UNIVERSITÄT BASEL

Solving the Traveling Tournament Problem with Heuristic Search

Bachelor's thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
Artificial Intelligence
ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Gabriele Röger

Patrik Dürrenberger
patrik.duerrenberger@stud.unibas.ch

February 9th, 2015

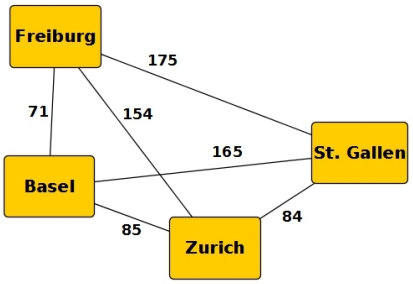
Abstract

This thesis discusses the Traveling Tournament Problem and how it can be solved with heuristic search. The Traveling Tournament problem is a sports scheduling problem where one tries to find a schedule for a league that meets certain constraints while minimizing the overall distance traveled by the teams in this league. It is hard to solve for leagues with many teams involved since its complexity grows exponentially in the number of teams. The largest instances solved up to date, are instances with leagues of up to 10 teams. Previous related work has shown that it is a reasonable approach to solve the Traveling Tournament Problem with an IDA*-based tree search. In this thesis I implemented such a search and extended it with several enhancements to examine whether they improve performance of the search. The heuristic that I used in my implementation is the Independent Lower Bound heuristic. It tries to find lower bounds to the traveling costs of each team in the considered league. With my implementation I was able to solve problem instances with up to 8 teams. The results of my evaluation have mostly been consistent with the expected impact of the implemented enhancements on the overall performance.

Table of Contents

1 Introduction

The Traveling Tournament Problem (TTP) is a sports scheduling problem where one tries to find a schedule for a league that minimizes the overall traveling distance of all teams in this league. This schedule has to have the form of a double round-robin tournament, meaning that every pair of teams plays each other twice with each of the two teams having a home game at one of these two games. The TTP is meant to be applied to leagues where the distances between teams are high and where therefore it is common to have a series of away games during which the traveling team is not returning to its home town. If these away trips are chosen well, traveling distances can be reduced. In Figure 1.1 a graphical representation of an instance of the TTP with 4 teams is given. An optimal schedule to this instance is shown to the right of the graph. The TTP, although relatively simple to define, is a hard problem to solve. This is mostly because its complexity grows exponentially with the number of teams in the league. Moreover in addition to normal constraints of a schedule, there are two additional constraints: The *at most constraint* that limits the number of consecutive away games a team can have and the *no repeat constraint* that prohibits that the same match-up, with different home teams, can be played on two consecutive matchdays. So, since Basel plays against Freiburg at home on matchday 3 it is prohibited for Basel to play in Freiburg on matchday 4.



Matchday 1	Zurich	-	Basel
	St.Gallen	-	Freiburg
Matchday 2	Zurich	-	Freiburg
	St.Gallen	-	Basel
Matchday 3	Basel	-	Freiburg
	Zurich	-	St.Gallen
Matchday 4	Basel	-	Zurich
	Freiburg	-	St.Gallen
Matchday 5	Basel	-	St.Gallen
	Freiburg	-	Zurich
Matchday 6	Freiburg	-	Basel
	St.Gallen	-	Zurich

Figure 1.1: A simple instance of the traveling tournament problem with distances between teams labeled on the edges and an optimal full valid schedule for this problem

Uthus et al. (2011) have developed the algorithm TIDA* that is capable of solving TTP instances with up to 10 teams optimally. TIDA* is an IDA*-based algorithm that has some

TTP-specific extensions. The goal of this thesis was to reimplement most of the approaches used in TIDA* to solve the TTP in an efficient manner. In a first phase of the thesis I concentrated on reimplementig the standard IDA* algorithm, followed by expanding it with the concepts of forced deepening, elite paths and subtree forests (Chapter 3). In a second phase, I also reimplemented a majority of the enhancements used by Uthus et al. (2011) to improve performance of my basic implementation. Enhancements I implemented are team reordering, symmetry breaking, team cache and multithreading (Chapter 5). In a third phase, all implemented extensions and enhancements have been examined in experiments (Chapter 6). The experiments consisted of a multitude of runs, where each run has had different settings regarding the extensions and enhancements. Thereby, I evaluated the influence of each extension and enhancement on the performance of my implementation.

Related work

My thesis, as mentioned above, has been strongly influenced by the work of Uthus et al. (2011) in which the Traveling Tournament Problem is solved with a specialized IDA* algorithm called TIDA*. Problem instances with 10 teams have been solved for the first time with TIDA*. In addition their work introduced the problem set GALAXY, which I used to test and evaluate my implementation.

Prior work of Easton et al. (2001) introduced the Traveling Tournament Problem and gave first problem classes and approaches to solve the problem. Easton et al. (2003) applied combined integer and constraint programming to the problem. As an approach to improve problem-solving methods, Irnich (2008, 2010) first described a compact formulation of the problem followed by a solving strategy based on a Branch-and-Price algorithm. Uthus et al. (2009) applied DFS* to the TTP, keeping heuristic estimates in memory since they are expensive to calculate.

Concepts used in Uthus et al. (2011) and my thesis build upon various works on subtrees (Uthus et al., 2009), parallelization (Hafidi et al., 1995; Rao et al., 1987; Powley and Korf, 1991), node ordering (Powley and Korf, 1991), disjoint pattern databases (Korf and Felner, 2002) and symmetry breaking (Irnich, 2008).

2
Background

2.1 Traveling Tournament Problem

In an instance of the *Traveling Tournament Problem* (TTP) there is an even number of teams n which have to play against each other in a double round robin tournament. This means each team has to play against each other team twice, once in a home and once in an away game. The games will be played on $2 \cdot (n-1)$ matchdays. On a matchday every team plays exactly one game, which results in $\frac{n}{2}$ games per matchday. So over the whole league there will be a total of $n \cdot (n-1)$ matches played.

The goal is to schedule these matches in a way that minimizes the overall traveling distance. The distances between teams are given as a symmetric matrix.

There are two more constraints that further complicate the problem. One is called the *at most constraint* (AMC), the other is the *no repeats constraint* (NRC) (Uthus et al., 2011). The AMC demands that the number of consecutive home or away games is at most a given parameter b. The NRC demands that a pair of teams that has played against each other at a certain matchday can not play against each other again the following matchday.

With these informations given, a TTP instance can formally be defined as follows:

Definition A *TTP instance* is given by a triple $\mathcal{I} = \langle T, \mathbf{D}, b \rangle$, where

- $T = \langle t_1, ..., t_n \rangle$ is a sequence of team names and $n \geq 4$ is an even integer.
- $\mathbf{D}$ is a symmetric $n \times n$-matrix of non-negative real values, where $\mathbf{D}_{i,j}$ denotes the traveling distance from the venue of team t_i to the venue of team t_j. Therefore, $\mathbf{D}_{i,i} = 0$ for all i.
- $b \in \mathbb{N}^+$ is a bound on the maximum number of consecutive home or away games.

Before we can define, what is a *solution* for TTP, we need to formally introduce the notion of a *schedule*:

Definition A (partial) *schedule* for a TTP instance $\mathcal{I} = \langle T, \mathbf{D}, b \rangle$ is a sequence $S = \langle D_1, ..., D_m \rangle$ of *matchdays* with $m \leq 2 \cdot (|T| - 1)$. Each matchday D_i is a set of *matches* $M = \langle t_h, t_a \rangle$, where $t_h, t_a \in T$ are the home and away team, respectively ($t_h \neq t_a$). Each

team may occur at most once in each matchday. For $1 \leq i < m$, it must hold that $|D_i| = \frac{|T|}{2}$, i. e. all teams are already scheduled for this matchday. The last matchday m can be partial, i.e. $1 \leq |D_m| \leq \frac{|T|}{2}$.

The *cost* of a (partial) schedule S is the sum of the traveled distances of all teams. To calculate the sum of traveled distances for a team an auxiliary function L is required. If team t_i plays at home on matchday D, it plays at location i. If it has an away game against t_j, it plays at location j. We denote this location with $L(t_i, D)$. To reach the location for matchday D_j, team t_i has travel cost

$$cost_{t_i}^{j}(S) = \begin{cases} \mathbf{D}_{i,L(t_i,D_j)} & \text{if } j = 1 \\ \mathbf{D}_{L(t_i,D_{j-1}),L(t_i,D_j)} & \text{otherwise} \end{cases}$$

We also need to account for the cost to travel home at the end of the tournament. This is only necessary after all matchdays ($m = 2 \cdot (|T| - 1)$) in the tournament have been scheduled completely ($|D_m| = \frac{|T|}{2}$). We define

$$cost_{t_i}^{\text{back}}(S) = \begin{cases} 0 & \text{if } m < 2 \cdot (|T| - 1) \text{ or } |D_m| < \frac{|T|}{2} \\ \mathbf{D}_{L(t_i,D_m),i} & \text{otherwise} \end{cases}$$

The total cost of the schedule is then

$$cost(S) = \sum_{t \in T}((\sum_{j=1}^{m} cost_t^j(S)) + cost_t^{\text{back}}(S)).$$

A (partial) schedule is *valid* if

- for all i, $\mathcal{L} = <L(t_i, D_1), L(t_i, D_2), ..., L(t_i, D_m)>$ has no subsequence $<s_1, s_2, ..., s_{b+1}>$ with $s_k = i$ for all $1 \leq k \leq b + 1$ or $s_k \neq i$ for all $1 \leq k \leq b + 1$ (AMC), and
- no combination of matches exists, where $<t, t'> \in D_j$ and $<t', t> \in D_{j+1}$ for $1 \leq j < m$ (NRC).

Now a *solution* can be defined as follows:

Definition For a TTP instance $\mathcal{I} = <T, \mathbf{D}, b>$, a *solution* is a full valid schedule, i.e. a valid schedule $S = <D_1, ..., D_m>$ with $m = 2 \cdot (|T| - 1)$ and $|D_j| = \frac{|T|}{2}$ for all $1 \leq j \leq m$. In addition a full valid schedule must hold that for all pairs of teams $t, t' \in T$ with $t \neq t'$ exists a pair of matchdays D_i, D_j with $i \neq j$ such that $<t, t'> \in D_i$ and $<t', t> \in D_j$. An *optimal* solution is a solution with minimal total cost.

2.2 Search space

I will now characterize the search space of a TTP instance. It can be represented as a search tree. The root node of this search tree corresponds to the empty schedule. All successors of a node n with schedule $S = <D_1, ..., D_m>$ and last matchday $D_m = \{M_1, ..., M_k\}$ extend the schedule with one match, where only the last matchday in the schedule may be partially assigned ($|D_i| = \frac{|T|}{2}$ for all $1 \leq i < m$). Therefore, if $k < \frac{|T|}{2}$ a schedule of a

successor node has the form $S' = <D_1, ..., D_{m-1}, \{M_1, ...M_k, M'\}>$. If $k = \frac{|T|}{2}$ it has the form $S' = <D_1, ..., D_m, D'>$ with $|D'| = 1$.

