, labeling has been applied to the same problem as in figure 2.2 on page 8. The difference is that the variables are changed from X, Y, Z to Z, X, Y. It is apparent, comparing figure 2.3 to figure 2.2, that the size of the search tree is affected by the variable ordering. The time it takes to find the first solution is further more impacted by the ordering of the values in the domains of the variables. If it is enough with one solution, or if the search is incomplete, this might greatly impact the speed of the labeling process.

Several different heuristic approaches that try to take advantage of this exist. They do often require some problem specific knowledge to be chosen and used properly. A general principle that is widely used when ordering variables is the so called first-fail principle. This principle was heuristically motivated in [10] by Haralick and Elliott. The principle is further investigated in [5] where they show that the principle leads to small search trees.

The first-fail principle says that the variables that most likely are causing inconsistencies are chosen first to find the inconsistencies at a low level of the search tree. The idea is to avoid that the same inconsistencies reoccur in different branches of a large search tree. A simple example of a first-fail heuristic is to

Algorithm 1 *MinSquare pruning of Sum using the supplied tasks*

```
for i = 1 to n do
  if domainSize(T_i) = 1 then
    for j = 0 to D_j − 1 do
      sums[T_i + j] ← sums[T_i + j] + C_i
    end for
  end if
end for

for i = 1 to n do
  if domainSize(T_i) > 1 then
    min ← indexOfMin(sums)
    for j = 0 to minValue(D_j)−1 do
      sums[min + j] ← sum[min + j] + C_i
    end for
  end if
end for

forall sum in sums do
  totalSum ← totalSum + sum^2
end for

impose Sum < totalSum
```

always choose the variable with the smallest domain.

Similar to the first-fail principle for variable ordering, there exist a succeed-first principle for ordering domain values. It is discussed and tested together with the first-fail principle in [16]. If the purpose of the search is to find a optimal solution, selecting values from the domains that are most likely going to be part in such solutions is a good idea. In a scheduling problem, where the last end time should be minimized, a succeed-first heuristic would always choose the smallest value of the domain.

The selection of a suitable search heuristic is somewhat of an art, and with large problems it is hard to show that a certain strategy is better than another. In these cases this has to be deduced by extensive testing.

2.6 Incomplete search methods

Since the labeling process is a NP-complete, or when it is a optimization problem a NP-hard problem, there will always be a sharp limit to how many variables the algorithm can handle if all the possible assignments are explored. When that limit is reached, some assignments have to be left out of the search in order for

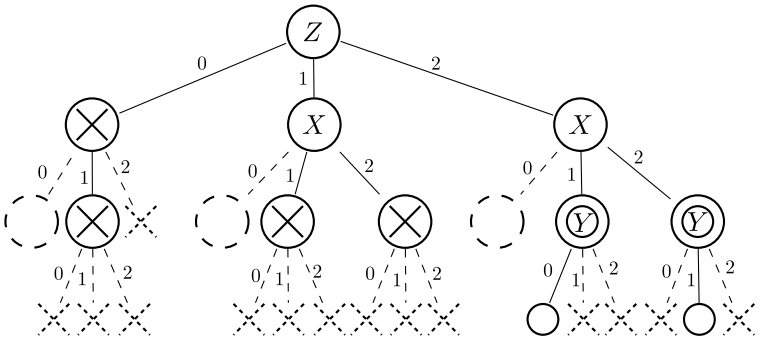


Figure 2.3: *The resulting search tree when traversing the CSP in example 2.1 with the variable ordering Z, X, Y.*

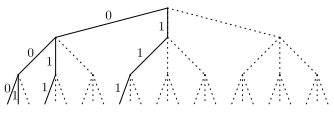
it to finish in a reasonable amount of time.

There exist several incomplete search methods that heuristically choose which branches of the search tree that should be explored. The most common ones can be found in [3], where they are described in simple and consistent way.

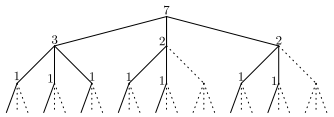
One of the earlier methods that where developed is the Limited Discrepancy Search (LDS) [11]. It simply limits the number of failed assignments that are allowed to take place in a branch. If the limit is reached, no more values are tried in that node and the search backtracks to the previous node. The number of branches LDS will explore is in the worst case

$$(d-1)\sum_{i=0}^{l}\binom{n}{i}$$

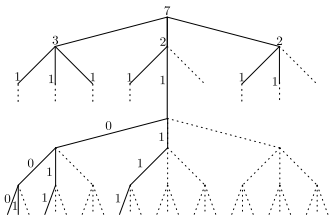
where n is the number of variables, d is the domain size of the variables and l is the discrepancy limit. With small values of l this time complexity is well behaved. An example of a LDS with one allowed discrepancy can be seen in figure 2.4(a).



(a) LDS with $l = 1$



(b) CS with 7 credits



(c) Combined LDS and CS

Figure 2.4: *Incomplete search examples. Dotted vertices indicates that the branch is not explored*

Credit Search (CS) is another incomplete search method described in [7]. It does not limit the number of discrepancies, but the number of nodes that can be visited in the search tree. It has an upper limit called credit, that controls the number of branches that are allowed in a node. At each node, the given credit

available at that node is distributed equally among the branches that originates from that node. If there are more branches than credits available, the branches without credits will not be explored. It means that when the credit drops to 1 in a branch, that brach will not allow any backtracks. Upon backtracking the search jumps back until it finds a node that has credits left. An example of a CS with 7 credits can be seen in figure 2.4(b).

The advantage of credit search is that it is very easy to control its time complexity. Since the number of nodes that are visited are controlled by the credit given, the time complexity is simply $O(c)$, where c is the credit.

In this thesis, when the variables grow to large for a complete search, a combination of CS and LDS is used. In the beginning it uses the credits as usual in CS, but when the credits drop to 1, it uses LDS to explore the current branch. The purpose of this construction is to be more thorough when examining the variables at the beginning of the search tree as the value ordering heuristics tend to be less informative there. As the search goes deeper down a branch of the tree, variables are assigned values which constrain the problem harder. This makes the value ordering heuristic more unambiguous and therefore it can be trusted to a greater extent. An example using both CS and LDS can be seen in figure 2.4(c). The figure does only show the LDS on one of the branches from the CS, but it does actually perform a LDS on all the branches in the CS tree that have credits left.

The choice of a incomplete search method is not obvious. As can be seen in [4], where the different search methods are compared, their results differ a lot depending on the problem at hand. It makes it hard to evaluate which search method that should be used using tests, since the random behaviour of the problem can influence the result more than the choice of search method.

2.7 Multiple Objectives

A important issue when optimizing solutions to a problem is that there may be many different measures that characterize a good solution. As it is described in the previous sections, a cost-variable was created and the labeling algorithm was told to either minimize or maximize it. One way of managing all the different measures would be to create such a cost-variable that takes all the different measures into consideration. It may simply add all the measures and then the labeling algorithm would optimize that sum. Each time a solution is found, the sum would be constrained like:

impose $Sum < Sum.\min$

$$Sum = \sum_{X \in \mathcal{X}_{cost}} w_X X$$

where $\mathcal{X}_{cost}$ contains all the measure variables and w_X is a optional weight that can be added to some measures that are more or less important. To use such a sum, one has to determine these weights quite accurately to make sure that each measure has a chance of affecting optimization. With a weight that is too small, the measure might drown in the noise from the other measures. With a too large weight, the measure might instead drown the others.

Sometimes two or more of the measures might be in conflict. If one of them is limited to much, the labeling algorithm might miss a solution that is extremely good according to the other measure. This is hard to control with a single aggregate cost-variable. The pruning that is done using a linear sum is shown as forbidden regions in figure 2.5(a). The angle of the edge of the regions depends on the weights in the sum. It is simple to see that with different weights, the second solution s_2 might not have been found.

