Sports Scheduling: An Artificial Intelligence Approach

David C. Uthus

A thesis submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

The University of Auckland
July 2010

ABSTRACT

This thesis looks at the Traveling Tournament Problem (TTP) from the sports scheduling literature. It presents two approaches to this problem: a metaheuristic Ant Colony Optimization (ACO) approach to find good solutions in a reasonable time frame and a heuristic search Iterative-Deepening-A* (IDA*) approach to find optimal solutions.

The first approach combines ACO with constraint processing techniques in order to handle the hard constraints of the TTP. The key component is creating a framework which uses forward-checking and conflict-directed backjumping to handle the constraints while using ACO for choosing the values. This is further improved by introducing new ideas of unsafe backjumping and pattern matching for constraint propagation while incorporating an old concept of ant restarts. This approach has been found to improve on past ACO approaches to the TTP and showed results which are more competitive with state-of-the-art metaheuristic approaches.

The second approach presents a parallel version of IDA*, combining past concepts of tree decomposition and node ordering with a new idea of subtree skipping. This new idea allows for parts of the search tree to be skipped for some iterations while still guaranteeing optimality for the final solution that is found. Two additional ideas are presented. The first, called forced deepening, helps to reduce node expansion when applying IDA*-like algorithms on real-world distance problems. The second, called elite paths, helps to both improve the performance of forced deepening while also allowing for the optimal solution to be found faster during the final iteration of IDA*. The results of applying this new approach to the TTP shows that it is state-of-the-art, finding known optimal solutions in a fraction of the time of past approaches and finding new optimal solutions to some unsolved problem instances.

Acknowledgements

I thank my supervisors Patricia Riddle and Hans Guesgen for all the help and guidance they have given me throughout these years of research. Thank you both so much for your time, advice, great ideas, and your friendship. I will greatly miss working with you both.

I also thank ScienceIT, who have helped greatly providing the hardware and software needed to make possible the experiments for this work.

I give thanks to Michael Trick, who helped get me started down the road of sports scheduling, and more importantly, the traveling tournament problem.

In addition, I thank Brad Clement and Michael Sims of NASA, who hosted my talks at their respective facilities, which allowed me to further refine my ideas.

Finally, I thank my wife, Emi, who has helped me in so many ways to become a better thinker, questioner, and scientist in the end.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Give me a place to stand, and I shall move the earth.

-Archimedes

From the early sporting competitions in Mesopotamia, Egypt, and the more organized and well known games of ancient Greece [31] to the modern day international sporting events and large-scale sporting leagues, sports have been an integral part of human society. Since ancient times, the organization of sports has grown more complex, with professional sports leagues now consisting of many teams playing long schedules. At the extreme is the North American Major League Baseball (MLB) schedule, involving 30 teams playing 162 games each. The creation of these sports league schedules has become a difficult task. Beyond the complexity of creating a schedule to be contained within a specified time period, there are also the issues of venue availability, travel time and cost, fairness in schedules between the teams, along with a multitude of other constraints and desired goals.

Sports scheduling has been addressed by the research community for almost 40 years now and has spread to address a wide variety of topics [27, 39]. These topics include real-world scheduling, referee assignments, round robin tournament scheduling, and math problems derived from sports scheduling. It has also attracted interest from different research communities such as operations research and artificial intelligence.

A theoretical problem was recently created which was inspired by the difficulty of scheduling a season for MLB. This problem, the Traveling Tournament Problem (TTP) [16], has become a benchmark in sports scheduling, with a variety of approaches having been applied to it. It has been difficult to solve with only the smallest of instances solved to optimality while the best solutions for the larger problem instances have required a large amount of computation time.

We present two new approaches to the TTP. The first approach is based on Ant Colony Optimization (ACO) [15] and is used for finding good solutions in a reasonable amount of time. This is accomplished by combining ACO with constraint processing techniques and introducing a new idea of using pattern matching for constraint propagation. This approach has been shown to outperform past ACO approaches to this problem and exhibits results that are competitive with other metaheuristics.