To eliminate the existence of multiple paths to the same (partial) solution (witch would be represented by different nodes), we apply an additional restriction that enforces the matches for a matchday to be added in a particular order: If $k < \frac{|T|}{2}$, we may only extend D_m with a match $<t_h, t_a>$ if D_m contains no match $<t'_h, t'_a>$ with $ord(t'_h) > ord(t_h)$, where ord defines some arbitrary but fixed total order on the teams.

Finally, a goal node is a node that is associated to a solution.

3 Search

3.1 IDA*

The IDA* algorithm (Korf, 1985) is a well known approach in the field of artificial intelligence to solve classical tree search problems. It combines the advantages of breadth first search and depth first search algorithms, that is requiring little memory while guaranteeing to find an optimal solution.

Algorithm 1 IDA*

```
1: procedure IDA*
2:     n_0 ← makeRootNode()
3:     f_cur ← 0
4:     solution ← none
5:     while solution = none do
6:         f_next ← ∞
7:         solution ← recursiveSearch(n_0, f_cur, f_next)
8:         f_cur ← f_next
9:     return solution
```

One way to implement IDA* is shown in Algorithm 1. When using IDA* for the TTP, there is no need for handling the case of an unsolvable instance, because TTPs are known to be always solvable. Therefore Algorithm 1 is a specialization of general IDA* search, where the unsolvable case is ignored.

IDA* search tries to find a solution throughout multiple iterations of depth first searches. Thereby, each of these depth first searches, executed by the procedure recursiveSearch, uses a limit f_{cur} to the f-values of nodes, that determines whether a node is expanded, i.e. its successors are created and examined. This limit increases with each iteration. The new limit is determined within the recursiveSearch procedure. Eventually, the limit to the f-value will be high enough for recursiveSearch to find a solution and the IDA* procedure will stop and return it. This solution is guaranteed to be optimal due to the properties of recursiveSearch, which will be shown in detail later.

The explained behaviour of Algorithm 1 is realized by first creating a root node (line 2) and setting the f-limit for the first iteration of the while loop to 0 (line 3). The root node has no parent assigned to it and consists of an empty schedule and an f-value that is set

to the heuristic estimate (cf. Chapter 4) of that schedule. After the creation of the root node and the assignment of f_{cur} the IDA* procedure starts with the while loop and runs it as long as no solution has been found (line 5). Within the while loop the f-limit for the next iteration is initialized to infinity (line 6). Then recursiveSearch is called with the root node and the two f-limits as parameters (line 7). After recursiveSearch returned f_{cur} is set to f_{next} (line 8), that has been updated by recursiveSearch to a proper next f-limit and is not equal to infinity anymore. Then, if recursive search has not returned a solution, a new iteration begins. If a solution to the problem has been found, the while loop will eventually stop and return the solution to the caller (line 9).

Algorithm 2 recursiveSearch

```
 1: procedure RECURSIVESEARCH(n, f_cur, f_next)
 2:     if f(n) > f_cur then
 3:         if f(n) < f_next then
 4:             f_next ← f(n)
 5:         return none
 6:     if isGoal(n) then
 7:         return n
 8:     for each n' ∈ successor(n) do
 9:         solution ← recursiveSearch(n', f_cur, f_next)
10:         if solution ≠ none then
11:             return solution
12:     return none
```

Algorithm 2 shows the recursiveSearch procedure. It first checks whether the f-value of the node given by the caller is greater than the current f-limit (line 2). In that case we can tighten the next f-limit (line 3-4) and return **none** (line 5). Thus, the given node n is not expanded, i.e. its successors are not generated, and the tree search proceeds one level higher in the search tree. If the f-value of the given node is lower than the next f-limit, recursiveSearch proceeds at line 6 and checks whether the given node is a goal node. If it is, the given node is returned to the caller as a solution.

Notice that until now all that recursiveSearch has done, was checking whether there is a reason to stop going deeper into the search tree at the given node. When no such reason has been found, all possible successors of the given node will be created and processed in a for loop (line 8). For each successor node, recursiveSearch is called recursively (line 9). If the return value of this call is a solution, it is handed to the caller (line 11). Otherwise, the next successor will be expanded. If no successor led to a solution, Algorithm 2 returns **none** (line 12) to indicate that there is no solution in this subtree within the given f-limit.

Figure 3.1 gives an example of a simple search tree with nodes $n_0, ..., n_{14}$ and their corresponding f-values. An IDA* search traverses this tree in the following manner:

- In the first iteration f_{cur} is zero. Therefore the root node n_0 has already a f-value exceeding f_{cur}. Thus, the value 5 will be set as the f-limit for the next iteration.

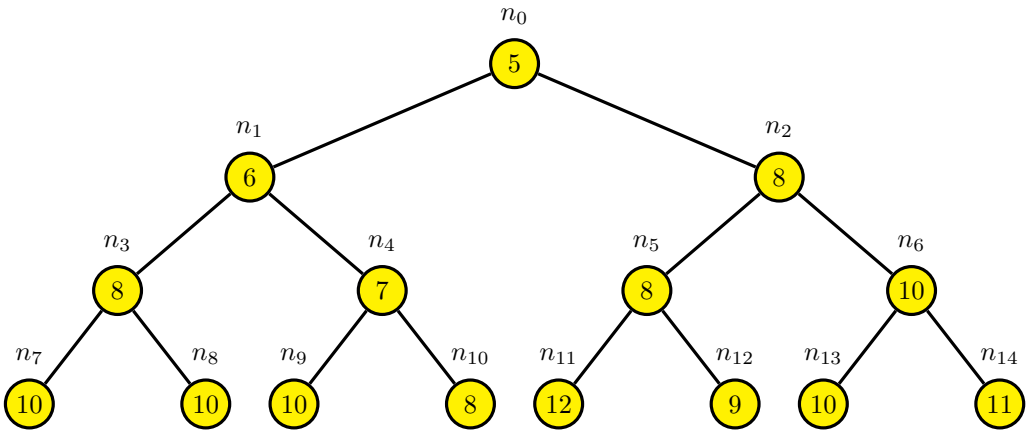


Figure 3.1: A simple search tree with a solution depth of 3, nodes $n_0, ..., n_{14}$ enumerated from top to bottom and from left to right and labeled f-values inside the nodes. Note that the root node is assigned a depth of 0.

- In the second iteration f_{cur} has a value of 5. Therefore the root node is expanded. But since the f-values of the children n_1 and n_2 both exceed f_{cur}, their are not expanded anymore and f_{next} is set to 6, which is the lowest f-value of all considered nodes that exceeds f_{cur}.

- In the third iteration f_{cur} has a value of 6. The nodes n_0 and n_1 can be expanded since their f-values do not exceed f_{cur}. So, every node up to and including node n_4 is considered. Thus, f_{next} is set to 7 because this is again the lowest f-value of all considered nodes that exceeds f_{cur}.

- In the fourth iteration f_{cur} has a value of 7. Thus, like in the previous iteration the nodes n_0 and n_1 are expanded. In addition n_4 gets expanded too. So, the considered nodes are all nodes up to and including n_4 and the two nodes n_9 and n_{10}. The lowest f-value of these nodes f-values exceeding 7 is 8. Therefore f_{next} is set to 8. Note that although the tree search went into the depth of a solution, no solution is found. The reason for this behaviour is that the goal test (Algorithm 2, line 6) is executed after the updating of f_{next} (Algorithm 2, lines 3 & 4). And since a solution, when it is seen for the first time, will always trigger the updating part of the procedure, the goal test can not be reached. This is an aimed behavior of the procedure. Suppose an earlier goal test is applied, then the guarantee for IDA* to find an optimal solution would be lost.

- In the fifth and final iteration f_{cur} has a value of 8 and a solution is found at node n_{10}.

3.2 Forced deepening

Forced deepening (FD) is an approach to reduce the number of iterations required by IDA* before finding a solution. It was introduced by Uthus et al. (2011). They also proofed that forced deepening does not affect the guarantee of the optimality of a solution as long as all solutions come from a constant depth and f is monotone. In a TTP instance, all solutions

come from a constant depth. Further, the Independent Lower Bound heuristic (cf. Chapter 4), applied to calculate f in this thesis, is monotone. Therefore, forced deepening can be exploited in this thesis without any concerns about loosing the guarantee of a solution's optimality.

Through forced deepening the number of iterations can be limited. This is done by defining an integer parameter $\lambda \geq 1$ and forcing the tree search in each iteration to go at least λ levels deeper into the tree than in the previous iteration. So nodes, whose f-values exceed the f-limit for an ongoing iteration (f_{cur}), might now, in contrast to the IDA* search without FD, still be expanded, as long as their depth does not go beyond the depth of the node where the f-limit was found in the previous iteration plus λ. However, there is one exception where nodes are not expanded, even though their depth level is sufficiently shallow. That is if their f-value exceeds not only f_{cur} but also f_{next}. In that case expanding the node would not make any sense, because this node could impossibly lower f_{next}.

We denote the depth of a solution as d_{solution}, the depth corresponding to the f_{cur}-value as d_{cur} and the depth corresponding to the f_{next}-value as d_{next}. Since by forced deepening it is guaranteed that d_{next} is at least as large as $d_{\text{cur}} + \lambda$, the maximum number of iterations whit forced deepening is limited to $\lceil \frac{d_{\text{solution}}}{\lambda} \rceil + 1$. But simultaneously the number of expanded nodes per iteration will be larger. This is because the algorithm can still search on at a certain node even though its f-value exceeds the current f-limit. This drawback however is pretty insignificant in cases where IDA* without FD goes through a large number of iterations. Furthermore, it is of course vital to chose λ properly, since a λ chosen too small still results in many iterations to be done and a λ chosen too big results in a greater overhead in every iteration by expanding many additional nodes that exceed the current f-limit.

Algorithm 3 recursiveSearch with forced deepening

```
1:  procedure RECURSIVESEARCH(n, f_cur, f_next, depth, d_cur, d_next)
2:      if f(n) > f_cur then
3:          if not (depth < d_cur + λ and f(n) ≤ f_next and d_cur < d_solution) then
4:              if f(n) < f_next or (f(n) = f_next and depth > d_next) then
5:                  f_next ← f(n)
6:                  d_next ← depth
7:              return none
8:      if isGoal(n) then
9:          return n
10:     for each n' ∈ successor(n) do
11:         solution ← recursiveSearch(n', f_cur, f_next, depth + 1, d_cur, d_next)
12:         if solution ≠ none then
13:             return solution
14:     return none
```

Algorithm 3 shows what applying forced deepening changes compared to Algorithm 2. First of all, it is now essential to keep track of the depth, which is why the *depth* variable was added. In the call to recursiveSearch (line 11) $depth + 1$ is given as a parameter to indicate that the depth gets bigger with each recursive step. Algorithm 3 also needs to keep track of the depth d_{next} of the node associated with the current f_{next}-value. Therefore the code on line 6 has been added. Note that when several different nodes have the same f_{next}

value the depth of the node that is the deepest is stored in d_{next} (added condition on line 4). The initial *depth* has to be passed by the IDA* procedure (Algorithm 1) as 0. In addition it has to pass and keep track of d_{cur} and d_{next}. The d_{cur} value for the first iteration is set to 0.