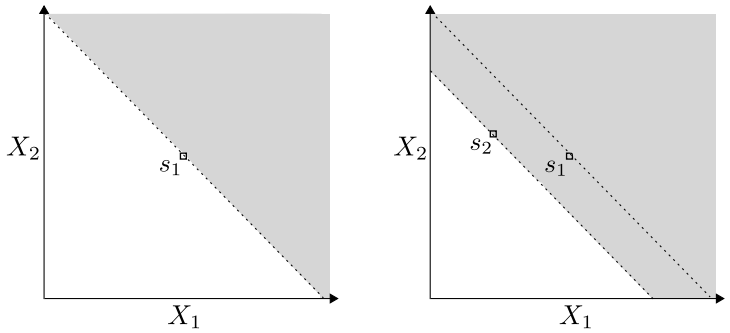
By using a concept originally from economics called Pareto optimality, it is possible to guarantee that no such solution is missed. A solution is said to be Pareto optimal if it is not dominated in the objective space by any other solution. A solution dominates another solution when all its measures are equal or better than the other solution and at least one measure is strictly better. All the optimal solutions form what is called a Pareto frontier which marks the optimal boundary of the problem. Both s_1 and s_2 in figure 2.5(b) are Pareto optimal and as such they form a Pareto frontier. When they are shown in the objective space, they are also called Pareto points. Each time a solution is found, the search space is restricted to be better than the found solution in the following way,:

impose $X_1 < X_1.\min \vee \ldots \vee X_n < X_n.\min$

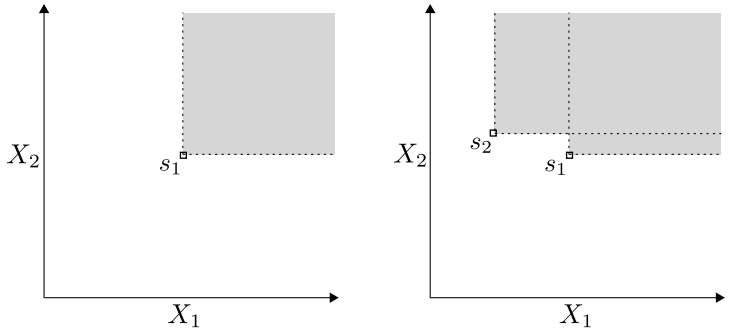
Both the weighted sum and the Pareto point solution could however cause a large performance issue. As described, when a solution is found, the cost-variable is limited to be smaller than the cost of the found solution. This can be used to efficiently prune the search tree as the cost-variable is linked to the rest of the CSP by one or more constraints.

When multiple measures are used, there will be many constraints connecting the cost-variable to the CSP. As they are connected through a sum, their propagators will work somewhat in parallel. If one of these constraints employ propagators that are not as efficient as the propagators used by the other constraints, the limit imposed on the cost-variable caused by the solution, will not be as effective as if the bad propagators where not used. It is not a case of simply removing the bad propagator or its constraint, since that would cause the measure linked to that constraint to be lost.

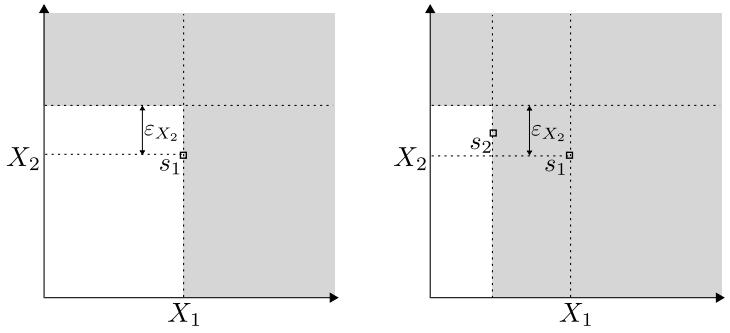
Instead of trying to optimize all the measures at once, the labeling algorithm will optimize them one by one, performing one search for each cost-variable or objective. The cost-variable that is currently being optimized in a search will as usual be limited to be smaller than the best solution found and the other



(a) Search using a linear sum.



(b) Search using Pareto points.



(c) Search using agressive Pareto points with slack ε.

Figure 2.5: *The figures show the forbidden regions (gray) with different multiple objective strategies when the solution s_1 and s_2 is found and the variables X_1 and X_2 are cost-variables.*

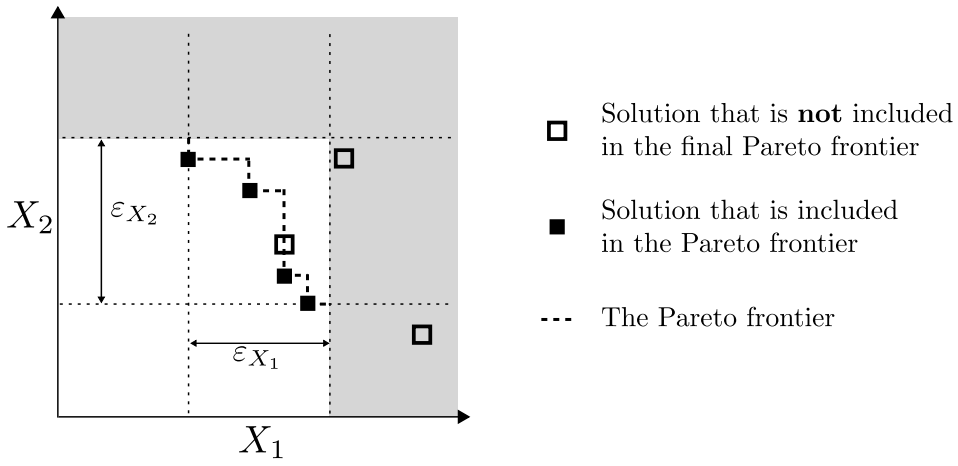


Figure 2.6: *An example of a Pareto frontier after a search has completed.*

cost-variables will be limited to be at least as good as the best value found. This is somewhat similar to the Pareto optimality concept, but the pruning is much more aggressive.

To make sure that it is not too aggressive, all objectives are also given a slack, ε, which controls how limited the variable gets when a solution is found. This is used when the variable in question is not being optimized. When a solution is found, the cost-variable is limited to be smaller or equal to the cost of the solution plus the variables slack. This is shown in figure 2.5(c), where X_1 is currently being optimized. The result of a finished search is shown in figure 2.6. Three of the found solutions in the figure are not a part of the final frontier. Two of them are not Pareto optimal and should never be a part of it, but the third is actually a Pareto point, but it lies outside the allowed slack region of X_1.

Using the aggressive Pareto concept, the ordering of the objectives are of great importance. Looking at figure 2.5(c) it is possible to see that a solution with the same X_1 cost as s_2 and a smaller X_2 cost, will be overlooked by the search because of the aggressive pruning. This is not a problem if X_1 is optimized before X_2, but the other way around, the solution will not be found. To handle this, a method called the ε-method is used [12]. It will perform the optimization of X_1 several times, increasing the upper bound of X_2 stepwise from the optimal X_2 value found until it reaches $X_2 + \varepsilon_{X_2}$.

3

The algorithm

3.1 Tournament model

The overall goal of the algorithm is to assign time and location to all of the matches in the tournament. These are represented by the variables $\{T_1, \ldots, T_n\}$ and $\{A_1, \ldots, A_n\}$. The size of the domains linked to A_1 to A_n depend on the number of arenas available as each integer the domains contain maps to one arena. All of the matches also need a integer duration, d_i, and the two teams that should be playing against each other. A match can thus be described with the following 5-tuple:

$$match_i = \langle A_i, T_i, d_i, team_j, team_k \rangle$$

where A_i and T_i are variables, d_i the matchs integer duration and $team_1$ and $team_2$ are the teams that are going to play against each other. The teams are used as markers to make it possible to identify all the matches that contain a certain team.

Example 3.1. A small tournament with only two divisions can be represented in the following way. Each division has four teams. Since all the teams should play one match against all the other teams in their respective division, each division also contains six matches.

Division 1:

$$Teams = \{team_1, team_2, team_3, team_4\}$$

$$\begin{aligned} Matches = \{\ & match_1 = \langle A_1, T_1, d_1, team_1, team_2 \rangle, \\ & match_2 = \langle A_2, T_2, d_2, team_1, team_3 \rangle, \\ & match_3 = \langle A_3, T_3, d_3, team_1, team_4 \rangle, \end{aligned}$$

$$match_4 = \langle A_4, T_4, d_4, team_2, team_3 \rangle,$$
$$match_5 = \langle A_5, T_5, d_5, team_2, team_4 \rangle,$$
$$match_6 = \langle A_6, T_6, d_6, team_3, team_4 \rangle\}$$

Division 2:

$$Teams = \{team_5, team_6, team_7, team_8\}$$

$$Matches = \{\ match_7 = \langle A_7, T_7, d_7, team_5, team_6 \rangle,$$
$$\ldots,$$
$$match_{12} = \langle A_{12}, T_{12}, d_7, team_7, team_8 \rangle\}$$

The tournament is to be scheduled in two arenas. This means the arena variables, A_1 to A_{12}, will all have the domain {1,2}. The arenas are shown in the following set where equivalence indicates which arena corresponds to which domain value.

$$Arenas = \{arena_1 \equiv 1, arena_2 \equiv 2\}$$

3.2 Generation of test data

The test data that was used during the different experiments and to validate the algorithm was extracted from a real tournament. The complete data can be found in Appendix A. The experiments were then run on random subsets of the entire data to try to vary the tests as much as possible. Some information about the tournament that was used to generate the data can be found in table 3.1.