The second approach presented is based on Iterative-Deepening-A* (IDA*) [29], an exact algorithm for finding provably, optimal solutions. This has been modified to create a new algorithm called Concurrent Iterative-Deepening-A* (CIDA*) that combines ideas of various past approaches for parallelizing IDA* and node ordering along with some new ideas for reducing node expansion. This approach has been shown to outperform all past approaches for finding optimal solutions to the TTP and has been able to find optimal solutions for a number of unsolved instances.

1.1 Contributions

The contributions of this work are three-fold. The first aspect is from the sports scheduling perspective. In the past, operations research has been used for finding optimal solutions to the TTP. CIDA* is the first AI-centric approach for finding optimal solutions, and has found results that far exceed other approaches. It can find known optimal solutions in a fraction of the time needed by past approaches. It has also been able to find new solutions, being the first approach to solve any 10 team instances that does not consist of constant distances.

The second contribution is from the metaheuristic perspective. The ACO approach contributes a new way of fusing ideas of ACO, which is good at optimization, and constraint processing, which has shown to handle the TTP constraints effectively. It also presents the first use of pattern matching for constraint propagation, which has had a profound effect on reducing the constraint conflicts with the TTP.

The third contribution is from the heuristic search perspective. CIDA* builds on the IDA* algorithm and uses old ideas in new ways. This is also the first approach which does not need to search the whole search tree up to the limiting threshold during an iteration, an important improvement on heuristic search. Another part of this contribution is two new ideas which are more problem-specific. These ideas, forced deepening (FD) and elite paths (EP), help to reduce the node expansion for IDA*. The former, FD, does this by reducing the number of iterations IDA* needed to solve certain combinatorial optimization problems which generally required too many iterations to solve, and the latter, EP, does this by both improving the performance of FD and by quickly finding the optimal solution in the final iteration of IDA*.

1.2 Organization

The layout of this thesis is as follows.

Chapter 2: The Traveling Tournament Problem

Chaper 2 begins by first explaining the sports scheduling terminology used and describes the TTP, the main focus of this work. It also gives an overview of past approaches to the TTP.

Chapters 3 – 5: Ant Colony Optimization

The next three chapters look at our work of applying ACO to the TTP. Chapter 3 explains the background of ACO and introduces a new way of combining ACO with constraint processing.

Chapter 4 first presents past ACO approaches to the TTP and then presents our approach of applying ACO to the TTP. This chapter also introduces pattern matching for constraint propagation.

Chapter 5 presents the results of ACO on the TTP. This involves comparisons with the past ACO approaches along with two leading metaheuristic approaches for the TTP.

Chapters 6 – 8: Concurrent Iterative-Deepening-A*

The following three chapters look at our work of applying CIDA* to the TTP. Chapter 6 explains the background of IDA* and explains our new algorithm, CIDA*. This chapter also looks at two additional new ideas, FD and EP. It ends with an analysis of these ideas using the Traveling Salesman Problem (TSP) as a test case.

Chapter 7 explains the application of CIDA* to the TTP, and also explains the new ideas to reduce the time needed to find optimal solutions when working with said problem. These ideas are the applications of disjoint pattern databases, symmetry breaking, and team ordering.

Chapter 8 presents the results of CIDA* on the TTP. This involves comparisons with past optimal approaches to the TTP and shows new results for problems never solved to optimality before.

Chapter 9: Conclusion

Chapter 9 summarizes the work that has been presented. It also discusses possible future areas of research.

Chapter 2

The Traveling Tournament Problem

Eleven seconds, you've got ten seconds, the countdown's going on right now! Morrow, up to Silk. Five seconds left in the game. Do you believe in miracles? YES!

-Al Michaels

USA vs USSR "Miracle on Ice", 1980 Winter Olympic Games

This chapter looks at the background of sports scheduling. It is split into three sections. The first section explains the sports scheduling terminology used in this thesis. The second section is an overview of the TTP, which is the focus of this thesis. It introduces the problem and the problem sets used in this research, including two new sets created during this research. The third section describes in depth the two areas of research that have focused on the TTP: finding good solutions with artificial intelligence and finding optimal solutions and lower bounds with operations research.