When applying FD and $f(n)$ exceeds the current f-limit, recursiveSearch can now, as mentioned above, still expand n as long as the condition on line 3 is not satisfied. This means, that recursiveSearch forces the algorithm to go deeper into the search tree when $depth < d_{\text{cur}} + \lambda$, $f(n) \leq f_{\text{next}}$ and $d_{\text{cur}} < d_{\text{solution}}$.

Of those three conditions $depth < d_{\text{cur}} + \lambda$ is the most important, that actually implements the forced deepening behavior. Through it recursiveSearch cannot return **none** (line 7) until it has reached a certain depth that is λ greater than the depth reached in the previous iteration.

The next condition is $f(n) \leq f_{\text{next}}$, which guarantees that the f-value $f(n)$ of the current node does not exceed the f-limit for the next iteration f_{next}. This is important because as soon as $f(n)$ is exceeding f_{next} the search in the subtree below n will not result in a better f_{next} value, than the one that we already have. Therefore searching these successors would be a waste of time and effort and is prohibited.

The last condition is $d_{\text{cur}} < d_{\text{solution}}$. It is on one hand there to improve performance by preventing forced deepening from being applied in the last iteration (where d_{cur} equals d_{solution}). So for the last iteration no additional nodes will be expanded and through that no overhead will be produced. On the other hand this condition is important for the guarantee of an optimal solution. If forced deepening is applied for the last iteration, the tree search would generate every node on the deepest level (d_{solution}) of the tree that has an f-value which does not exceed f_{next}. But, this would also allow the f-value of a solution found at node s to be in the range $f_{\text{cur}} < f(s) \leq f_{\text{next}}$ when it should actually be equal to f_{cur}. Hence, forced deepening is prohibited for the last iteration.

If $\lambda \neq 1$ one has to be aware that for late iterations $d_{cur} + \lambda$ could get bigger than $d_{solution}$. In that case $d_{cur} + \lambda$ is set to $d_{solution}$ to prevent errors. For better clarity this fact is omitted in Algorithm 3. Still, it should not be forgotten when implementing the algorithm.

Consider the tree from Figure 3.1 again. To clarify the differences of an IDA* search with FD to the search without FD, the manner of traversing the tree with FD and $\lambda = 2$ follows (compare to the manner of traversing without FD):

- In the first iteration f_{cur} and d_{cur} are zero. Therefore the root node n_0 has already a f-value exceeding f_{cur}. But since *depth* is only 0 for the root node and $d_{\text{cur}}+\lambda = 0+2 = 2$ exceeds it, $5 = f(n_0) < f_{\text{next}} = \infty$ and $0 = d_{\text{cur}} < d_{\text{solution}} = 3$, the search can expand the root node. The first successor node n_1 is on depth 1 of the search tree. Since $f_{\text{cur}} < f(n_1) < f_{\text{next}}$ and still $d_{\text{cur}} < d_{\text{solution}}$ this node can be expanded too. The generated nodes n_3 and n_4 have a depth of 2 and $depth < d_{\text{cur}}+\lambda$ is no longer satisfied. Thus, they cannot be expanded anymore. The lowest f-value of those two nodes is set as f_{next}. Now the tree search continues with generating the second successor n_2 of the root node. But since f_{next} is now 7, $8 = f(n_2) < f_{\text{next}}$ is not satisfied. Hence, n_2 is not expanded.

- In the second iteration f_{cur} has a value of 7 and d_{cur} has a value of 2. Now all nodes until depth $d_{\text{cur}}+\lambda = 2+2 = 4$ can be generated if their parents f-values do not exceed f_{next}. Since the tree goes only into a depth of 3, $d_{\text{cur}} + \lambda$ has now to be considered as 3 (spacial case described above). The sequence of generated nodes for this iteration starts with $n_0, n_1, n_3, n_7, n_8, n_4, n_9, n_{10}$ (as one can verify with Algorithm 3). After these nodes have been generated, f_{next} is 8 since it is the lowest f-value of the already generated nodes on level 3. Now only the sequence of nodes $n_2, n_5, n_{11}, n_{12}, n_6$ is still generated. The node n_6 is not expanded since its f-value of 10 does exceed f_{next}.
- In the third and last iteration f_{cur} has a value of 8 and d_{cur} has a value of 3. Since $d_{\text{cur}} = d_{\text{solution}}$ no forced deepening is applied. A solution is found and returned at node n_{10}. As predicted by $\lceil \frac{d_{\text{solution}}}{\lambda} \rceil + 1$ the number of iterations needed was 3.

3.3 Elite paths

When applying elite paths (EP) (Uthus et al., 2011), the recursiveSearch algorithm is expanded such that for each iteration it stores the path from the root node to the node where f_{next} was found. This is the so-called *elite path* for the upcoming iteration. In an IDA* search with EP recursiveSearch traverses the elite path first at the beginning of each iteration by choosing the first successor n' according to the elite path (Algorithm 4, line 10). Thus, recursiveSearch first searches the children and siblings of the node associated to f_{cur}.

Or to say it in other words: If we consider the node associated with f_{cur} as our last and best partial solution so far, then an elite path actually ensures that recursiveSearch first tries to expand this best partial solution to a new partial solution closer to a final solution. Any other partial solution is not tried until then. This works of course best, when the best partial solution for the next iteration is a descendant from the current partial solution.

To ensure that, whenever there is more than one best partial solution for an iteration, the partial solution with the largest depth is considered as the elite path, the condition on line 3 of the recursiveSearch procedure was added. To make the verification of this condition possible Algorithm 4 has to take track of the depth, which is why the depth variables *depth* and d_{next} were added.

Note that when elite paths are applied without forced deepening the number of expanded nodes stays the same for every iteration but for the last. For the last iteration a reduction of the number of expanded nodes is probable. The probability grows with the depth of the elite path in the last iteration. It is not necessarily the solutions depth because, in contrast to an IDA* search with forced deepening, it is possible that when a optimal solution has a parent with the same f-value, only the parent node of the optimal solution is generated in the second last iteration and not the node associated to the optimal solution itself. This can of course also apply to the parent of the parent of the solution and so on. Thus, the deeper the elite path in the last iteration already is, the higher is the probability that an optimal solution is located below the elite paths final node. If the elite path is more shallow, the probability grows that the solution is located in a different subtree than the elite path is in. In the optimal case, when the elite path has the depth of a solution, exactly $d_{\text{solution}} - 1 = n \cdot (n-1) - 1$ nodes along the elite path have to be expanded for the last

iteration.

If elite paths are applied in combination with forced deepening, they can reduce the number of additionally expanded nodes forced deepening considers for any iteration. This is due to the fact that the f_{next}-value is likely to decrease faster when elite paths are applied. And if f_{next} decreases faster, forced deepening will have to expand fewer nodes per iteration. This behaviour can be explained by looking at the condition on line 3 in the recursiveSearch procedure with forced deepening (Algorithm 3). If f_{next} decreases faster, $f(n) \leq f_{\text{next}}$ will be unsatisfied more often. Hence, the whole condition on line 3 will be satisfied more often and the recursiveSearch procedure will prevent the current node from expanding by returning with no solution (line 7).

Algorithm 4 recursiveSearch with elite paths

```
1:  procedure RECURSIVESEARCH(n, f_cur, f_next, depth, d_next, ep)
2:      if f(n) > f_cur then
3:          if f(n) < f_next or (f(n) = f_next and depth > d_next) then
4:              f_next ← f(n)
5:              d_next ← depth
6:              ep ← extractPath(n)
7:          return none
8:      if isGoal(n) then
9:          return n
10:     for each n' ∈ successor(n, ep) do
11:         solution ← recursiveSearch(n', f_cur, f_next, depth + 1, d_next, ep)
12:         if solution ≠ none then
13:             return solution
14:     return none
```

A special property of Algorithm 4 is that the elite path variable is both used to read the elite path of the previous iteration (line 10), as well as to store the elite path for the current iteration (line 6). On the first grasp this may seem problematic, but since in the course of an iteration all reading access (line 10) is performed before any writing access (line 6), this is a tolerable way to save memory space for one variable and simplify the algorithm. But why is it that all reading access takes place before any writing access? As one can see at line 2, for a writing access $f(n) > f_{\text{cur}}$ has to hold. But since on the elite path by definition every $f(n)$-value is smaller or equal to the f_{cur}-value, the writing access can never take place while traversing the elite path. The question still remains what happens in the successor function (line 10) after a writing access (line 6) took place? Isn't it possible that the newly assigned *ep* introduces errors to the successor function? Not if the successor function is able to determine whether the last reading access to *ep* for an ongoing iteration already took place. After that last access, the successor function simply changes its behaviour to that of a simple successor function that does not work with elite paths (like in Algorithm 2) for the rest of the iteration.

It turns out that a successor function, that is able to determine when the last reading access to *ep* for an ongoing iteration took place, is simple to realize. The overlaying IDA* procedure only has to count every call to the successor function for an iteration. Whenever the number of calls is higher then the length of the elite path, the last reading access to *ep*

has already taken place.

The IDA* procedure from Alogrithm 1 has to be changed further to work together with Algorithm 4. Mainly it has to ensure that the path extracted in recursiveSearch at line 6 is handed over as the elite path parameter to the first call of recursiveSearch in the next iteration of the IDA* procedure. It also has to hand over initial values for *depth* and d_{next}. Because these changes are minor, no adapted version will be shown here.

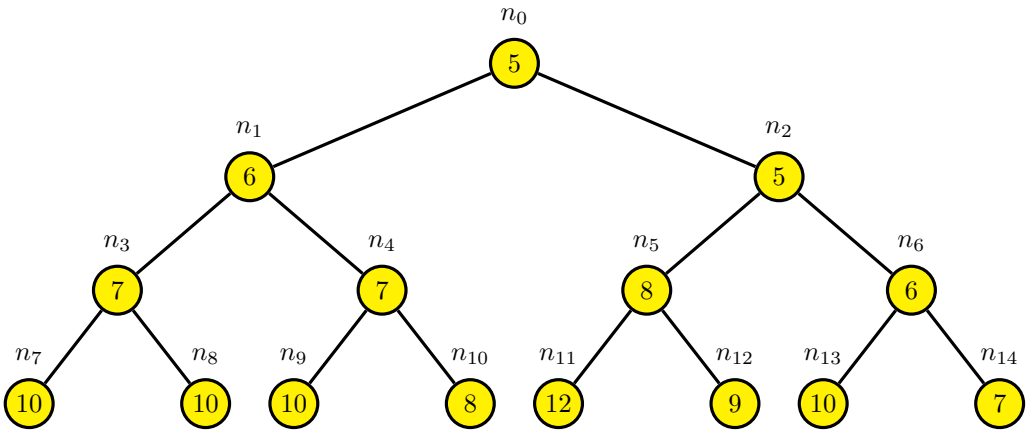


Figure 3.2: A simple search tree with a solution depth of 3, nodes $n_0, ..., n_{14}$ enumerated from top to bottom and from left to right, and labeled f-values inside the nodes. Note that the root node is assigned a depth of 0.