The large issue with using data from a single tournament is whether the results are valid in general. Another way of solving this would be to create totally random tournaments and try the algorithm with them. This would make it possible to have more general results, but it might also result in that a lot of work is spent on trying to handle situations that never take place in the real world. Using random subsets of a large real tournament somewhat solves this, but it is still not possible to say that the results are really valid in general.

Table 3.1: *Information about the test data tournament*

	Amount
Teams	480
Matches	1005
Arenas	29
Days	2

3.3 Overview of the algorithm

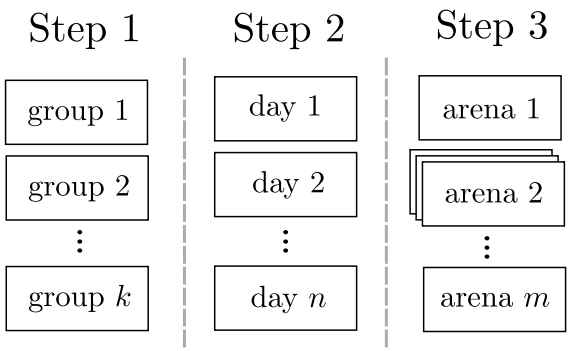


Figure 3.1: *Schematic image of the entire algorithm, divided into 3 steps.*

A useful algorithm has to take several things into consideration. Other than its speed or time complexity, the algorithm has to be transparent. A user must be able to follow the process and correct possible flaws in the presented solutions. Manual intervention is a important tool to improve the results from the algorithm. It is not likely that a fully automatic algorithm can handle all demands from different users. If it is possible for the user to make improvements during the execution, many of these problems can be resolved.

As seen in figure 3.1 the algorithm is divided into tree steps. Between these steps, it is possible for the user to modify the intermediate results. Another benefit from this design is that the algorithm is modular in the sense that there could be several ways to handle one of the steps. This also makes it possible to just run one of the steps and do the rest manually.

Other than the division into steps, the algorithm is also divided into parallel parts. These can run independently of each other. In the first step, this has little impact on the performance, since the time complexity of this step is practically linear or even constant. In the two following steps however, it reduces the complexity of the algorithm dramatically.

3.4 Step 1. Generate game templates

In this step, all the matches from all the divisions are divided into rounds. A round in this context is a set of matches that should be scheduled in the same arena and at the same day. The purpose of this is to reduce the amount of variables that needs to be labeled in step 2. This can be done since the following rule from the problem formulation exist:

- All matches from one division that are scheduled during a single day should be played in the same arena

There exists simple algorithms that spread the teams into the rounds to make an even distribution of matches for every team within each round. Even simpler is to use static templates. The construction of such algorithms or templates are not a part of this thesis work, but the following example is included to make the overall algorithm complete. Note that in the test data in Appendix A, another game template is used.

Example 3.2. Example of a static game template[1] from Svenska Handbollsförbundet[2] that are used during their youth tournaments.

3-team group:
Day 1: 1-2, 2-3, 1-3

4-team group:
Day 1: 1-2, 3-4
Day 2: 1-3, 4-2, 2-3, 1-4

5-team group:
Day 1: 1-3, 2-5, 3-4, 1-5, 4-2
Day 2: 3-5, 1-4, 2-3, 5-4, 1-2

6-team group:
Day 1: 2-5, 1-6, 3-4, 1-5, 4-2, 6-3
Day 2: 1-4, 5-6, 2-3, 4-5, 1-3, 6-2, 5-3, 4-6, 1-2

Example 3.3. Rounds created according to SHF-templates (4-team groups) using the small tournament in example 3.1:

Day 1:

$$Rounds = \{\ round_1 = \{match_1, match_6\},\\ round_2 = \{match_7, match_{12}\}\}$$

Day 2:

$$Rounds = \{\ round_3 = \{match_2, \ldots, match_5\},\\ round_4 = \{match_8, \ldots, match_{11}\}\}$$

3.5 Step 2. Assign arenas

In this step all the rounds created in step 1 get an arena assigned to them. To be able to assign arenas to rounds one variable, A_{round_i}, is created for each round.

[1] http://www.handboll.info/t2.aspx?p=1418750
[2] http://www.handboll.info

The duration of each round, d_{round_i}, is calculated as the sum of all the durations of its matches.

$$d_{round_i} = \sum_{k\,:\,match_k \in round_i} d_k$$

CSP Models

As indicated in figure 3.1, one CSP is constructed for each day. This is possible due to that step 1 has already divided all the matches between the days in question. The expected number of variables that has to be labeled in each CSP ranges between very few to a couple of hundred. With as much as a couple of hundred variables, it is not possible to expect that the labeling process can be complete. In the tests presented later on in this chapter, CS together with LDS was used. Some other incomplete search methods where also tried, but the difference in performance was not obvious. The random ordering of variables made more difference. It is clearly an area that should be examined more, but for now, the implemented combination of LDS and CS works.

The purpose of this step is to divide the rounds equally to the different arenas. The solutions also have to comply with two rules from the problem formulation:

- All the matches of a category might be restricted to a subset of all arenas
- Arenas might have varying opening hours

The first point is easily handled with the In constraint. It restricts a rounds arena variable to a subset of its original domain. As the model grows more restricted, the smaller the domains get, which reduces the complexity of the resulting CSP. This means that a restricted model is much better than a model without or with very few restrictions in terms of speed. Too many restrictions could however cause inadequate results, since the CSP might not be allowed to use all of the arenas or some of the arenas might get cluttered with rounds. In the next section an alternative and less strict way of enforcing such rules, by introducing an additional objective is presented.

The algorithm also has to take the arena opening hours into consideration. In this solution, the opening hours will not be strictly enforced by the algorithm. It will simply try to minimize the amount of rounds in each arena taking the open hours into consideration. But if it fails to comply with the opening hours, it will still continue. In the manual phase between step 2 and step 3, these failures can be reviewed, maybe resolved or step 2 could be redone with less arena restrictions.

The task concept introduced in Chapter 2 will be used to construct the CSPs. One task will be constructed for each round according to

$$task_i = \langle A_{round_i}, ?, 1, d_{round_i} \rangle$$

where the the question mark indicates that we do not care about the start time of the rounds. The order of the task parameters might look strange since the time and resource variables have switched places with each other. This is because we will use the Cumulative constraint to schedule resources (as oppose to scheduling start times) later on in this step. This makes no difference other than that it might be slightly confusing for the reader.

Opening hours will also be modelled using tasks. Since the CSP should not strictly enforce the opening hours, only those that differ between arenas will have to be considered. This is done by creating special tasks that will occupy time within arenas that have shorter opening hours.

Example 3.4. The tournament in example 3.1 has two arenas. Assume that one of them, for example $arena_1$, has shorter opening hours than $arena_2$. This means that a task

$$task_{oo} = \langle 1, ?, 1, d_{oo} \rangle$$

will be created. d_{oo} will have to be calculated to match the actual shortening of the opening hours. For example if a match takes 40 min to play, and the opening hours is 2 h shorter in $arena_1$ than in $arena_2$, d_{oo} will be $\lceil 120/40 \rceil \cdot d_{match} = 3 \cdot d_{match}$.

Objective

As mentioned, the purpose of step 2 is not to create a complete schedule, but to try to divide the rounds to the different arenas equally. No arena should have much more rounds assigned to it than any other arena. A good measurement of how well spread the rounds are between the arenas is the variable Sum from

$$\text{MinSquare}(Tasks, Sum)$$

where $Tasks$ are all the tasks, both those created from matches and those created to simulate open hours.

By minimizing Sum the rounds will be spread between the arenas. The problem with using MinSquare is its poor ability to prune inconsistent branches of the search tree as pruning is only implemented on the Sum variable. However, instead of only using Sum as a single cost-variable, it could be used in combination with another variable that is linked to a constraint with better propagators. Hence, the $Limit$ variable from

$$\text{Cumulative}(Tasks, Limit)$$

is used as a cost-variable without slack, to prune the search tree before *Sum* is optimized. This will efficiently remove many branches with a *Limit* above the smallest possible *Limit* found by the search. As mentioned earlier, *Limit* will not be the resource limit as it usually is when the Cumulative constraint is used, but instead it will be the time limit.