2.1 Round Robin Tournament Nomenclature

Using the nomenclature presented by Rasmussen and Trick [39], a round robin tournament is a tournament structure where every team must play every other team, also

Team	1	2	3	4	5	6
1	-3	-2	-4	+3	+2	+4
2	-4	+1	+3	+4	-1	-3
3	+1	-4	-2	-1	+4	+2
4	+2	+3	+1	-2	-3	-1

Figure 2.1: Example double round robin tournament schedule with even number of teams.

called an *opponent*. The number of times teams play each other can vary. Commonly seen are single round robin tournaments which have the teams playing each other once and double round robin tournaments which have them playing each other twice.

Often associated with every team is a *venue* where they play at. When tournaments use venues, a team playing at their own venue is considered playing *home* and the team visiting another venue is considered playing *away*.

A round robin tournament will consist of a *schedule*, which determines where and when a team will play. A schedule will generally consist of a certain number of columns for all the sets of matches of the tournament and rows for each team playing in the tournament. A column in the schedule is called a *time slot* while a row for a team is called a *tour*. A team can only play once in a given time slot and the length of a tour will be the same length for all teams. An example of a double round robin tournament schedule can be seen in Figure 2.1. Here, a + next to a team number represents a home game and a - represents an away game. As can be seen, when a team t plays another team t' at home, team t' is always playing away against team t in the same time slot, showing the interconnection between tours.

During a tour, when a team plays consecutive games at home, this is considered a *home stand*. When a team plays consecutive away games, this is then considered an *away trip*. This is important for problems where the goal is to minimize distance, as longer away trips can result in less overall distance traveled during a tour.

2.2 Traveling Tournament Problem

The problem that has been looked at in this research is the TTP. The TTP abstracts some of the important features and goals of MLB while leaving out many of the smaller constraints that are a part of MLB. The problem requires a construction of a double round robin tournament and consists of both an optimization goal, in this case minimization of total travel distance, along with constraints that are abstracted from the MLB schedule.

This problem has proven to be incredibly difficult since its inception, while at the same time popular as can be seen by the number of papers on the TTP. What has also helped its popularity is the website that has been maintained by Michael Trick for the TTP, located at `http://mat.gsia.cmu.edu/TOURN/`. The website maintains all problem instances that have been introduced along with the best solutions that have been found to date, making it easy for researchers to compare their work to others' work to see how well their approaches perform.

There are two main reasons for choosing to work on this problem for this research. The first is that even though there have already been many papers published about this problem, there is still plenty of room for new approaches. Due to its difficulty, many problem instances are left unsolved optimally and better solutions are still being found. The second is that the problem's unique structure, having both an optimization goal and feasibility constraints, allows for new ideas that might otherwise be missed if working on problems which have only one of these two aspects.

2.2.1 Problem Description

The TTP is a combinatorial optimization problem which takes in an even set of $n \in 2\mathbb{N}$ teams, $\mathcal{T}$, and a $n \times n$ symmetrical matrix of distances between teams. The objective of the problem is to construct a double round robin tournament which minimizes the total travel distance amongst all the teams while maintaining feasibility with the TTP constraints.

As stated, a double round robin tournament is a round robin tournament with every team playing its opponents twice. A team will play once at home and once away against each opponent. Teams are required to play once during every time slot, with the number of time slots set to $2(n-1)$. Teams also start at home prior to the first time slot and end at home after the final time slot.

The objective is to minimize the total travel distance. Travel distance is first calculated individually for each team, then summed together to get the total distance. Teams only accumulate distance when they are traveling between venues and there is no distance for playing consecutive games at home. Essentially, it is desirable for a team to play as many games as possible consecutively during away trips to help reduce the distance of traveling home between away trips.