To give an example on how elite paths work, consider the search tree shown in Figure 3.2. The order of expanded nodes for this tree in the final iteration with $f_{\text{cur}} = 7$ is $<n_0, n_1, n_3, n_4, n_2, n_6>$ when not applying elite paths (cf. Algorithm 2) and given that left successors are expanded before right successors. When on the other hand elite paths are applied, the expanding order is $<n_0, n_2, n_6>$ since the elite path found in the previous iteration is the path from the root node to the node n_{14}. So instead of expanding 6 nodes of the search tree like IDA* without EP does, IDA* with EP expands only 3 nodes in order to find the solution at node n_{14}.

3.4 Subtree forests

Subtree forests (SF) extend the IDA* algorithm in a way that reduces the number of expanded nodes per iteration. Like FD and EP they were introduced by Uthus et al. (2011).

A *subtree forest* S is defined as the set of all nodes taken from a certain depth d in the original search tree associated with the path from the original root to this nodes. For example the subtree forest with $d = 2$ of the search tree shown in Figure 3.2 is $S = \{<n_0, n_1, n_3>, <n_0, n_1, n_4>, <n_0, n_2, n_5> <n_0, n_2, n_6>\}$.

A *subtree* is an element of a subtree forest and consists of the path from the root node to the frontier node of the subtree (i.e. the deepest node in the subtree), the frontier node itself and a variable f_{sub} that holds the lowest f-limit of an unexpanded node below the frontier node (not shown in the equation above).

The general idea behind IDA* with subtree forests is to handle subtrees individually within an iteration in an order that has a high probability of finding the next f-limit (or a solution) faster than without subtrees. This order should prefer the subtrees with low f_{sub}-values over those with high f_{sub}-values. In addition subtrees that cannot expand a node that has not yet been expanded in previous iterations can be skipped. This is the case for all subtrees that have a f_{sub}-value exceeding the f_{cur}-value of an iteration. *Subtree skipping* of course improves performance and is at the same time not affecting the optimality of the solution.

Algorithm 5 IDA* with subtree forest

```
1:  procedure IDA*
2:      subtreeQueue ← createSubtreeForest()
3:      f_cur ← 0
4:      solution ← none
5:      while solution = none do
6:          f_next ← ∞
7:          while subtreeQueue not empty do
8:              currentSub ← subtreeQueue.pop()
9:              f_sub ← currentSub.getF()
10:             if f_sub ≤ f_cur then
11:                 f_sub ← ∞
12:                 n ← currentSub.getNode()
13:                 solution ← recursiveSearch(n, f_cur, f_sub)
14:                 if solution ≠ none then
15:                     return solution
16:                 currentSub.setF(f_sub)
17:             if f_sub ≤ f_next then
18:                 f_next ← f_sub
19:             nextSubtreeQueue.push(currentSub)
20:         f_cur ← f_next
21:         subtreeQueue ← nextSubtreeQueue
```

Algorithm 5 shows IDA* with subtree forests. At the very beginning the subtree forest is created and stored in a priority queue called *subtreeQueue* (line 2). Note that, after that, no root node is created (in contrast to Algorithm 1) since the frontier nodes of the subtrees will be given to recursiveSearch instead of the root node. Then, as in the standard IDA* algorithm, the f-limit for the first iteration of the while loop is set to 0 (line 3) and the while loop is started with the f-limit for the next iteration set to infinity (lines 5 & 6).

Subtrees are ordered, as mentioned above, by the associated f-value. Before starting recursiveSearch on a frontier node, the corresponding subtree has to be inspected, using the concept of *subtree skipping* (line 10). Subtree skipping declares that every subtree whose f-limit exceeds the current iterations f-limit can be ignored (or skipped). A skipped subtree will not be explored and so many nodes are not expanded that would have been re-expanded in an IDA*-search without a subtree forest.

For the first iteration, however, the f_{sub}-values will all be zero. This is because they are initially set to zero in createSubtreeForest (line 2). So, at the beginning of the tree search every subtree is considered. But since f_{cur} is zero (line 3) the call handing the frontier

node to recursiveSearch (line 13) will not expand any nodes and will immediately set f_{sub} to the f-value of the frontier node. If however for later iterations a subtree is skipped, the algorithm will still update the f_{next}-value if necessary (lines 17 & 18) and push the skipped subtree into *nextSubtreeQueue* (line 19).

For each subtree that is not skipped the algorithm proceeds with resetting the f-limit for the current subtree to infinity (line 11). After that the node that is associated with the current subtree is assigned as frontier node (line 12). Now recursiveSearch is called with the reset f_{sub} to see whether it can find a solution below the current frontier node within the current limit f_{cur}. If it can, the found solution is returned. If not, the new f_{sub} value, found while executing recursiveSearch, is set as the f-limit of the current subtree (line 16). Note that this new f_{sub} value will always be greater than the old one since $f_{\text{sub}} \leq f_{\text{cur}}$ (line 10) was true for the old f_{sub} and the new f_{sub} can only be set (line 4 in Algorithm 2) if it is greater than f_{cur} (line 2 in Algorithm 2). As a next step, the f_{next} value will be updated if necessary and the current subtree will be pushed into *nextSubtreeQueue* before the algorithm proceeds with the next subtree.

If recursiveSearch has been called for all unskipped subtrees and none of them has returned a solution, *subtreeQueue* will eventually run empty and control flow will proceed at line 20. There, the variables f_{cur} and *subtreeQueue* are updated to be properly set for the next iteration. Note that the procedure shown is again a subtype of an IDA* algorithm that ignores the unsolvable case (cf. Chapter 3.1). Therefore, there will always be a next iteration when line 21 is reached. The outer while loop can only be exited when returning a solution (line 15).

3.5 IDA* with forced deepening, elite paths and subtree forests

In this section, I combined the IDA* algorithm with all three extensions seen above. This resulted in Algorithm 6. In red you can see the additional code segments that are added compared to Algorithm 5. There are also a few lines of code that are no longer needed (Algorithm 5, lines 17 & 18) since the update functionality of those lines is shifted into the recursiveSearch-method (as you will see below).

In an IDA* search with forced deepening, elite paths and subtree forests each subtree keeps track of the depth of the node with the best f-value and of the corresponding elite path. Mainly, the added lines in Algorithm 6 are for allocating, assigning and updating new variables needed to keep track of these values. The f_{sub}-variable, for example, which, after line 11, holds the f-limit of the current subtree, is handed over to recursiveSearch as a parameter and might be updated during its execution. After that, it is written back into the current subtree at line 20. Like f_{sub}, the variables f_{next}, d_{next}, d_{sub} and ep can also be updated within recursiveSearch.

In addition to these new variables, Algorithm 6 holds another important modification: The comparison that is responsible for the subtree skipping (line 14) now considers f_{next} as the skipping limit and not f_{cur} anymore. This is due to the combination of subtree forests with forced deepening. Therefore the skipping limit will now be narrowed during an iteration, whereas in Algorithm 5 it was a constant value in the course of one iteration. The

Algorithm 6 IDA* with FD, EP and SF

```
1:  procedure IDA*
2:      subtreeQueue ← createSubtreeForest()
3:      f_cur ← 0
4:      d_cur ← 0
5:      solution ← none
6:      while solution = none do
7:          f_next ← ∞
8:          d_next ← 0
9:          while subtreeQueue not empty do
10:             currentSub ← subtreeQueue.pop()
11:             f_sub ← currentSub.getF()
12:             d_sub ← currentSub.getDepth()
13:             ep ← currentSub.getElitePath()
14:             if f_sub ≤ f_next then
15:                 f_sub ← ∞
16:                 n ← currentSub.getNode()
17:                 solution ← recursiveSearch(n, f_cur, f_sub, f_next, depth(n), d_cur, d_sub, d_next, ep)
18:                 if solution ≠ none then
19:                     return solution
20:                 currentSub.setF(f_sub)
21:                 currentSub.setDepth(d_sub)
22:                 currentSub.setElitePath(ep)
23:             nextSubtreeQueue.push(currentSub)
24:         f_cur ← f_next
25:         d_cur ← d_next
26:         subtreeQueue ← nextSubtreeQueue
```

narrowing behavior can be easily put across by recognizing that at the beginning of each iteration f_{next} is set to infinity (line 7) and within recursiveSearch it is updated continuously to smaller and smaller values.

In Algorithm 7 it is shown in detail, what recursiveSearch does when applying all three presented extensions at the same time. Everything that is different compared to the recursiveSearch algorithms with only one added extension (Algorithms 3 & 4) is emphasized in red.

The only aspect that really changes is that recursiveSearch receives two additional parameters f_{sub} and d_{sub} to keep track of (what is done on lines 4-6). This is necessary because, as mentioned above, the f_{next}-value is already properly set within recursive search and not only later on outside of recursiveSearch (as in Algorithm 5, lines 17 & 18). So f_{sub} can no longer be handed over to recursiveSearch as f_{next} (like in Algorithm 5, line 13). Rather f_{sub} and f_{next} are handed over separately and recursiveSearch simultaneously keeps track of both these limits.

Note that since the initial caller of recursiveSearch (Algorithm 6) keeps track of a d_{sub} value for every subtree, the order in which the subtrees are handed over by the subtree queue (Algorithm 6, line 10) can be modified. The subtrees are now ordered by greatest d_{sub} value, using the f_{sub} value for tie-breaking. Thereby we give consideration to the fact that a subtree with a high d_{sub} value is associated to a partial schedule that has a high number of matches already scheduled and is therefore closer to a full schedule than lower

Algorithm 7 recursiveSearch with FD, EP and SF

```
1:  procedure RECURSIVESEARCH(n, f_cur, f_sub, f_next, depth, d_cur, d_sub, d_next, ep)
2:      if f(n) > f_cur then
3:          if not (depth < d_cur + λ and f(n) ≤ f_next and d_cur < d_solution) then
4:              if f(n) < f_sub or (f(n) = f_sub and depth > d_sub) then
5:                  f_sub ← f(n)
6:                  d_sub ← depth
7:              if f(n) < f_next or (f(n) = f_next and depth > d_next) then
8:                  f_next ← f(n)
9:                  d_next ← depth
10:                 ep ← extractPath(n)
11:             return none
12:     if isGoal(n) then
13:         return n
14:     for each n' ∈ successor(n, ep) do
15:         solution ← recursiveSearch(n', f_cur, f_sub, f_next, depth + 1, d_cur, d_sub, d_next, ep)
16:         if solution ≠ none then
17:             return solution
18:     return none
```

d_{sub}-valued subtrees.