One of the main advantages of using CP is how easy it is to add additional constraints and in this case objectives. For example, if the tournament that should be scheduled has a large amount of rules that are more used as guidelines, it might be a good idea to add an additional objective.

There might of course be strict or hard rules that have to be followed. These should be modelled with ordinary constraints. Usually, all of the rules in a tournament are not that strict. All rounds might for example have a favourite arena they would like to be scheduled to. It is good if they are scheduled there, but if that fails it should not cause the algorithm to fail finding a solution. These kind of rules (that are often called soft rules) can be handled in another way by adding one more objective, which tries to maximize the amount of soft rules that are fullfilled. The following construction does this with the Reified and Sum constraints.

$$\forall C_i \in \mathcal{C}_{soft} \ : \ \text{Reified}(C_i, B_i) \text{ where } B_i \in \{0, 1\}$$

$$\text{Sum}(\mathcal{B}, Count)$$

where $\mathcal{B}$ contains all binary variables B_i from all the Reified constraints and $Count$ is the variable that should be maximized. The set of all soft constraints, $\mathcal{C}_{soft}$ can contain any type of constraint. In the example above, with the favourite arenas, it would be In constraints restricting the variables to a small domain that contains the rounds favourite arenas.

The labeling algorithm would in this case optimize the following three objective functions:

1. minimize *Limit* from the Cumulative constraint.
2. minimize *Sum* from the MinSquare constraint.
3. maximize *Count* from the Reified and Sum constraints.

It is not obvious that the use of both *Limit* and *Sum* does actually increase the search performance. Testing has however shown that this is the case. In figure 3.2 results from such a test are shown. The graphs show how well the search managed to optimize the *Count* variable described above. Without limit, the search hardly reached any improvements of the solutions at all. After 200 seconds it only managed to increase *Count* from 40 to 45. If *Limit* was used as a first objective, the search managed to find a solution with $Count = 45$ after only 10 seconds. After that it continued to improve until it reached 50 after 25 seconds. After 200 seconds it reached 53.

The optimal *Count* value for this test problem has not been calculated, but the results does quite clearly show that *Limit* and its Cumulative propagators does a good job pruning the search tree.

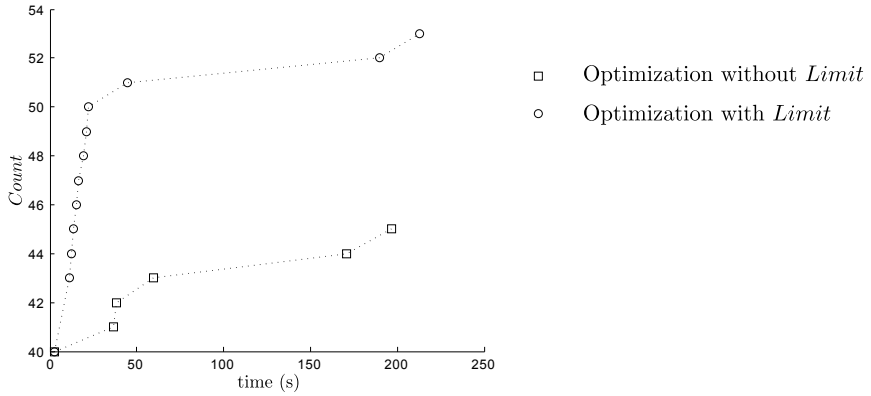


Figure 3.2: *Optimizing Count using with and without the Cumulative Limit objective*

Example 3.5. Continuing where example 3.3 left off. The two following CSPs should be created,

Day 1:

$$\begin{aligned}\mathcal{P}_1 = \langle\ &\mathcal{X}= \{A_{round_1}, A_{round_2}\},\\ &\mathcal{D}= \{\{1,2\},\{1,2\}\},\\ &\mathcal{C} = \{\text{Cumulative}(\{task_1, task_2\}, Limit),\\ &\quad \text{MinSquare}(\{task_1, task_2\}, Sum),\\ &\quad \text{In}(\mathcal{D}_1, A_{round_1}), \text{In}(\mathcal{D}_2, A_{round_2})\}\ \rangle\end{aligned}$$

Day 2:

$$\begin{aligned}\mathcal{P}_2 = \langle\ &\mathcal{X}= \{A_{round_3}, A_{round_4}\},\\ &\mathcal{D}= \{\{1,2\},\{1,2\}\},\\ &\mathcal{C} = \{\text{Cumulative}(\{task_3, task_4\}, Limit),\\ &\quad \text{MinSquare}(\{task_3, task_4\}, Sum),\\ &\quad \text{In}(\mathcal{D}_3, A_{round_3}), \text{In}(\mathcal{D}_4, A_{round_4})\}\ \rangle\end{aligned}$$

$$\begin{aligned}A_{round_1} &= A_1 = A_6\\ A_{round_2} &= A_2 = \ldots = A_5\\ A_{round_3} &= A_7 = A_{12}\\ A_{round_4} &= A_8 = \ldots = A_{11}\end{aligned}$$

Variable and value ordering

Both the value and the variable ordering turned out to be very important when labeling the CPSs. Two functions are used to order both the variables and the values in their domains. The first one is called the *Ground* of a domain value and it is defined as:

$$Ground(x) = \sum_{k\,:\,\{x\} = \text{domain}(A_{round_k})} d_{round_k}$$

It is a measurement of how many matches that are fixed to a certain arena. This can be used to make sure that the rounds get spread out by avoiding domain values with a high *Ground*.

The second one, called *Density* calculates the density of a domain value. Unlike *Ground* it also takes account of the variables that have not been fixed yet. It is defined as:

$$Density(x) = \sum_{k\,:\,x \in \text{domain}(A_{round_k})} \frac{d_{round_k}}{\text{domainSize}(A_{round_k})}$$

They were both constructed to be able to mimic the way manual scheduling is normally executed.

Example 3.6. A simple example of how *Ground* and *Density* is calculated can be found in figure 3.3. In the left part of the figure the rounds are shown as rectangles. Their width show their current domains and the height their duration. Round 3 and 4 are fixed and as such, their duration can be seen in the *Ground* measure. The *Density* measure, does also contain contributions from round 1 and round 2. They add their duration divided by the size of their domain to the *Density* of each of the values in their domains.

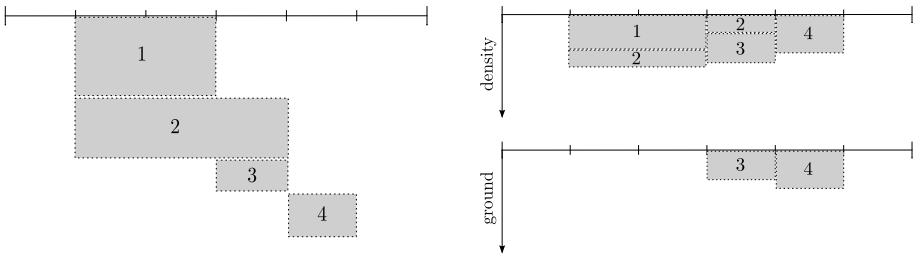


Figure 3.3: *Example showing how Density and Ground is calculated.*

The value ordering is based on the succeed-first principle. Since the idea is to minimize the opening hours of the arenas, this simple means choosing the value with the lowest *Ground*. If two or more values exist with the same *Ground*, the one with the lowest *Density* is selected.

The variable ordering was initially based on the fail-first principle with the simple minimum domain variable ordering. With that ordering, the labeling process never seemed to find the optimal solution with regards to the to the MinSquare cost-variable. By changing the minimum domain ordering into the more advanced

$$\text{order by} \left\lceil \text{domainSize}(X) \,/ \sum_{x \in \text{domain(X)}} (Density(x) - Ground(x)) \right\rceil,$$

the results got much better. The domain size is still a part of the ordering metric, but both *Ground* and *Density* have been added to improve its performance. The density part follows straight from the first-fail principle. A variable with a higher average density in its domain should be chosen first because it is more likely going to cause inconsistencies.