More formally, the distance of a team's tour t^d is the summation of distances between consecutive venues, $d_{\pi_i^t,\pi_{i+1}^t}$, found in the tour π^t of team t. Locations of π_0^t and $\pi_{2(n-1)+1}^t$ must be at home. Thus, a tour distance is:

$$t^d = \sum_{i=1}^{2(n-1)+1} d_{\pi_{i-1}^t,\, \pi_i^t}$$

The problem objective is to then find a schedule which meets all feasibility constraints and minimizes the total distance:

$$\sum_{t \in \mathcal{T}} t^d$$

There are two additional constraints for this problem. Both of these constraints have their roots in MLB. The first is the At Most Constraint (AMC). This restricts the number of consecutive home or away games to three. In the MLB schedule, teams play the same opponent between two to four consecutive games. This can be treated as a single set. They will then play between one to three sets consecutively at home or away against different opponents. This helps ensure that a team is not at home for too long or away for too long a stretch at a time.

The second constraint is the No Repeat Constraint (NRC). This requires that a team does not play the same team in consecutive time slots. In MLB, teams will play teams within their division many times throughout the season. It is desirable that a team does not play an opponent for one set at home, then play the same team again in the next set away.

2.2.2 Problem Sets

The TTP is composed of multiple problem instances organized in groups of problem sets. A problem set will have a name, and for each instance in the set, a number will be attached to the name to indicate how many teams are within the instance. For example, NL12 indicates this instance belongs to the NL set, with the particular instance composed of 12 teams.

The first two problem sets were introduced with the original paper [16]. One of these two is NL, which consists of distances between teams found in the National League of MLB. The set ranges in size from the smallest at NL4 up to NL16. The way the teams are set in each instance is that beginning with NL4, all instances are subsets of the next larger instance. Thus, NL4 $\subset$ NL6 $\subset \ldots \subset$ NL14 $\subset$ NL16. This formulation has been used for all real-distance problem sets of the TTP.

The other of the two initial sets is CIRC which uses artificial distances. CIRC has all teams placed on a circle with arcs between neighboring teams. The distance between two teams is the minimal number of arcs that must be passed through. The problem instances range in size from four teams to twenty teams.

The next problem set, CON [47], uses constant distances between all teams. This leads to teams being the same distance from one another, and also leads to a change of problem formulation. Instead of finding minimal distance, the problem objective is finding the maximal number of breaks. A break is a set of two consecutive home games or away games. This has lead to CON being solved for all but CON18 and CON24. The motivation for these instances was for real-life problems where the goal is to reduce the number of trips taken instead of travel distance.

The final problem set is NFL, which is unattributed to any author. This uses the distances between teams in the National Football League, and is considered the larger of the problem sets as the problem ranges in size from 16 to 32 teams.

A variation of the TTP is the mirror TTP [41]. This involves first creating a single round robin tournament, with teams assigned home or away, and then repeating the same schedule but with venues reversed. Thus if a team played an opponent at home the first half, it would play the same opponent at the opponent's venue the second half. All constraints are still required to be met for this problem, so all solutions for a mirrored TTP instance is also a solution for a non-mirrored TTP instance, but not the other way around. The mirrored variant uses the same problem sets as the non-mirrored TTP with one exception. There is a single instance based on the 2003 Brazilian soccer championship composed of 24 teams [41], which has only been worked upon with mirror TTP approaches.

Two new sets have been introduced by us during the span of this research, SUPER and GALAXY. The main purpose of introducing these sets was to increase the number of small team sets available for research. NFL is too large to work with and CON has been too simplified through the problem reformulation.

SUPER

The first problem set created was the SUPER team set. This problem set is based on the Super 14 Rugby League, which is composed of fours teams in Australia, five teams in New Zealand, and five teams in South Africa. The motivation behind using the Super 14 Rugby League was due to its unique geography. Essentially, there are two clusters of teams. The first, larger cluster is composed of teams from Australia and New Zealand. The second cluster, which is located at a very far distance, is the teams from South Africa. This differs greatly from other problem sets, where teams are spread out in one, large cluster.

The problem instances range in size from the smallest at four teams up to the full fourteen team set. For the fourteen team set, the teams are ordered so they alternate between each country. This then leads to each of the smaller instances to have a balanced set of teams from the three countries.

GALAXY

GALAXY is composed of teams based on exoplanets found throughout the galaxy along with the earth. Distances are the number of light years between each of the planets' host star. The motivation behind this problem set is that the stars are located in a 3D plane, with all other problem sets being in a 2D plane.