Despite these little changes in Algorithm 7, there is still one subject to discuss: the one of the *ep*-parameter. As said in Chapter 3.3 the *ep* parameter is both used to read the elite path of the previous iteration (line 14), as well as to store the elite path for the current iteration (line 10). We have seen that this is permitted when using elite paths without forced deepening and subtree forests. But is it also unproblematic to use *ep* in this manner when all tree extensions are activated? It turns out that it is, as one can realize by looking at the conditions under which recursiveSearch can write onto *ep*. Again $f(n) > f_{\text{cur}}$ (line 2) has to be satisfied for that. For the elite path that stems form the subtree where f_{sub} equals f_{cur}, this condition is always broken (as shown in Chapter 3.3) and therefore the elite path is traversed completely before any writing access. But for elite paths that stem from subtrees where f_{sub} is bigger than f_{cur}, $f(n) > f_{\text{cur}}$ could be satisfied before the elite path is traversed completely. But however, from the additional conditions on line 3, that can be written as $depth \geq d_{\text{cur}} + \lambda$ **or** $f(n) > f_{\text{next}}$ **or** $d_{\text{cur}} \geq d_{\text{solution}}$ by applying De Morgan's law, only $f(n) > f_{\text{next}}$ can be true while being on an elite path (because d_{cur} in the first condition is equal to the depth of the elite path and the third condition is irrelevant to consider in this matter since it is only there to prevent forced deepening from being applied in the last iteration and has no further effect). But if $f(n) > f_{\text{next}}$, then $f(n) < f_{\text{next}}$ and $f(n) = f_{\text{next}}$ (line 7) cannot be true and therefore the writing access on *ep* (line 10) can never be reached while being on an elite path. So, having only one *ep*-variable per iteration is again unproblematic.

4

Heuristics

4.1 The independent lower bound heuristic – ILB

The *independent lower bound* heuristic (Easton et al., 2001) is used to estimate the minimal cost to extend a partial schedule of the TTP to a full schedule. The minimal distance is thereby estimated by finding a lower bound for it, meaning that there is no extension to a full valid schedule of lower cost. When calculating the ILB heuristic for an already full schedule, the result is 0 if the schedule is valid and ∞ if it is unvalid.

The independent lower bound consists of multiple lower bounds that are estimated separately for each team and summed up afterwards. Thereby the dependencies between teams are ignored. For example, after estimating team 1's minimal remaining traveling distance, which could induce team 1 to play an away game against team 3 on matchday 4, it would still be allowed for team 2 to consider playing against team 3 away on the very same matchday, when estimating team 2's minimal remaining distance to travel, although it is not possible for both, team 1 and 2, to play against team 3 at the same time.

This simplification allows a relatively fast calculation of the remaining traveling distance for a team, while guaranteeing that it is a lower bound. This is the case because the simplified calculation has less severe constraints for the completion of the schedule than the calculation where teams depend on each other.

To calculate lower bounds for teams, we will have to introduce a data structure that represents an away trip of a team. I will call this structure an *away sequence*, since it is a sequence of away games for the considered team. In this sequence possible opponents will be ordered such that the first opponent appearing in the sequence is also the first team played on the away trip associated with it, and so on.

The aim, when calculating a lower bound for team i, is to find every possible away sequence referring to an away trip team i can still perform, considering the given partial schedule. Since team i might have visited other teams already on the previous matchdays and every team is only visited once, these teams can no longer be part of an away sequence for team i. In addition an away sequence cannot be chosen in a way that violates the at most constraint (cf. Chapter 2.1), meaning that an away sequence is forbidden to have more than b members.

When all legal away sequences for team i have been determined, a combination of these

sequences has to be found that has the lowest traveling cost. This can be expressed as a constraint optimization problem: Let S be the set of all legal away sequences s_j for team i, d_j be the traveling distance associated with s_j and T be the set of teams that team i still has to play in an away game. Then the resulting traveling distance can be expressed as $d_{res} := \sum_j d_j \cdot x_j$, where $x_j \in \{0, 1\}$ is a variable that indicates whether s_j is a chosen away sequence. The constraint optimization problem to solve is:

$$\text{Minimize} \quad d_{res} = \sum_j d_j \cdot x_j$$

$$\text{Subject to} \quad \sum_{\{j|t \in s_j\}} x_j = 1 \text{ for all } t \in T$$

What it does is, minimizing the resulting traveling distance d_{res} such that in the set of chosen away sequences (indicated by x_j) all members are disjoint. So every team t, has to be in exactly one of the chosen subsets, which is ensured by summing up the indicator values x_j for every subset j that includes team t and ensuring that this sum adds up to one.

Note that this formulation of the constraint optimization problem by now only works for nodes where team i is not currently on an away trip (so, actually the heuristic only works for the root node and the goal nodes, since for every other state half the teams are on away trips). For a refined formulation of the constraint problem we will first have to discuss the following issue.

It is not sufficient, as proposed above for reasons of simplicity, to consider all legal away sequences s_j only once. This is because the associated traveling distances d_j depend on whether s_j is launching the away trip from its home location or continues an away trip, that it has already started on one of the previous matchdays. The latter case occurs whenever team i has played an away game on the previous matchday. In that case the away sequence is considered as *continuation* of an away trip already started. Therefore the size of such a sequence is not allowed to exceed the maximum sequence size b minus the number c of consecutive away games of team i at this point of the schedule.

When team i is currently on an away trip the *conclusion* of this trip, meaning that team i returns home before visiting any other teams, has to be considered as well. Therefore an empty away sequence is allowed to join the set of all legal away sequences. It will be assigned with the traveling distance required for team i to get from its current location to its home location. Thus, it is possible for a team to conclude its away trip, which was not the case without an introduction of an empty away trip.

As a consequence, whenever team i of which the lower bound is calculated is currently on an away trip, all away sequences that have a size small enough, have to be considered twice when solving the constraint optimization problem. And in addition the empty away sequence has to be considered as well. Therefore an additional set S' with members s_k is defined that contains all $s_j \in S$ with $|s_j| \leq b - c$ and an additional member $s' = <>$ that is an empty sequence. Each sequence $s_k \in S'$ is associated with a traveling distance d_k that is generally different from the distance d_j of the corresponding sequence s_j and s' is associated with the home traveling distance of team i.

The refined formulation of the constraint optimization problem is: Let $\mathcal{S} = S \cup S'$ be the set of all legal away sequences s_l for team i where $S = \{s_1, ...s_a\}$ and $S' = \{s_{a+1}, ..., s_{a+a'}\}$,

d_l be the traveling distance associated with s_l and T be the set of teams that team i still has to play in an away game. Then the resulting traveling distance can be expressed as $d_{res} := \sum_{l=1}^{a+a'} d_l \cdot x_l$, where $x_l \in \{0, 1\}$ is a variable that indicates whether s_l is a chosen away sequence. The constraint optimization problem to solve is:

$$\begin{aligned} \text{Minimize} \quad & d_{res} = \sum_{l=1}^{a+a'} d_l \cdot x_l \\ \text{Subject to} \quad & \sum_{\{l|t \in s_l\}} x_l = 1 \text{ for all } t \in T \\ & \sum_{k=a+1}^{a+a'} x_k = 1 \text{ if } a' > 0 \end{aligned}$$

The additional constraint ensures that whenever team i is currently on an away trip, exactly one continuation or the conclusion of an away sequence is among the chosen away sequences (indicated by x_l). Note that when team i is not currently on an away trip S' is an empty set and a' is therefore zero. This is why in that case the additional constraint can be ignored and the constraint optimization problem will be the same as before the refinement.

4.2 ILB as disjoint pattern database

In order to calculate an ILB heuristic value of a given partial schedule, it is not required to have complete information about the given schedule. It is sufficient to know 4 values for each team of the given problem instance. These values are the number of remaining away games, the set of teams to still play against away, the number of consecutive away games and, if the last match was an away match, the last opponent team.

One can exploit this fact by establishing a *pattern database* for each team that contains a precomputed lower bound value for every possible pattern of the 5 required values. The set of all pattern databases is then called a *disjoint pattern database* because the ILB heuristic is additive – meaning it can be calculated for every team individually and the sum of these individual values is the overall lower bound heuristic value.

If a disjoint pattern database is established before starting the tree search, the time to calculate heuristic values of nodes within the search tree is reduced in a big scale. For each required heuristic value there will only be need of determining the pattern for each team of the given problem instance, looking up the lower bound value of the determined patterns in the pattern databases and adding up these values to receive the ILB value.

Therefore the effort of establishing a disjoint pattern database before starting with the search for a solution is a well spent effort and is very likely to improve performance of the search drastically. However, the larger the problem instance is, the more memory space will be required for storing and the more time will be spent calculating the database entries. In fact, memory requirements and processing time grow exponentially with the number of teams in the problem instance. Nevertheless, for instances that are possible to solve in a reasonable time with today's hardware, the time and memory consumption of establishing a disjoint pattern database is normally not the bottleneck of the solution finding process.

5 Enhancement

In this chapter I present all other enhancements used in my implementation of the TTP-solver. They have been inspired by Uthus et al. (2011), who use the same enhancements.

5.1 Team reordering

Team reordering is a feature, where the static order in which the teams are tried to be allocated to matches of the schedule can be modified before starting the IDA* search.

Initially, teams are ordered by their appearance in the distance matrix. So, the team that has its distances to the other teams written on the first column of the distance matrix is called team 0 and is the first team that is tried to be allocated to a match, when using no special ordering. After that, team 1 will be tried (which consequently stems from column 2 of the distance matrix) and so on.

Different team orderings that can be chosen in my implementation are a random order, the ordering by maximal total distance of one team to all other teams and equivalently the ordering by minimal total distance.

5.2 Symmetry breaking

The search space of a TTP instance (characterized in Chapter 2.2) does already break some existing symmetries – namely, the symmetry of order of matches on a matchday. Since a matchday is considered as a set of matches in the problem definition (Chapter 2.1), schedules with matchdays that have the same matches scheduled in different orders have to be considered identical. Hence, a fixed total order of matches on a matchday is defined for the search space to generate only one path per identical schedule.

Symmetry breaking is a feature that further reduces the search space which has to be traversed by the IDA* algorithm in order to find the optimal solution. It does that by benefiting from symmetrical distances between teams, which are given by definition.

If the distance matrix for a TTP instance is symmetric, then every solution has the property that it results in another solution when the order of matchdays is reversed. I will call this process of reversing the order of all matchdays *mirroring*. Because of the symmetry

of distances the overall traveling distance of a mirrored solution stays the same as in the original solution.

So for an optimal solution there always exists a corresponding mirrored solution, which is optimal too. But since we only want to find one optimal solution and not all of them, we do only have to consider one solution of a pair of mirrored solutions when traversing the search tree.

Therefore, we eliminate half of the mirrored solutions as soon as we can, which is before the first match of the second half of the schedule. At this point we look at the remaining number of home and away games of a fixed team, e.g. the first team of the team ordering. If it has more home than away games still to come, we will not further pursue this subtree of the search tree. Uthus et al. (2011) call this symmetry-H. There also exists symmetry-A, which is almost the same, despite for the fact that we switch the roles of the remaining home and away games. This means, we check that the remaining number of away games is greater than the remaining number of home games for the considered team before the first match of the second half of the schedule.