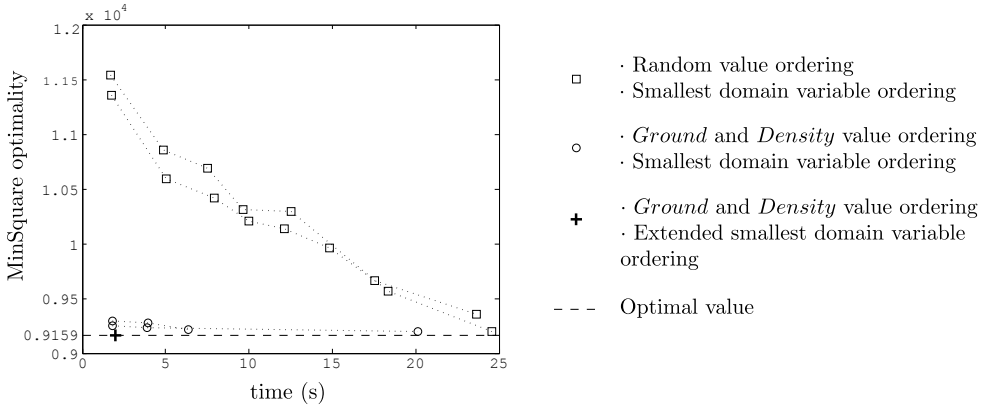
The *Ground* part is added after testing and studying live manual scheduling. Is causes the variable ordering to switch between variables with different domains, allowing all of them the opportunity to get scheduled at common domain values. This is especially important in the beginning of the search.

With the new ordering, the labeling process does not only find the best solution, it seems like it is always the first solution it finds. This means that a lot less time can be spent finding a optimal limit on the first two cost-variables. It also means that when searching for good solutions with respect to the remaining cost-variables, the search tree will get pruned efficiently by both the Cumulative and the MinSquare constraint.

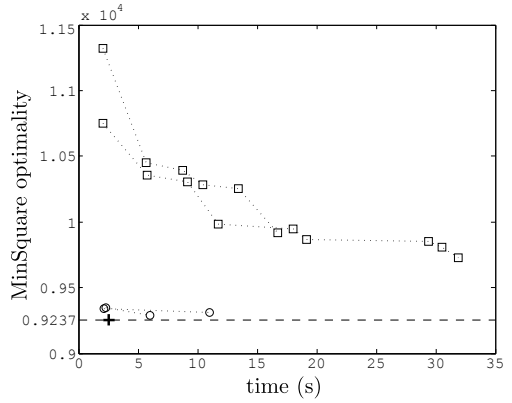
Some results using these different heuristics can be seen in figure 3.4. The squares show labeling results with a random value order and a smallest domain variable order. The circles show when the random value order has been exchanged to the one using *Density* and *Ground*. When the variable ordering is exchanged to the one using *Ground* and *Density* the optimal solution is found directly, indicated by a bold plus sign in the lower left part of the graphs. The optimal solution is also indicated by one dashed horizontal line in both the graphs.

So far, the value and variable ordering have only considered the MinSquare objective. Introducing additional objectives will most likely change how the variables and the values should be ordered. In the previous section another objective counting how many of the rounds that are scheduled within their favourite arena was introduced. Some results from optimizing that count is shown in figure 3.5.

With nothing else than the variable and value ordering described above, the first solution found have a count equal to 17. By changing the value order to take the favourite arena into consideration – favourites where chosen over non-favourites – the count was improved to 40. This small example shows that the search heuristics has to be considered if and when new objectives are added.



(a) Loose arena restrictions



(b) Tight arena restrictions

Figure 3.4: *Labeling results using different variable and value orderings. The arena restrictions are constrained with the In constraint.*

Composite Arenas

Most arenas contain a multitude of different lines on the floor to make the arena usable to as many sports as possible. Sometimes a single arena can contain lines for one handball field and two or more volleyball fields at the same time. This means that two or more volleyball matches can be scheduled at the arena at any one time, but only one handball match. If a tournament contains matches from both handball and volleyball, the scheduling model has to be able to handle this. This is also applicable to football, where teams of age 12 and less play on a smaller field (with less players) than the older teams. The large field does often contain two smaller fields that can be used by the younger teams. This can be modelled by changing the tasks in the following way:

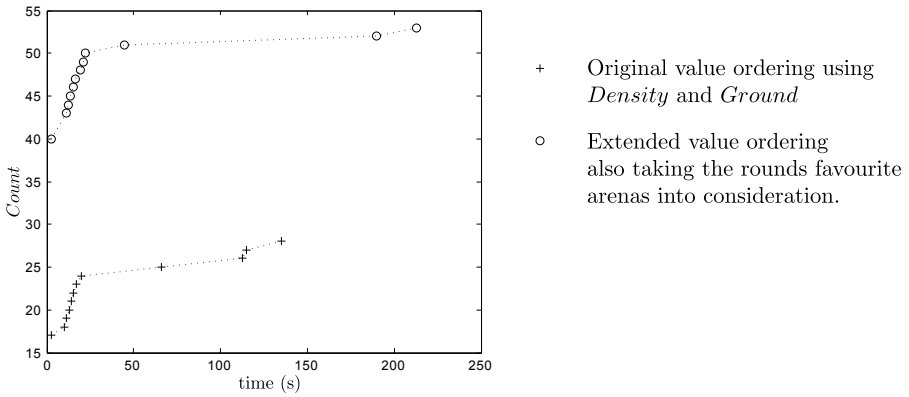


Figure 3.5: *Optimizing Count using different value orderings.*

$$task_i = \langle A_{round_i}, ?, S_i, d_{round_i} \rangle$$

where the 1 in the original representation is replaced with a S_i. Instead of forcing the rounds to just occupy one arena, they can now occupy S_i arenas. For the matches than can be played on the small fields within the larger field, S_i can still be restricted to 1. But the matches that require the larger field must occupy all the smaller fields within that field. That means that for all the fields that contain several smaller fields, S_i must be equal to the number of smaller fields. Since not all large fields contain smaller fields or contains the same amount of smaller fields, S_i will have to be different depending on in which large field the task is scheduled. This can be handled by the *Element* constraint.

Example 3.7. Assume that $arena_1$ from example 3.1 actually also contains two smaller arenas, $arena_3$ and $arena_4$. A task that can be scheduled in the two large arenas, $arena_1$ and $arena_2$, would then be defined as:

$$A_{large} :: \{arena_1 \equiv 1, arena_2 \equiv 3\}$$

$$task_{large} = \langle A_{large}, ?, S_{large}, d \rangle$$

$$\text{Element}(A_{large}, \{2, 1, 0\}, S_{large})$$

and a task that can be scheduled in the two small arenas, $arena_3$ and $arena_4$, can be defined as:

$$A_{small} :: \{arena_3 \equiv 1, arena_4 \equiv 2\}$$

$$task_{small} = \langle A_{small}, ?, 1, d \rangle$$

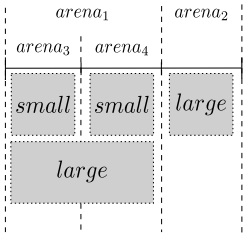


Figure 3.6: *Composite arena example*

In figure 3.6 two small and two large tasks have been placed into the arenas. The small tasks are not allowed to be placed into $arena_2$ and any large task placed within $arena_1$ will have size 2 instead of 1, as the Element constraint imposes. Note that $arena_1$ and $arena_3$ share the same domain value. It is therefore up to the model to keep track of where certain tasks are allowed to be scheduled. For example that the large task must be scheduled in $arena_1$ since it is large, but the small task must be scheduled in $arena_3$.

3.6 Step 3. Assign timeslots

In this step all the matches are assigned start times. Because of the work done in the previous steps, this is actually much simpler than it was assigning the arenas in step2, even though there are more time variables than there were arena variables. This is because the results from the two first steps can be used to split this step into more parallel parts, one for each arena and day. Each of these parts are modelled as a CSP that can be independently solved without interfering with one another. There will be a lot more CSPs in this step, but they will take much less time to solve. Unlike the CSPs in step 2 they can actually be solved completely most of the time.

CSP Models

The two main rules from the problem formulation that have to be followed are,

- Two matches cannot be played at the same time and place
- A team cannot play two consecutive matches

Both of the rules are standard scheduling rules and can be modelled with tasks and Cumulative constraints. The second rule can be somewhat generalized by enforcing a minimum amount of rest between two matches for a team. The rule above says that a team has to rest for at least one match between two of its consecutive matches. This can be generalized to r number of matches. Each match,

$$match_i = \langle A_i, T_i, d_i, team_j, team_k \rangle$$

is then modelled with three different tasks,

- $task_i = \langle T_i, ?, d_i, 1 \rangle$,
- $task_i^j = \langle T_i, ?, (r+1) \cdot d_i, 1 \rangle$,
- $task_i^k = \langle T_i, ?, (r+1) \cdot d_i, 1 \rangle$

The tasks are then used in several Cumulative constraints. The first task is used in the main Cumulative constraint for the CSP. It constrains the tasks so that no two tasks, scheduled in the same arena the same day can be scheduled at the same time. This means that one such Cumulative constraint will be created for each day and arena containing all match tasks that should be scheduled in that arena during that day.