Exoplanets were picked such that each exoplanet was located in a different constellation. There are two reasons for this. The first is that having them each from a different constellation makes it easier to spread the exoplanets out so they are not all bunched together. Secondly, the constellation names can then be used to identify each team, as many of the stars and exoplanets do not have formal names.

The problem set ranges from the smallest team set of four up to forty teams. This makes GALAXY the largest set available for the TTP, both in terms of the number of instances within the set along with the largest problem size.

2.3 Literary Review of Past Work

To date, much of the work on the TTP has fallen in two camps. Finding good solutions in a reasonable amount of time has been the focus of artificial intelligence while finding provable optimal solutions has been the focus of operations research.

2.3.1 Artificial Intelligence and Metaheuristics

A lot of the success with the TTP has come from the AI community, especially with the usage of metaheuristics to find good solutions in a reasonable amount of time.

The most successful approaches to date for finding good solutions have been with Simulated Annealing (SA). The first of these SA approaches is Traveling Tournament Simulated Annealing (TTSA) [2]. TTSA is a straight-forward application of SA to the TTP with a couple of advanced techniques: strategic oscillation and reheats. The strategic oscillation allowed it to traverse through the solution space of feasible and infeasible solutions. The reheats allowed it to escape local minima in situations where the temperature had become too low such that TTSA was unable to escape from its current local minimum. TTSA separated the problem's constraints into two groups, hard and soft constraints. The hard constraints were those of the double round robin tournament while the soft constraints were the NRC and AMC. All traversed solutions would meet the hard constraints, but at times violate the soft constraints. TTSA then used an objective function which penalized soft constraint violations, and the strategic oscillations helped to control the exploration of feasible and infeasible solutions. This approach also introduced many of the neighborhood moves that have been used by subsequent local search approaches, which allow for a schedule to be modified into another schedule with the double round robin constraints still satisfied. This approach has been one of the strongest single-processor approaches, having improved many of the NL solutions.

Another SA approach [49], which used parallel processing, has found the best solutions for many of the larger problems. This work used the earlier SA approach as a black box, and focused on parallelizing the usage of SA by running multiple SAs in parallel. It used a series of waves of parallel running SAs. At the end of each wave, the best few would keep running while the others would be restarted from the current best solution. They tested their approach from two starting initial solutions: one from the previously known best solution, and one from an initial solution created by TTSA. With both methods, their approach was able to improve many of the best known solutions, including some of the NL instances which had not been improved in awhile.

A third approach using SA [33] combined it with hill climbing. The algorithm would first create an initial solution using a modified three-phase approach which was originally used for a real-league scheduling problem. They improved it by applying beam search to create a stronger initial solution. After the initial solution was created, it was pumped into a controller that would control the flow between the simulated annealing and hill climbing components. The search is split into two, with one looking at the timetable and the other looking at the team assignment. For this paper, the timetable is a pattern that candidate teams can play on, while the assignments reflect the assignment of teams to a pattern. The simulated annealing component, given a fixed assignment of teams, would take a candidate solution and improve upon it by using conditional local jumps to change the timetable. The hill climbing component, after receiving a fixed timetable, would first randomly assign teams to the timetable provided and then modify the team assignments. When this paper was published, it had improved on all CIRC instances greater than 8, and showed comparable results to TTSA for the NL instances.

One of two Tabu Search (TS) approaches, Composite-Neighborhood Tabu Search (CNTS) [12], has been shown to be another strong approach for the TTP. While the application is straight-forward, the authors looked at using various neighborhoods to help strengthen the approach. They introduced neighborhood moves which are similar to the ones used by TTSA, but with some minor modifications. Some key concepts they explored is the overlap of neighborhoods resulting in equivalent moves, since this can cause moves to be performed that return to a previous state, even if the original move was tabu. As they noted, this was not an issue with the SA approaches as SA moves randomly while TS searches through all moves. They also looked at which combination of moves resulted in the best solution quality. They were able to find new results to larger CON instances, some of which were later proven optimal. They were also able to find good quality solutions which were only slightly worse than TTSA, but required a shorter running time to find a solution on average.