Both symmetry breaking types exploit the fact that when half of the schedule is assigned, each team has played an uneven number of matches. This is because a full valid schedule has $2 \cdot (n-1)$ matchdays, where n is the number of teams in the given problem instance, and therefore a half full schedule has $(n-1)$ matchdays, which is uneven by definition.

5.3 Team cache

Even though calculating a heuristic value for a node in the search tree is already relatively efficient through the use of disjoint pattern databases, it can still be speed up by the usage of a *team cache*. A team cache exploits the property of the TTP that when assigning a new match to the schedule (or in other words when going deeper in the search tree) only the heuristic values for the teams that are involved in the newly assigned match change. So for all other teams the heuristic values stay the same.

Therefore, it is advisable to have a cache that stores the last seen heuristic value at every level of the search tree for every team. Thereby, when calculating the overall heuristic value for a specific node, the individual team heuristic values can be fetched from the parent node for all teams that are not involved in the last match assigned to the schedule. Thus only two values per node have to be read from the disjoint pattern databases.

This is more efficient since for reading from a pattern database, the calculation of the pattern is required which has linear costs in the number of teams, whereas reading from the team cache is a simple access with constant costs to a known index which does not have to be calculated first. The linear costs for the calculation of the pattern result from the generation of a hash value for the set of teams to still play against away. To have a distinct hash value for each possible set, every team in the set has to be considered (i.e. a loop through the (at most) $n-1$ opponent teams has to be performed).

So by using a team cache we again trade in additionally used memory for a better time performance. But since a team cache consumes much less memory than a disjoint pattern database, there is no reason not to apply it when already using disjoint pattern databases.

And even when no disjoint pattern databases are used the application of a team cache is advantageous, since without it the heuristic value for every team has to be calculated again at every node.

5.4 Multithreading

The computation time when solving problem instances with the help of subtree forests can further be reduced with multithreading. So far, subtrees have been searched one after another in a sequential manner. But this is not mandatory. They might as well be searched in parallel. This is relatively easy to achieve when the tree search is done without forced deepening and gets a little bit more complicated otherwise.

Algorithm 8 IDA* with FD, EP, SF and multithreading

```
1:  procedure IDA*
2:      subtreeQueue ← createSubtreeForest()
3:      f_cur ← 0
4:      d_cur ← 0
5:      solution ← none
6:      while solution = none do
7:          f_next ← ∞
8:          d_next ← 0
9:          while subtreeQueue not empty do
10:             currentSub ← subtreeQueue.pop()
11:             startThread(solution, f_cur, f_next, currentSub, nextSubtreeQueue)
12:         joinAll()
13:         if solution ≠ none then
14:             return solution
15:         f_cur ← f_next
16:         d_cur ← d_next
17:         subtreeQueue ← nextSubtreeQueue
```

First we will look at why parallelisation can be done without any concerns about loosing correctness of the algorithm when applying no forced deepening. In Algorithm 5 we see that for every subtree there is a call (line 13) to recursiveSearch (Algorithm 2) with f_{sub} given as what the recursiveSearch procedure considers as f_{next} . This is already the important point why parallelisation is so easy. Since only n, f_{sub} and the constant (throughout one iteration) f_{cur} are given to recursiveSearch, it can exclusively work with read-only variables (f_{cur}, n) and a subtree specific variable (f_{sub}) which therefore can only be manipulated by one thread at a time. So introducing multiple threats working in parallel can not have an influence on the result of recursiveSearch. But in the IDA* procedure it could still have influence and it turns out that it does interfere with the updating process of f_{next} (lines 17 & 18).

Since f_{next} can be reduced by any thread at any point in time, it is possible that, when two threads t_1 and t_2 run in parallel, thread t_1 reads f_{next} at line 17 and comes to the conclusion that it has to set f_{next} to a lower value because its f_{sub} value is smaller than f_{next}. But now, before t_1 can set the new f_{next} value, thread t_2 also realizes at line 17 that is has a lower f_{sub} value than the current f_{next}. Therefore t_2 updates f_{next} at line 18.

Afterwards, t_1 is also updating f_{next} at line 18 and thereby erases the update of t_2.

If this sequence of events takes place, it is possible that inconsistencies have been introduced. This is the case whenever t_2's f_{sub} value is lower than the one of t_1, because then t_1's f_{sub} value would be the f_{next} value for the continuation of the algorithm, which is false because f_{sub} of t_2 is lower. Therefore the lines 17 & 18 have to be made threadsafe by introducing a lock which has to be acquired before line 17 and released after line 18.

When in addition forced deepening is applied, things get slightly more complicated. If we look back at Algorithm 6, we can see all changes compared to Algorithm 5 marked in red. Only a part of these changes can have an impact on the behaviour of the algorithm when introducing multiple threads. These are the changes on line 12, 13, 14, 17, 21 and 22 All other changes can not have an influence since parallelisation is only applied within the while loop starting at line 9. From the remaining changes those which concern the read and write operations on the depths and paths associated to the subtrees (lines 12, 13, 21 & 22) are also unproblematic since they are subtree (and therefore thread) specific. So we only have to take a closer look to the call of recursiveSearch (line 17) and the altered skipping limit for the subtree skipping (line 14).

Algorithm 9 startThread

```
1:  procedure STARTTHREAD(solution, f_cur, f_next, currentSub, nextSubtreeQueue)
2:      f_sub ← currentSub.getF()
3:      d_sub ← currentSub.getDepth()
4:      ep ← currentSub.getElitePath()
5:      if f_sub ≤ f_next then
6:          f_sub ← ∞
7:          n ← currentSub.getNode()
8:          temp ← recursiveSearch(n, f_cur, f_sub, f_next, depth(n), d_cur, d_sub, d_next, ep)
9:          if temp ≠ none then
10:             solution ← temp
11:             return
12:         currentSub.setF(f_sub)
13:         currentSub.setDepth(d_sub)
14:         currentSub.setElitePath(ep)
15:     nextSubtreeQueue.push(currentSub)
```

Let us first look at the call to recursiveSearch (Algorithm 7). Unlike in the case without forced deepening the result of recursiveSearch now depends on the other threads, because what recursiveSearch considers as f_{next} is now in fact an actual reference to the f_{next} used in the IDA* procedure and no longer just the f_{sub} handed over as f_{next} parameter. And since f_{next} can be altered by any thread at any time and is read at lines 3 & 7 of recursiveSearch, the result depends on the progress of the f_{next} value. But what is important, is that the lowest possible f_{next}-value in an iteration will still be found and set as f_{cur} at the end of the iteration. It might not be found in a predictable manner (because the management of threads by the operating system itself is not predictable), but still it will be set correctly when all subtrees have been processed. This is because the node with the lowest f_{next}-value for an iteration will never satisfy $depth < d_{\text{cur}} + \lambda$ (because forced deepening is applied), which makes the critical condition $f(n) \leq f_{\text{next}}$ (line 3) needless to consider. Furthermore,

for said node $f(n)$ will always be lower or equal than the current f_{next} value and therefore satisfy the condition on line 7.

Keep in mind that again, to realize a threadsafe updating behaviour of f_{next}, it is important to introduce a lock which is acquired before line 7 and released after line 10. Amongst other things, this is realized in algorithm 10, which will be discussed below.

We still need to consider the impact of the altered skipping limit on line 14 of Algorithm 6 and whether it can do any harm when searching with multiple threads. Since the skipping limit is only used to prevent the IDA* procedure from doing useless work by considering subtrees that can not improve the f_{next} value, line 14 still fulfills its purpose. Whenever a subtree has a f_{sub} that exceeds f_{next} it cannot improve f_{next} regardless of whether this f_{next} value was found by single- or multithreading.

Algorithm 10 recursiveSearch with FD, EP, SF and multithreading

```
1:  procedure RECURSIVESEARCH(n, f_cur, f_sub, f_next, depth, d_cur, d_sub, d_next, ep)
2:      if isSolutionFound() then
3:          return none
4:      if f(n) > f_cur then
5:          if not (depth < d_cur + λ and f(n) ≤ f_next and d_cur < d_solution) then
6:              if f(n) < f_sub or (f(n) = f_sub and depth > d_sub) then
7:                  f_sub ← f(n)
8:                  d_sub ← depth
9:              acquireUpdateLock()
10:             if f(n) < f_next or (f(n) = f_next and depth > d_next) then
11:                 f_next ← f(n)
12:                 d_next ← depth
13:                 ep ← extractPath(n)
14:             releaseUpdateLock()
15:             return none
16:     if isGoal(n) then
17:         acquireSolutionLock()
18:         if not isSolutionFound() then
19:             setSolutionFound(true)
20:             releaseSolutionLock()
21:             return n
22:         else
23:             releaseSolutionLock()
24:             return none
25:     for each n' ∈ successor(n, ep) do
26:         solution ← recursiveSearch(n', f_cur, f_sub, f_next, depth + 1, d_cur, d_sub, d_next, ep)
27:         if solution ≠ none then
28:             return solution
29:     return none
```

Algorithms 8, 9 and 10 show versions of the IDA* and recursiveSearch procedures that use forced deepening, elite paths, subtree forests and multithreading. All code that was added compared to Algorithms 6 and 7 is emphasized in red. Note that to realize multithreading the procedure startThread has been split off from the original IDA* procedure. The two procedures together still contain all the lines of code of the original procedure.

To correctly understand the workflow of Algorithm 8, it has to be mentioned that the

call to startThread (line 11) is a non blocking call, which gets executed in a newly created thread. Further, the call to joinAll (line 12) is of course blocking until all threads have terminated. Furthermore, it is important that the subtree queue used in Algorithm 8 is a thread-safe queue.

Most of the changes that were introduced in Algorithm 9 and 10 influence what happens after a solution was found. First of all, in Algorithm 9 at line 8 a temporary variable *temp* is required to store the result of recursiveSearch. This is the case because *solution* is a variable that is shared by all threads and is likely to be overwritten by another thread in the time that passes between the storing (Algorithm 9, line 8) and the return of the solution to the caller (in Algorithm 8, line 14), whereby the solution would be lost.

Secondly, the changes between line 17 and 24 in Algorithm 10 are introduced to guarantee that only the first solution found over all threads is returned. Possible further solutions will be ignored and **none** will be returned (line 24). This is also why the condition $temp \neq$ **none** in Algorithm 9 at line 9 is sufficient to determine whether a temporary solution has to be made permanent or not.

To guarantee that only the first solution found over all threads is returned, the methods isSolutionFound and setSolutionFound are required. They manage a Boolean which stores, whether a solution has been found or not. In addition, a thread is only allowed to enter the if statement beginning at line 18, if it is the first thread to enter this statement. This is guaranteed through the solution lock acquired at line 17 and released at line 20. Thereby as soon as the if statement is entered no other thread can enter it. And after the lock is released, still no other thread can enter it because isSolutionFound is guaranteed to return **true**.