Both the other tasks are linked to one of the teams in the match. They are used enforce that there always is at least r number of matches between each match that a team plays. This is also modelled with Cumulative constraints. One constraint is created for each team and day.

To limit the amount of possible solutions, the order of the variables is also constrained. The order that used is defined by the game template that was used in step 1, for example the game template in example 3.2 on page 22. The restriction is imposed with many

$$\text{XplusYleqZ}(T_i, d_i, T_j)$$

where T_i and T_j are the start times for two consecutive matches within one round.

The reason for limiting the schedule in this way is that the teams are not unique. It is possible to construct a new solution from a old solution by just switching two of the teams. This can also be done with an entire branch of the search tree. By only allowing one match ordering within the schedule, only one of those symmetrical branches has to be explored.

In example 3.9 on page 37, two of the CPS, $\mathcal{P}_1$ and $\mathcal{P}_3$, look very similar. The only difference between them is the indices of the tasks that should be scheduled. This means that any solution to $\mathcal{P}_1$ will also be a solution to $\mathcal{P}_3$. Solving both of them would be a waste of time. Since almost all of the divisions in a tournament uses the same game template and have the same size, this kind of symmetrical CSPs will be common. The test problem in Appendix A does almost only consist of symmetrical CSPs in step 3. The 58 original CSPs could be reduced to around 6-10 unsymmetrical CSPs, depending on what the solution from step 2 looked like when the algorithm was tested.

Example 3.8. Assuming that step 2 found a solution to the CSPs in example 3.5 that was,

$$A_{round_1} = 1$$
$$A_{round_2} = 2$$
$$A_{round_3} = 1$$
$$A_{round_4} = 1$$

the following four CSPs could be constructed for step 3,

Day 1, $arena_1$:

$$\begin{aligned}\mathcal{P}_1 = \langle\ &\mathcal{X}= \{T_1, T_6\},\\ &\mathcal{D}= \{\{1,\ldots,t_{end}\},\{1,\ldots,t_{end}\}\},\\ &\mathcal{C} = \{\text{Cumulative}(\{task_1, task_6\}, 1),\\ &\quad \text{XplusYleqZ}(T_1, d_1, T_6)\}\ \rangle\end{aligned}$$

Day 2, $arena_1$:

$$\begin{aligned}\mathcal{P}_2 = \langle\ &\mathcal{X}= \{T_2,\ldots,T_5,T_7,\ldots,T_{11}\},\\ &\mathcal{D}= \{\{1,\ldots,t_{end}\},\ldots,\{1,\ldots,t_{end}\}\},\\ &\mathcal{C} = \{\text{Cumulative}(\{task_2,\ldots,task_5,task_7,\ldots,task_{11}\}, 1),\\ &\quad \text{Cumulative}(\{task_2^1, task_3^1\}, 1),\ldots\\ &\quad \text{XplusYleqZ}(T_2, d_2, T_5),\ldots\}\ \rangle\end{aligned}$$

Day 1, $arena_2$:

$$\begin{aligned}\mathcal{P}_3 = \langle\ &\mathcal{X}= \{T_7, T_{12}\},\\ &\mathcal{D}= \{\{1,\ldots,t_{end}\},\{1,\ldots,t_{end}\}\},\\ &\mathcal{C} = \{\text{Cumulative}(\{task_7, task_{12}\}, 1),\\ &\quad \text{XplusYleqZ}(T_7, d_7, T_{12})\}\ \rangle\end{aligned}$$

Day 2, $arena_2$:

$$\begin{aligned}\mathcal{P}_4 = \langle\ &\mathcal{X}= \{\},\\ &\mathcal{D}= \{\},\\ &\mathcal{C} = \{\}\ \rangle\end{aligned}$$

It is easy to see that the solution provided by step 2 is not optimal since no round was scheduled in $arena_2$ at the second day of the tournament. This is of course not what a real solution would look like, but it illustrates how the CSPs, $\mathcal{P}_1$ to $\mathcal{P}_4$, is created. Since none of the rounds was scheduled in $arena_2$ at the second day, $\mathcal{P}_4$ is completely empty. $\mathcal{P}_3$ does on the other hand contain the start times of the matches from both $round_2$ and $round_4$.

In figure 3.7 a solution to $\mathcal{P}_2$ is shown where the different Cumulative constraints and the involved tasks are illustrated. The main constraint is shown to the left with tasks of duration d_{match} is scheduled on a single resource. Other than that, there is one Cumulative constraint for each team with tasks of duration $2 \cdot d_{match}$. That will make sure that no team will have to play two consecutive matches.

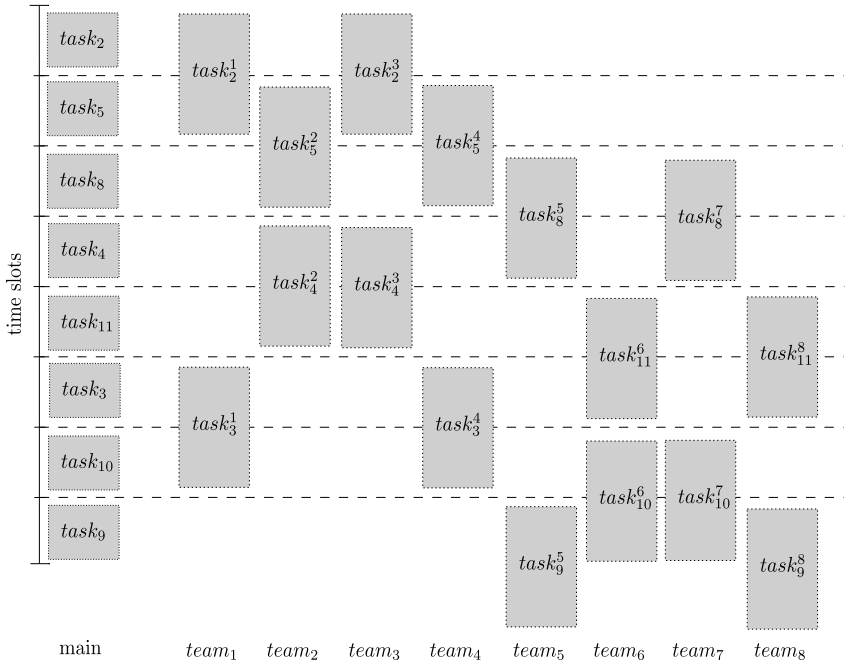


Figure 3.7: *Solution to $\mathcal{P}_2$ from example 3.9 where all the different Cumulative constraints are illustrated.*

Objective

Trying to keep the opening hours as short as possible is still one of the objectives of the algorithm. It can be done by minimizing

$$\max_j (T_j + d_j)$$

Other than that, it should also try to minimize each teams waiting time, *wait*. The waiting time can be approximated with

$$wait_i = \max_j T_j^i - \min_j T_j^i$$

where T_j^i is the start time for $task_j^i$ and $wait_i$ is the waiting time for $team_i$. It is not the exact waiting time, but minimizing it is equivalent to minimizing the actual waiting time. It is also relatively easy to construct using the Min and Max constraints.

The waiting time can itself be minimized in two ways. Either the total waiting time could be minimized or the maximal waiting time for a single team. One benefit of using the concept of multiple objectives is that it is possible to use both of them. It is motivated since we want to minimize the total waiting time, but at the same time we cannot do it if it causes a very long wait for one of the

teams. By using both objectives, it is possible to restrict the maximal waiting time, making sure that no team has to endure such a period of rest.