The other TS approach was by Lee et al. [32]. In their work, the authors first presented a mathematical model of the problem for an integer programming solver. As this solver could only solve up to four teams, the authors then presented a TS approach. While this work did use some of the same neighborhood searches as done in TTSA, it also used a focusing method of looking for particular aspects of the schedule to be improved. Their best results did not improve on any instances, but did display results competitive with the best results of other approaches.

A third area of metaheuristics has been using ACO. The first ACO approach [10] to the TTP was a direct approach. It used backtracking search for dealing with the hard constraints when constructing solutions. Their approach also used a couple of local improvement techniques. One was searching for additional solutions once a solution had been constructed. This involved backtracking and continuing to build more solutions until a certain number has been obtained. They also combined with a hill-climbing local search for improving the solutions. This approach showed very poor results when compared with other metaheuristics. Many of the problems it suffered were due to the feasibility constraints of the problem. The authors were constructing solutions in a depth-first manner using only backtracking search. Because of this, as the problem size grew, the time needed to find a single solution grew exponentially, resulting in very poor results for larger problem sets. ACO approaches require the construction of many solutions in order to find good solutions.

The next ACO approach used ACO as a hyper-heuristic [6]. Instead of directly applying ACO to the problem, they used it to manage a hyper-heuristic. This also differed in that instead of constructing multiple candidate solutions, it worked with one initial solution and used the hyper-heuristic to perform a local search on the problem. It consisted of a network of vertices which the ants would traverse. Each vertex represented a heuristic to apply to the solution. The ants could revisit a vertex in order to apply the same heuristic multiple times. The pheromone would then relate to how the ants should apply the heuristics. The heuristics used were inspired by the TTSA approach described earlier. This hyper-heuristic approach did improve upon the results of the first ACO approach, but fared poorly when compared with other metaheuristic approaches.

A constructive approach using tiling was done by Bar-Noy and Moody [3]. The motivation of this approach was to be able to quickly create good solutions with tiling. They did this by modeling the road trips as tiles on a team-by-team basis. To create the tiles, they first created a minimum spanning tree for each team, with the team being treated as the root node of the tree. The tiles were then derived from this minimum spanning tree by using a tree collapsing algorithm. Following the tile creation, the tiles were placed on a scheduling grid in such a manner as to minimize distances while meeting constraints. Once no more tiles could fit, the unplaced tiles were broken into individual blocks and the algorithm used backtracking search to rearrange the tiles in order to find more solutions. The authors achieved their goal of being able to create schedules within 5-7% of the best known solutions for NL8 and NL10, but their approach was unable to create a schedule for NL16.

Another constructive approach was done by Kendall et al. [28] which used a two-stage approach. The first stage involved treating the TTP as a Traveling Salesman Problem and finding an optimal tour through the teams. The second stage involved using a heuristic that modified the optimal tour found in the first stage by adding home and away edges, which was done separately for each team. This process was done with patterns, with each individual pattern representing a combination of groupings of consecutive away games split up by single home games such that the AMC was satisfied. These patterns ignored the number of home games, since they have no impact on travel distance for a team. When applying a pattern to the optimal tour, it would break the tour where the designated team would travel home between sets of away games as indicated by the pattern. The final step of the process was combining patterns which led to good quality solutions, and then using local search to improve on the solutions. No results were presented in this paper, as the authors stated the results would be presented at the respective conference where this paper was published.

A metaheuristic approach for the mirrored TTP variant was the work by Ribeiro and Urratia [41], which combined Greedy Randomized Adaptive Search Procedure and Iterated Local Search into a algorithm called GRILS. They first presented a fast constructive method to create initial solutions of good quality. This involved a three-stage process, which began with a single round robin tournament and by the third stage had resulted in a full mirrored double round robin tournament. The motivation for creating these good solutions was that good initial solutions can be beneficial for certain metaheuristics. They also presented some new neighborhoods for local search, some which are similar to those presented for SA and TS along with a unique one that builds on ejection chains. Their final contribution was GRILS, which encompassed all their other concepts. This was applied to the mirrored TTP and was found to be good at finding solutions for the mirrored TTP and even some of the mirrored solutions turned out to be new best solutions for non-mirrored CIRC instances.