Third, to avoid that threads are still running, long after the solution was found and thereby blocking the IDA* procedure at line 12 of Algorithm 8, the first thing recursiveSearch does is returning **none** (line 3) if a solution has already been found (line 2). Through that, all threads are guaranteed to stop their search quite fast and terminate soon after that.

6 Evaluation

In this chapter I evaluate my implementation of a TTP-solver written in C++ with different settings and enhancements. The runs for my evaluation have been performed on a compute cluster located at the University of Basel. It has 24 nodes each with 2 eight core CPUs running at 2.20GHz and 64GB RAM. For each run I set a time limit of 24 hours. I did not set a customized limit to memory because the pattern databases fitted into the memory of the cluster and the IDA*-based search itself has very low memory requirements.

The independent lower bound heuristic, that I used to calculate the f-values for nodes in the search tree, requires solving constraint optimization problems. To do so, I used an open source C++ toolkit called *Gecode* (Gecode Team, 2006). It provided all data structures and functionalities needed to define and solve the constraint optimization problem formulated in Chapter 4.1.

For the evaluation I worked with some predefined problem sets of the TTP. They consist of a list of team labels with a corresponding distance matrix.

The first problem set is the SUPER set (introduced by Uthus et al., 2009) that is based on the Super 14 Rugby League, a league with 14 teams from South Africa, New Zealand and Australia. By now this league has an odd number of teams (15) and is thereby no longer an instance of the TTP. The SUPER set still refers to the old league with 14 teams.

The second problem set is the GALAXY set (introduced by Uthus et al., 2011), that is based on the distances between some of the exoplanets in the universe and earth. There are 40 exoplanets (earth included) in the set which are considered as teams playing in an intergalactic league.

It is possible to consider only subsets from either the SUPER or the GALAXY set. In order to do so, the number of team labels and the dimension of the distance matrix is reduced to an even number smaller than the original number of teams. The SUPER6 instance for example consists of the first 6 team labels and a corresponding 6×6 distance matrix.

Without multithreading my implementation was able to solve problem instances with up to 8 teams for both the SUPER and GALAXY problem set. Table 6.1 shows the best configuration for each problem instance and the required time. Forced deepening (with one exception for the smallest SUPER instance) and elite paths are always activated in the best configuration. They seem to be undoubtedly profitable when trying to solve the TTP as

	time required	FD	EP	SF	SB	TO	SD	λ
SUPER4	< 1msec	×	✓	×	none	none	-	-
SUPER6	1.20sec	✓	✓	✓	symmetry-H	max total	$n/2$	n
SUPER8	1h 09min	✓	✓	✓	symmetry-A	max total	$n/2$	n
GALAXY4	< 1msec	✓	✓	×	none	none	-	n
GALAXY6	1.27sec	✓	✓	×	none	none	-	n
GALAXY8	1h 24min	✓	✓	✓	none	none	$n/2$	$2n$

Table 6.1: Fastest runs for each problem instance: SB = symmetry breaking, TO = team ordering, SD = subtree depth, n = number of teams in the problem instance

FD	EP	SF	SUPER4	GALAXY4	SUPER6	GALAXY6	SUPER8	GALAXY8
×	×	×	100	100	100	100	-	100
×	×	✓	**45.31**	**35.91**	35.41	96.95	-	99.70
×	✓	×	83.20	83.43	99.99	100.58	-	101.07
×	✓	✓	54.30	37.02	36.01	97.10	-	99.70
✓	×	×	278.91	141.44	0.57	36.83	100	17.23
✓	×	✓	91.80	63.54	**0.16**	12.23	97.18	10.09
✓	✓	×	103.52	72.10	0.20	**10.73**	**92.25**	11.95
✓	✓	✓	81.64	57.46	0.18	11.65	94.37	**9.77**

Table 6.2: Relative number of expanded nodes: The value where none of the extensions were activated is considered 100% (Except for SUPER8, where the fifth line is considered 100%). Where SF was applied, the depth of the subtree forest was $\frac{n}{2}$ with n given as the number of teams in the problem instance. Symmetry breaking, team reordering and multithreading were not activated and $\lambda = n$ for the forced deepening.

fast as possible. Subtree forest as well seem to be profitable in terms of time performance for large instances. Apparently, the best choice for the depth of a subtree forest is $n/2$ and for λ it is n (although for GALAXY8 $2n$ was the better choice). Symmetry breaking and team reordering only improved time performance for large SUPER instances.

6.1 Forced deepening, elite paths and subtree forests

I will discuss the impact of the three basic extensions (FD, EP and SF) on the standard IDA* algorithm with the help of the values given in Table 6.2. It shows the relative number of nodes that were expanded in order to find an optimal solution to the given problem instances. For the SUPER8 instance not all values are given. This is because these values could not be found within a reasonable time.

First I will discuss the impact when applying forced deepening. We can see that in cases where FD is applied there are often much fewer expanded nodes than without FD. This effect is particularly strong for the SUPER6 problem instance, where less than 1% of nodes are expanded independent of whether the other 2 extensions are activated. For instances with only 4 teams, however, there are also configurations where FD is harmful, in the worst case leading to 179% more expansions. Both observations can be explained by considering the

FD	SUPER4	GALAXY4	SUPER6	GALAXY6	SUPER8	GALAXY8
×	5	6	2510	73	>2045	125
✓	3	3	6	6	8	8

Table 6.3: Number of iterations with and without applying FD

	$\lambda = 1$	$\lambda = 2$	$\lambda = n/2$	$\lambda = \lfloor \frac{2}{3}n \rfloor$	$\lambda = n$	$\lambda = 2n$
SUPER4	**201**	233	233	233	209	202
SUPER6	1061190	952592	**904182**	909246	949001	947642
SUPER8	$4.099 \cdot 10^9$	$3.050 \cdot 10^9$	$2.713 \cdot 10^9$	$2.538 \cdot 10^9$	$\mathbf{2.505 \cdot 10^9}$	$2.538 \cdot 10^9$
GALAXY4	325	252	252	252	**208**	294
GALAXY6	2330961	1805716	1445428	1416313	**1112714**	1507564
GALAXY8	$4.297 \cdot 10^9$	$3.214 \cdot 10^9$	$2.314 \cdot 10^9$	$2.288 \cdot 10^9$	$\mathbf{2.039 \cdot 10^9}$	$2.041 \cdot 10^9$

Table 6.4: Number of expanded nodes depending on the choice of λ: The lowest number for every problem instance is written in bold. All runs have been performed with FD, EP and SF but without symmetry breaking or team reordering. The depth of the subtree forest was $\frac{n}{2}$ with n given as the number of teams in the problem instance.

number of iterations that can be saved through FD (Table 6.3). Note that the application of EP and SF does not influence the number of iterations, which is why the configuration of EP and SF is not specified in Table 6.3. We can see that for instances with 4 teams we only save 2 or 3 iterations. But for instances with more than 4 teams (especially the SUPER6 and SUPER8 instances) there is a drastically lower number of iterations. Thus, when applying FD on a 4 team instance there are indeed a few iterations less, but this is not an advantage (node expansion wise) because each of the remaining iterations has a larger number of expanded nodes. For instances with many teams, however, this larger number of expanded nodes can be neglected because so many iterations are saved. But for small instances they do in fact matter and it is better not to apply forced deepening on them.

I further examined the influence of the parameter λ on forced deepening. Table 6.4 shows the corresponding results. Apparently, the best result for problem instances with a large number of teams are achieved when λ is equal to n. This is, although valid for small GALAXY instances, not valid for small SUPER instances. However, the gain from using any other λ than n is minor. For the SUPER6 instance for example there are about 45 thousand fewer nodes to be expanded. Since in my implementation there are roughly 500 thousand nodes expanded per second, the gain of performance is very little in both of the small SUPER instances. I would therefore recommend to use a λ-value of n when trying to solve new instances.

The impact of elite paths on the number of expanded nodes is, as expected, a lot weaker than the one of FD. Often EP even appear to slightly slow down the algorithm (e.g. when FD and SF are applied in SUPER6). It is hard to identify some principle impact of EP on the number of expanded nodes. There seems to be a large amount of randomness involved in the application of EP. This is of course linked up with the fact that EP work best when best partial solutions for an iteration are found by expanding the best partial solution of

	SD = 1	SD = $n/2$	SD = n	SD = $\frac{n(n-1)}{2}$
SUPER4	232	209	128	**79**
SUPER6	1177876	949001	**781826**	-
SUPER8	2549564855	**2504874152**	-	-
GALAXY4	243	208	131	**79**
GALAXY6	1031381	1112714	**975263**	-
GALAXY8	2419616487	**2039468168**	-	-

Table 6.5: Number of expanded nodes depending on the choice of SD: The lowest number for every problem instance is written in bold. All runs have been performed with FD, EP and SF but without symmetry breaking or team reordering. The parameter λ was n with n given as the number of teams in the problem instance.

the previous iteration. But whether and how often this is the case is hard to foresee and highly dependent from the properties of the considered problem instance.

As expected, applying subtree forests as well does lower the number of expanded nodes in general. Although, there seem to be some exceptions when also applying FD and EP (second last and last row in Table 6.2). In that case, a third of the problem instances that I tested, required more node expansions in order to find an optimal solution. It is hard to tell, why this is the case, since the occurrences of these events seem to be quite random within the given data. Once they happen in a SUPER instance, once in a GALAXY instance. They happen in a small problem instance (with 6 teams) as well as in a larger problem instance (with 8 teams). However, the amount of gains or losses in the number of expanded nodes for large instances is around 2%. Therefore, if someone is risk-avers it might still be a good choice to apply subtree forests, even when both forced deepening and elite paths are already applied. Because the impact for other cases of the given data is quite large (e.g. SUPER 6 on row 1 and 2 of Table 6.2), it seem to be advisable to always apply subtree forests when either forced deepening or elite paths are not applied.

I also tested the influence of the depth of a subtree forest on the number of expanded nodes (Table 6.5). For this purpose I compared the cases where there are subtrees for each valid partial schedule where the first match is scheduled (subtree forest depth 1), where all matches of the first matchday are scheduled (subtree forest depth $n/2$), where all matches of the first two matchdays are scheduled (subtree forest depth n) and where half of the schedule has been scheduled (subtree forest depth $\frac{n(n-1)}{2}$) . Having subtree forest where the depth is bigger than $n/2$ has proven to be unfeasible for bigger instances since the number of subtrees in the forest explodes and they no longer fit into memory. Therefore I would recommend to choose a subtree forest depth of $n/2$ because this size was unproblematic in memory consumption for all tested instances. For small instances subtree forests with a larger depth can be advisable, as long as they still fit into memory.

Finally, when I compare my evaluation of the three extensions forced deepening, elite paths and subtree forsests to the empirical results of Uthus et al. (2011), I can state that they are congruent with each other. Especially, the values in Table 6.2 are very similar to the values evaluated in Uthus et al. (2011) – although they used $\lambda = 1$ instead of $\lambda = n$.