The problem with all three objectives is that they do not work in the same direction. If we restraint one of the to hard, it might not be possible to find a solution that is good according to the other objectives. By using slack with all the objectives this problem is avoided. The slack variables could even be chosen by the user, for example by specifying the maximum amount of allowed empty slots in the schedule. The amount of empty slots would then be used as the slack variable for the open hours objective. Using slack will cause the search to return all Pareto optimal solutions. It will then be up to the user to choose which solution that should be used. To sum up, the three following objectives are used,

1. minimize the opening hours of the arena with slack ε_{oo}
2. minimize the total waiting time for all teams with slack ε_{tot}
3. maximize the maximum waiting time for any one team with slack ε_{max}

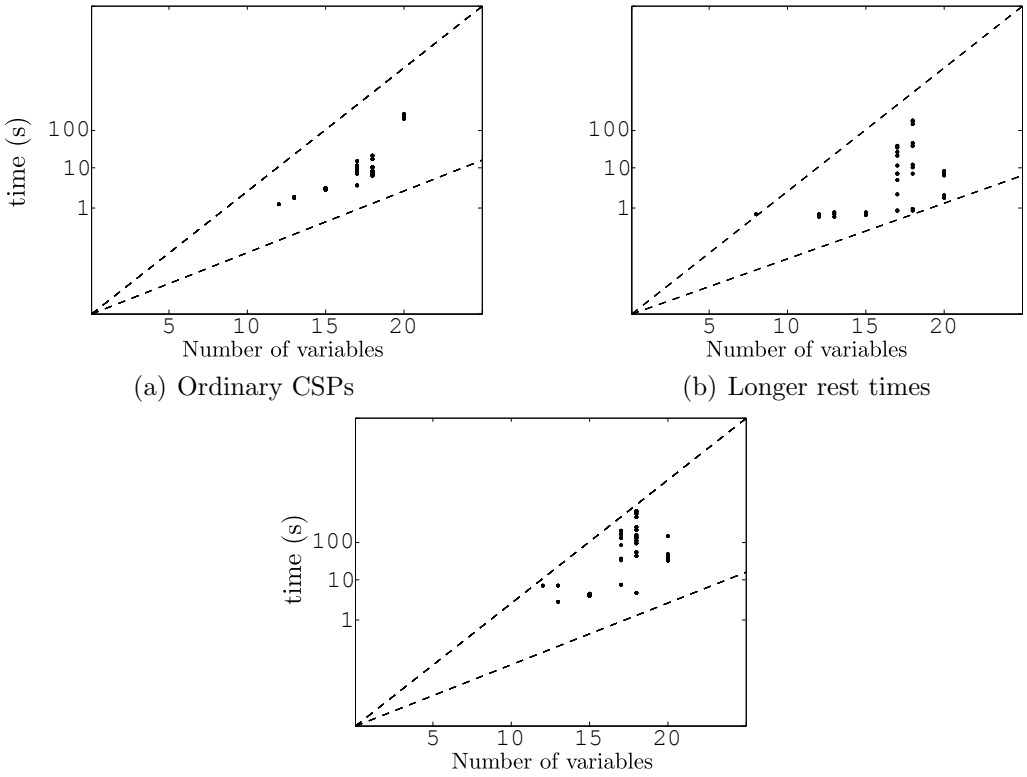
In figure 3.8(a) and (c), labeling is done on several CSPs with different sizes. In the first figure, $\varepsilon_{oo} = 1$, $\varepsilon_{tot} = 10$ and $\varepsilon_{max} = 0$. In figure (c), ε_{oo} has been changed to 3, ε_{tot} to 20 and ε_{max} to 2. The higher slack in figure 3.8(c) does not allow for as much pruning as the search in figure 3.8(a). The search tree that is explored can because of this be much larger. This results in the higher maximal search times that can be seen in the figure. At the same time, the lower bound stays within the same interval around 1 second.

Variable and Value ordering

Both the variable and the value ordering are chosen according to the basic heuristical principles that were discussed earlier.

The variables are ordered by the size of their domain – smallest domain first. It is hard to see how this could be improved in a simple way. Since the amount of variables in the CSPs is fairly small it is not such a big issue compared to the variable orderings of the CSPs in step 2. The same goes for the value ordering which simply chooses the smallest possible value from the domains to satisfy the open hours objective.

Watching figure 3.8 it is possible to see that the variable ordering is not optimal. Most of the CSPs with the same number of variables are actually totally symmetrical. The only difference is their variable ordering which is randomized before the search begins. Even though they are exactly alike, the difference in how much time they take is very large. By constructing a variable ordering that manages to order the variables in the best way, the search time would be cut dramatically. Especially when the CSPs are complex as in figure 3.8(b) and (c). As of now, that variable order has not been found, but it is as mentioned a small problem since the number of variables are not that large.



(a) Ordinary CSPs

(b) Longer rest times

(c) More slack in optimization

Figure 3.8: *Three different simulations and the amount of time it took to solve CSPs with different amount of variables. Note the logarithmic scale. The dashed lines are just a visual aid to help compare the figures.*

Related Arenas

In the problem formulation it was stated that:

- All matches from one division that are scheduled during i single day should be played in the same arena

In step 2 of the algorithm this was used to conclude that all the matches in one round had to be scheduled in one arena. In the real world, there exist locations which contain several arenas. Since these arenas lie close to each other, it would be possible to schedule a teams matches a certain day in both of the arenas. This is not possible otherwise, since you cannot expect a team to travel between different arenas during one day.

To model related arenas, some extra variables has to be introduced. For each match that should get scheduled, a extra arena variable A' is created. It will then

be used to schedule the match into one of those related arenas. Since we now have to schedule both the time and arena for these matches, the main Cumulative constraint has to be exchanged to a Diff2 constraint. As two or more matches from a round can be scheduled at the same time, the XplusYleqZ constraint has to be exchanged to a XleqY constraint. This will however reduce the amount of pruning that can be done. By also adding another constraint, XplusYleqZ, between a teams matches, some of that pruning will still be possible.

As the number of variables is at least quadrupled, the complete search method has to be replaced with a incomplete search method. The variable ordering should also be changed to take the new type of variables into consideration. One common way of doing so is to group the arena variable and the time variable for each match, and assign them together at the same time in the search.

Example 3.9. Assuming that $arena_1$ and $arena_2$ are related, the CSPs in example 3.9 will be combined into two CSPs instead of the original four. $\mathcal{P}_1$ and $\mathcal{P}_3$ would be combined into $\mathcal{P}_{1+3}$, and $\mathcal{P}_2$ and $\mathcal{P}_4$ would be combined into $\mathcal{P}_{2+4}$. Since $\mathcal{P}_4$ was empty already from the beginning $\mathcal{P}_{1+3}$ is the most interesting one. Four new arena variables, A'_1, A'_6, A'_7 and A'_{12} has to be created for $\mathcal{P}_{1+3}$. The result would look like:

$$
\begin{aligned}
task'_1 &= \langle T_1, A'_1, d_1, 1 \rangle \\
task'_6 &= \langle T_6, A'_6, d_6, 1 \rangle \\
task'_7 &= \langle T_7, A'_7, d_7, 1 \rangle \\
task'_{12} &= \langle T_{12}, A'_{12}, d_{12}, 1 \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}_{1+3} = \langle\ & \mathcal{X} = \{T_1, T_6, T_7, T_{12}, A'_1, A'_6, A'_7, A'_{12}\}, \\
& \mathcal{D} = \{\{1, \ldots, t_{end}\}\text{x4}, \{1, 2\}\text{x4}, \} \\
& \mathcal{C} = \{\text{Diff2}(task'_1, task'_6, task'_7, task'_{12}), \ldots\}\ \rangle
\end{aligned}
$$

4

Discussion

The most prominent feature of the constructed algorithm is how it is divided into several small CSPs. It has lead to two mayor gains. First of all, the smaller CSPs does not contain that many variables. This means that they can be run during only a few minutes with good results. The total amount of variables in a real case can easily reach a thousand or even several thousand. Finding a somewhat optimal solution to a CSP with that many variables would be almost impossible. It would require a perfect value and variable ordering to avoid any branching of the search tree.

One may also qestion whether an algorithm that managed this – that can solve everything within one large CSP – would be better than the current solution anyway. The big problem with a large, do-it-all, one step algorithm is to construct a user interface that is easy to use. Before the algorithm can be run, the user has to feed the algorithm with all the necessary settings. This can be a real problem for an user that is new to the system and it is likely that many of the features of the algorithm will remain unused. Additionally, if the algorithm does not find a solution that the user finds sufficiently good, the whole process has to be restarted and the user has to figure out which settings it was that caused the problems.

The second large gain from having multiple CSPs is that many of these problems can be avoided. The user only needs to feed the currently necessary settings to the algorithm. It is furthermore even possible to run the different parts with different settings. If one of the CSPs does not produce a sufficiently good solution, it is possible to let that part run a little longer using the current Pareto frontier to prune the search tree.

4.1 About the results

Since the algorithm that was presented in the previous chapter is divided into three steps, their results will be discussed independently. As the first step does not use Constraint Programming but is more of a tournament management problem, it is left out of this discussion.