2.3.2 Operations Research, Exact Algorithms, and Lower Bounds

In terms of finding optimal solutions, operation research has been at the forefront of this area. Some of the papers that have been published have been for finding solutions, while a few have been for finding lower bounds to optimal solutions. The latter is due to the difficulty of this problem, as it is very difficult to find optimal solutions for problem instances larger than eight teams. For finding optimal solutions, some of the approaches have been using Integer and Constraint Programing, Lagrange Relaxations, and Branch-and-Price with Column Generation.

The first work in finding optimal solutions was by Easton et al. [16]. In this work, they used a relaxation of the problem called the independent lower bound (ILB), which they used to find optimal solutions. The ILB is a lower bound estimate which is calculated as the sum of the optimal distance of each individual team, with the individual distances calculated independently of the other teams' constraints. Optimal solutions were found by incorporating the ILB into a pattern generation and matching approach. The ILB was used as a lower bound to determine a cutoff of which patterns could be discarded without losing guarantee of optimality. They were able to successfully find optimal solutions for NL4, NL6, CIRC4, and CIRC6. In addition, they also found lower bounds to NL8, NL16, and CIRC8.

Later on, the same authors published another paper for an exact method [17]. Here they used a Branch-and-Price algorithm which combined Integer Programming and Constraint Programming. Their motivation for combining the two was based on the history of these two techniques. Integer Programming had been successful with optimization problems like the TSP and vehicle routing problem. Constraint Programming had been successful at finding complex home and away pattern constraints in sports scheduling. Thus by combining the two techniques, they had hoped to take the strengths of both approaches in order to be able to solve TTP instances to optimality. These two techniques were then integrated into a Branch-and-Price algorithm, with the Integer Programing responsible for solving the master problem and the Constraint Programming responsible for solving the pricing problem. They further improved this by designing a parallel approach. When applied to the TTP, they were able to solve NL8, but only when it was relaxed as their work did not respect the NRC.

Another approach for finding optimal solutions was an approach that used Lagrange Relaxation techniques with Constraint Programming [5]. In their work, they first discuss different models for Lagrange Relaxation. This is an important component of their approach, which can be seen as a hierarchy of Constraint Programming models and sub-problem solvers. They also presented improvements to their model, allowing for better lower and upper bounds to be found. In the end, their work was able to prove optimality for NL4 and NL6, but was unable to compete with the work by Easton et al. [16] in terms of finding better lower or upper bounds to other problem instances.

The most successful approach to date in finding optimal solutions is a Branch-and-Price with Column Generation approach by Irnich [25]. This approach incorporated a new formulation of the problem, treating the tour of each team as a time-discrete network. Their work then built on the work of Easton et al. [17] for designing the program to solve this formulation, but included the NRC which had been omitted by the earlier work. One of the novel ideas presented, which is pertinent to this work, is the ability to break symmetry in the TTP to reduce the solution space. We also cite an earlier version of this work [26], which presented an additional symmetry for CIRC instances which they did not include in the current work. The end results of this approach was that it was the first to solve CIRC6 and NL8 with all constraints satisfied, and was able to find new lower bounds for NL10, NL12, and CIRC8.

The work by Cheung [7] looked at finding optimal solutions for the mirrored NL8 and CIRC8 instances. Their approach was a two-phase approach. The first phase involved creating all permutations of one-factorizations for single round robin schedules, which can be used to create the mirrored TTP instances. The second phase involved going through each one-factorization found in phase one and trying to find the minimal timetable and pattern set which gives the minimal travel distance for that one-factorization. Overall, this approach did find the optimal solutions for mirrored NL8 and CIRC8, taking over three days for each instance.