TO	SUPER4	GALAXY4	SUPER6	GALAXY6	SUPER8	GALAXY8
none	100	100	100	100	100	100
max total	121.05	147.12	99.15	95.82	99.83	100.21
min total	95.69	100	93.91	99.83	100.04	99.17
random	95.69	147.12	101.91	96.02	99.97	99.44

Table 6.6: Relative number of expanded nodes: Values have been obtained whit all 3 extensions (FD, EP and SF) activated. The number of nodes expanded when no team reordering is applied is considered 100%. No multithreading or symmetry breaking was applied. The subtree depth is $\frac{n}{2}$ with n given as the number of teams in the problem instance and $\lambda = n$.

6.2 Team reordering

The impact of team ordering on the number of expanded nodes is not quite clear. In some problem instances the number of expanded nodes decreases, in other problem instances it increases (Table 6.6). To examine whether the team ordering might have in fact a random impact on the number of expanded nodes, I implemented an additional random team ordering. As you can see, the random ordering as well performs better in some cases, but worse in other cases. These random values are, however, not by any means representative and are to be considered with caution, since they stem from a single run. Nevertheless, we can conclude that there is probably some sort of randomness involved in the relation between team ordering and number of expanded nodes.

What stands out is that for instances with a large number of teams (e.g. SUPER8 and GALAXY8), team ordering does not really affect the number of expanded nodes. We observe that changes are only within a range of one percent, regardless of whether the change is for the good or for the bad. Obviously, this is not a very pleasing behaviour. In addition, for small instances (e.g. SUPER4 and GALAXY4), we can quite often see that, in cases where the computational effort required grows, a change of team ordering does heavily increase the number of expanded nodes, whereas, in cases where the effort drops, the team reordering only induces a little decrease in number of expanded nodes. So, positive effects are outweigh by negative effects for the small instances considered.

In conclusion, since team reordering is not sure to improve performance when solving an arbitrary problem instance, applying it is in general not helpful in order to speed up computations. It might be helpful, though, if we had additional information about the optimal schedule (e.g. if we would know that team 1 should be playing a lot of home games early in the schedule). But in reality we never have this kind of information. Therefore, I would advice not to apply team reordering when solving TTP instances.

6.3 Symmetry breaking

The influence of symmetry breaking on the number of expanded nodes when solving a TTP is shown in Table 6.7. We can see that, although the solution space gets smaller by applying symmetry breaking, a solution is in general not found within fewer steps. E.g. for some small instances (SUPER4, GALAXY4 and SUPER6) the usage of symmetry-A causes

SB	SUPER4	GALAXY4	SUPER6	GALAXY6	SUPER8	GALAXY8
none	100	100	100	100	100	100
symmetry-A	122.49	120.19	141.58	83.98	88.08	105.46
symmetry-H	97.13	101.92	87.24	95.83	3013.26	109.31

Table 6.7: Relative number of expanded nodes: Values have been obtained whit all 3 extensions (FD, EP and SF) activated. The number of nodes expanded when no symmetry breaking is applied is considered 100%. No multithreading or special team ordering was applied. The subtree depth is $\frac{n}{2}$ with n given as the number of teams in the problem instance and $\lambda = n$.

my implementation to expand more nodes than in the case without symmetry breaking. This is also the case for GALAXY 8. The amount of additionally expanded nodes is thereby quite high for some instances (up to 40%). An even worse outcome can be observed for the SUPER8 instance combined with symmetry-H. There the amount of expanded nodes is more than three times higher than in the non symmetry breaking case. Apparently, in this case the solutions that can be found quickly are all eliminated by symmetry breaking when half of the schedule is fixed, leaving the algorithm with only solutions that are found slowly, meaning that these solutions are not found in subtrees that are located near the front of the subtree queue or that these solutions have lots of early matches that involve teams that appear at the end of the team ordering.

However, in less than half of the cases that I tested, symmetry breaking did improve performance of the tree search. Yet the improvement was, especially when compared with the possible degradations, not very high (up to 12%). So, symmetry breaking seems to be a risky enhancement for my implementation and I would not recommend to use it.

6.4 Multithreading

For my implementation of a multithreaded tree search, I used datastructures partly based on examples from the book by Williams (2012). The experiments shown in this section have been performed in another setting than the other experiments shown in this chapter. This was due to the incompatibility of the compute cluster with multithreaded programs. The hardware I used for the following experiments was a laptop computer with 4 cores that simulate 8 virtual cores running at 2.00GHz with 3.8GB RAM.

In Table 6.8 we see that computation time is, as expected, reduced for both tested problem instances when increasing the number of threads. The time reducing impact is clearest in the steps from 1 thread to 2 threads (about 45%) and from 2 threads to 4 threads (about 40%). In the last step time reduction is only about 12%. This is due to the fact that the hardware of the test setting has only 4 actual cores. The other 4 cores are virtual. Therefore time reduction in the last step cannot be as significant as in the first two steps.

We can also see that the number of expanded nodes slightly increases with the number of threads. This behaviour can be explained with the dynamically changing f_{next}-value that is crucial for the decisions whether subtrees are skipped and whether successors of nodes in the search tree are generated. When subtrees are searched in parallel, they tend to have a higher

	1 thread			2 threads		
	time	nodes	nodes/s	time	nodes	nodes/s
SUPER8	122mins 40s	$2.50 \cdot 10^9$	340338.4	66mins 36s	$2.50 \cdot 10^9$	626388.8
GALAXY8	128mins 52s	$2.04 \cdot 10^9$	263777.7	69mins 20s	$2.03 \cdot 10^9$	488285.2
	4 threads			8 threads		
	time	nodes	nodes/s	time	nodes	nodes/s
SUPER8	39mins 3s	$2.50 \cdot 10^9$	1068347.8	34mins 7s	$2.52 \cdot 10^9$	1230007.7
GALAXY8	41mins 46s	$2.06 \cdot 10^9$	823710.7	36mins 28s	$2.12 \cdot 10^9$	967260.0

Table 6.8: Performances of multithreaded runs: All runs have been performed with FD, EP and SF but without symmetry breaking or team reordering. The depth of the subtree forest was $\frac{n}{2} = 4$ with n given as the number of teams in the problem instance and λ was $n = 8$.

	DPD	TC	required time		DPD	TC	required time
GALAXY4	✓	✓	150msecs	GALAXY6	✓	✓	1.68secs
	✓	×	150msecs		✓	×	3.21sec
	×	✓	829msecs		×	✓	>24h
	×	×	825msecs		×	×	>24h

Table 6.9: Performances of runs when either disjoint pattern database (DPD), team cache (TC) or both are deactivated: All runs have been performed with FD, EP and SF but without symmetry breaking or team reordering. The depth of the subtree forest was $\frac{n}{2}$ with n given as the number of teams in the problem instance and λ was n.

f_{next}-value at the beginning of and throughout the search in this subtree as they would have had without parallelization. Therefore the tree search tends to expand more nodes and to skip fewer subtrees. Nevertheless, the time required for a multithreaded search is lower than in the single-threaded case. This is due to the expanded nodes per second rate that gets significantly higher the more threads are employed.

6.5 Team cache and disjoint pattern database

The influence of a team cache and a disjoint pattern database on the time performance of my implementation is shown in Table 6.9 using the two problem instances GALAXY4 and GALAXY6. For GALAXY4 only the impact of the disjoint pattern database can be recorded. The instance can be solved almost four times quicker with a disjoint pattern database. There is no recordable impact of a team cache for GALAXY4. For the larger instance GALAXY6 we can see that a team cache does double the speed of the solution finding process. When not applying a disjoint pattern database for GALAXY6, an optimal solution cannot be found in the course of 24 hours, whereas with a disjoint pattern database the solution is found in about 2 seconds.

Thus, a team cache and a disjoint pattern database are, as expected, vital enhancements when trying to find optimal solutions for the TTP efficiently.

7
Conclusion

I implemented an optimal solver for the Traveling Tournament Problem, a very hard problem that is the subject of numerous past and recent scientific work. In my implementation I realized an iterative-deepening A* algorithm with various extensions and enhancements and tried two similar approaches to calculate heuristic values - the direct independent lower bound heuristic and the independent lower bound heuristic as a disjoint pattern database. In the following I evaluated all implemented extensions and enhancements on the SUPER and GALAXY problem sets to verify their benefits to the performance of the tree search. In summary I can say that, while all three implemented extensions – forced deepening, elite paths and subtree forests – are beneficial for large problem instances, not all enhancements did consistently improve the performance of my implementation. Some enhancements, namely symmetry breaking and team reordering, have proven to be risky to apply, since they show an improvement for some problem instances but harm the performance significantly for other problem instances. The usage of a team cache and multithreading however have shown only performance improvements. They do, in combination with all three extensions and a disjoint pattern database, strongly improve the standard IDA* algorithm, that is initially only able to solve problem instances with 4 teams in a reasonable amount of time.

Bibliography

Easton, K., Nemhauser, G., and Trick, M. (2001). The Traveling Tournament Problem description and benchmarks. In *Principles and Practice of Constraint Programming — CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 580–584. Springer-Verlag.

Easton, K., Nemhauser, G., and Trick, M. (2003). Solving the Travelling Tournament Problem: A combined integer programming and constraint programming approach. In *Practice and Theory of Automated Timetabling IV*, volume 2740 of *Lecture Notes in Computer Science*, pages 100–109. Springer-Verlag.

Gecode Team (2006). Gecode: Generic constraint development environment. Available from: http://www.gecode.org.

Hafidi, Z., Talbi, E., and Goncalves, G. (1995). Load balancing and parallel tree search: The MPIDA* algorithm. In *Proceedings of the Fifth International Conference on Parallel Computing (ParCo'95)*, pages 93–100. Elsevier.

Irnich, S. (2008). A new branch-and-price algorithm for the Traveling Tournament Problem. Presented at the *International Workshop on Column Generation 2008*, Aussois, France, June 17-20, 2008. Available from: https://www.gerad.ca/colloques/ColumnGeneration2008/slides/SIrnich.pdf [Accessed 3 Feb, 2015].

Irnich, S. (2010). A new branch-and-price algorithm for the Traveling Tournament Problem. *European Journal of Operational Research*, 204(2):218 – 228.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

Korf, R. E. and Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1–2):9–22.

Powley, C. and Korf, R. E. (1991). Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477.

Rao, V. N., Kumar, V., and Ramesh, K. (1987). A parallel implementation of iterative-deepening-A*. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, pages 178–182. AAAI Press.

Uthus, D. C., Riddle, P. J., and Guesgen, H. W. (2009). DFS* and the Traveling Tournament Problem. In *Integration of AI and OR Techniques in Constraint Programming*

for Combinatorial Optimization Problems, volume 5547 of *Lecture Notes in Computer Science*, pages 279–293. Springer-Verlag.

Uthus, D. C., Riddle, P. J., and Guesgen, H. W. (2011). Solving the traveling tournament problem with iterative-deepening A*. *Journal of Scheduling*, 15:601–614.

Williams, A. (2012). *C++ Concurrency in Action*. Manning.