Step 2

The second step was the most demanding as its CSPs are handling a large amount of variables. A lot of work was spent trying different variable and value orderings to improve the labeling speed. It turned out that the standard first-fail variable orderings did not make the CSPs solvable in reasonable time. Even after test runs that lasted over night for tens of hours, the soultions found where not even close to the optimal solutions. Visualizing the found solution, it was easy to find errors in it and an algorithm that produces obvious errors to the user is hardly any good.

Using the idea of the first-fail-principle, but taking it one step further and adapting the heuristic to the problem at hand, the domain value metrics *Ground* and *Density* were introduced. With them, it was possible to construct both a variable ordering and a domain value ordering that actually found the optimal solution (of the basic algorithm) in just a few seconds.

From this success, as the basic construction seemed to work, two extensions were also implemented to evaluate how hard it would be. Both a structural extension handling composite arenas and a objective extension where rounds where scheduled to their favourite arenas were constructed and tested successfully. Because Constraint Programming was used as a platform, the extensions were easy to model.

Step 3

The third step has the advantage that the CSPs are fairly small in their number of variables. This means that the labeling actually could be complete unlike in step 2. With the used test data, the largest number of variables that were used in one CSP was 22. With that amount of variables, the search completed within a couple of minutes most of the time.

In figure 3.8 on page 36, the exponential worst time complexity can be seen with some different CSPs. It is also possible to see that sometimes the CSPs can be solved in a lot less time than the worst case time. This is most visible in figure (b) and (c). The difference in running time does actually not depend on that the CSPs are different, since almost all of the CSPs with the same number of variables will have the same number of rounds and size of the rounds and so on. It is actually the initial variable ordering from when the CSPs are created that

make all the difference. If that variable ordering could somehow be calculated and the variables ordered accordingly, the labeling speed would be increased dramatically.

An alternative to this would be to restart the labeling with a new random variable ordering when a time limit it exceeded, hoping that the new variable ordering was better than the last. The new search could of course use all the results that the last search used, to make sure it was not a total waist of time.

4.2 Future work and improvements

The most important work that remains is probably the Graphical User Interface. The algorithm can be used as it currently is, but it will not make much sence without a user interface where the users can make necessary settings and supervise the execution.

The structure of the algorithm allows for a user interface where the user is involved, being able to correct and change tings as the process goes along. The most obvious way to take advantage of the algorithms step structure is to allow the user to modify the intermediary results between the steps. It would also be possible to create an interface where the users can choose to solve individual CSPs. For example scheduling a single arenas matches. When incomplete search methods are used, the user could also decide to refine the search, letting a single CSP work some more to improve its solutions. The possibilities are endless and they will not all be discussed here, but the shown ideas show the important modular feature of the algorithm.

Other than the interface, some loose ends exist within the algorithm. The biggest part is to construct and find additional test data and run more tests to find if and where there might be bottlenecks in the algorithm. Since the optimization problem most likely is NP-hard, the time it takes to find optimal solutions can suddenly sky rocket when the amount of variables grow. With the additional test data, the incomplete search methods that are used should be evaluated, to find optimal values for limits, credits and perhaps time-outs.

4.3 Conclusion

The purpose of this thesis was to investigate whether it was feasible to construct a basic algorithm for scheduling Sports Tournaments using Constraint Programming. The results and the algorithms presented in the previous chapters, show such a basic algorithm that works well with the test data that was used. In that sense, the algorithm that have been presented can be considered to be a success.

Other than that, the algorithm also proved to be transparent and modular, which will make it easier to construct a good graphical user interface. Much work is still left undone. Both with improving the basic algorithm, creating extensions

and of course the graphical user interface, which from the finished product point of view is almost the most important part.

Bibliography

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Springer, 2001.

[3] R. Barták. Incomplete depth-first search techniques: a short survey. In *Proceedings of the 6 th Workshop on Constraint Programming for Decision and Control, Ed. Figwer J*, pages 7–14, 2004.

[4] Roman Bartk and Hana Rudov. Limited assignments: A new cutoff strategy for incomplete depth-first search. In *In Proceedings of the 2005 ACM Symposium on Applied Computing. ACM*, pages 388–392. ACM, 2005.

[5] J.C. Beck, P. Prosser, and R.J. Wallace. Trying again to fail-first. *Recent Advances in Constraints. Papers from the*, pages 41–55, 2004.

[6] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In *Seventh International Conference on Principles and Practice of Constraint Programming, LNCS 2239*, pages 377–391. Springer, 2001.

[7] E. Boutteau, P. Chan, and D. Rivreau. Partial Search Strategy in CHIP. In *2nd International COnference on Metaheuristics*, 1997.

[8] D. Briskorn and A. Drexl. A branch-and-price algorithm for scheduling sport leagues. *The Journal of the Operational Research Society*, 60:84–93, 2009.

[9] J.P. Hamiez and J.K. Hao. Using solution properties within an enumerative search to solve a sports league scheduling problem. *The Journal of the Operational Research Society*, 156:1683–1693, 2008.

[10] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[11] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 607–615. Lawrence Erlbaum Associates LTD, 1995.

[12] C.L. Hwang and M. Abu Syed Md. Author Multiple objective decision making, methods and applications: a state-of-the-art survey. *Berlin: Springer-Verlag, 1979.*, 1979.

[13] K. Marriott and P.J. Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

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

[15] A. Schaerf. Scheduling sport tournaments using constraint logic programming. *Constraints*, 4:43–65, 1997.

[16] Barbara M. Smith. Succeed-first or fail-first: A case study in variable and value ordering, 1996.

[17] M. Stolevik, G. Hasle, and O. Kloster. Solving the Long- Term Forest Treatment Scheduling Problem. *Geometric Modelling, Numerical Simulation, and Optimization Applied Mathematics at SINTEF*, 2007.

A

Table A.1: *The gametemplates used with the test data*

	Size 5	Size 6
Day 1	1-3, 2-5, 3-4, 5-1, 4-2	1-5, 2-4, 6-3, 2-1, 3-5, 4-6, 3-2, 4-1
Day 2	3-5, 4-1, 2-3, 5-4, 1-2	6-5, 4-3, 5-2, 1-6, 5-4, 1-3, 6-2

Table A.2: *The basic test data that was used in the evaluations of the algorithm*

Divisions	*Hard restrictions day 1* *Hard restrictions day 2*	*Favourite arena(s) day 1* *Favourite arena(s) day 2*
1. 4x5, 2x6	{1..7, 9..13, 16, 20..23, 26, 27}	{1..3}
	{1..5, 7, 8, 9, 12..17, 21, 23, 25, 26}	{1..5}
2. 4x5		{5, 6, 16}
		{8}
3. 1x5, 2x6		{7}
		{9}
4. 8x5, 1x6		{1, 9, 11, 12, 13, 23}
		{3..5, 14, 15, 23}
5. 2x5, 1x6		{6}
		{16}
6. 7x5, 1x6		{2, 4, 20, 21}
		{2, 3, 7, 13}
7. 4x5		{22}
		{21}
8. 1x5, 2x6		{4, 16, 23}
		{1, 12, 17}
9. 8x5, 1x6		{9, 10, 26, 27}
		{12, 13, 23, 25, 26}
10. 4x5	{1, 2, 4, 6, 8..10, 15..19, 23..25, 28, 29}	{2, 4, 10}
	{1..3, 5..7, 10..13, 17..20, 22..24, 27..29}	{2, 3, 6, 7, 13}
11. 3x5		{6}
		{10}
12. 6x5		{8..10}
		{11..13, 23}
13. 5x5		{4, 15..17, 23}
		{1, 12, 17, 18}
14. 4x5		{7, 18}
		{19}
15. 3x5, 1x6		{6, 16, 19}
		{10, 20}
16. 7x5		{1, 2, 17, 24, 25, 28}
		{1..3, 5, 24, 27, 28}
17. 6x5, 1x6		{1, 9, 17, 23..25}
		{3, 5, 22..24}
18. 3x6		{29}
		{29}

B

Word list

Cost-variable *see page 7 and 15*

CP – Constraint Programming

CS – Credit Search *see page 13*

CSP – Constraint Satisfaction Problem *see page 5*

Cumulative *see page 10*

Diff2 *see page 11*

Domain *see page 5*

Game template *see page 21*

JaCoP – Java Constraint Programming Library *see page 6*

Labeling *see page 7*

LDS – Limited Discrepancy Search *see page 13*

Pareto frontier *see page 15*

Pareto optimal *see page 15*

Propagator *see page 8*

Value *see page 5*

Variable *see page 5*