Another work by Cheung [8] looked at finding lower bounds for mirrored TTP instances, with focus on the NL and NFL sets. They accomplished this by using a Benders decomposition approach, which was integrated into a mixed-integer linear programming approach for calculating the minimum number of trips lower bound [46]. This approach was able to improve the lower bounds for the mirrored NL and NFL instances up to the size of 24 teams, but was unable to calculate any for NFL26 or larger instances due to memory limitations.

Another approach for finding lower bounds was the work by Urratia and Ribeiro [47] which focused on finding lower and upper bounds for round robin tournaments and the mirrored CONS instances. Their work explored the connection between break maximization and distance minimization, showing that these two are equivalent on the CONS instances. They showed how to construct solutions for certain classes of mirrored CONS instances resulting in optimal solutions, which is proven with the previous GRILS-mTTP heuristic finding upper bounds matching their lower bounds. They were able to solve all mirrored CONS instances up to size 16, which were the largest instances solved at the time of the writing of their work.

Yet another approach for finding lower bounds was a work which looked at improving upon the ILB [46]. The authors created a new lower bound called minimum number of trips lower bound. This was achieved by using the optimal solutions found for CONS instances, or lower bounds if none had been found, to check whether the ILB creates an estimate which consists of too few trips. The reason they were able to do this is that the best solution for a CONS instance of equivalent size will indicate the minimum number of total trips the teams must make in order to create a feasible solution. If the ILB does consist of too few trips, their approach will then increase this lower bound by finding the minimal estimate with regards to the minimal number of trips needed. Using this new lower bound technique, they were able to find new lower bounds to many problem instances that had yet to be solved, including being the first to find lower bounds for the NFL instances of size 16 to 22.

The work by Fujiwara et al. [19] looked at both finding lower bounds for the CONS instances for the full TTP and also presented two algorithms to produce feasible solutions for the same instances. Their approach for finding the lower bounds was a simple calculation based on the number of teams present in the instance, which they proved to be a feasible lower bound for these instances. Along with this method for calculating the lower bounds, they presented two constructive algorithms. These algorithms essentially worked by first constructing single round robin tournaments, then transforming these tournaments into double round robin tournaments. Their work was shown to be effective, with the gap between their lower bounds and best solutions found by their algorithms to be small even for up to 50 teams. For some instances, they were able to prove optimality if certain criteria were met, including finding an optimal solution for a 46-team instance.

Chapter 3

Ant Colony Optimization

This was Galileo's great point which started the modern scientific revolution – look at Nature not in books!

-Richard Hamming

The Art of Doing Science and Engineering

This chapter introduces the ACO metaheuristic, the basis of the first of two approaches for the TTP. It also describes the Foward Checking and Conflicted-Directed Backjumping (FC-CBJ) algorithm [37] for constraint processing, and then shows how it is integrated with ACO to create an ACO algorithm for optimization problems which require constraint processing.

The reason why ACO is being considered for the TTP is because its past performance for this problem has had poor results, especially when compared to other metaheuristics. This is surprising since it has had better performance when applied to similar problems. This work looks at the reasons of why it performed poorly for the TTP and what can be done to create a strong ACO algorithm for problems like the TTP.

3.1 Ant Colony Optimization Metaheuristic

ACO is a metaheuristic inspired by the ability of ants to find the shortest path between a food source and their nest. The basic premise of the metaheuristic is a colony of ants are going through cycles of candidate solution construction and pheromone updating, with the pheromone influencing the actions of the ants. ACO differs from metaheuristics like TS and SA in that it is a constructive metaheuristic. It traverses through the solution space by creating many solutions during the running of the algorithm. TS and SA are local search approaches in that they will generally work with one solution and traverse through the solution space by modifying the solution repeatedly.

Algorithm 1 briefly describes the ACO metaheuristic as a whole, with the various aspects being described in more detail throughout this chapter. At the beginning, the pheromone matrix is initialized to a constant value and the set of ants are initialized. It then enters a cyclic process until some pre-specified ending condition is met. During the cyclic process, each ant in the colony will create a candidate solution. This is done by choosing each component of the solution using a proportional-random method, which combines heuristic information and the pheromone matrix. After all solutions have been created, the algorithm will then try to improve the candidate solutions with local search. This part is optional and is not used for all applications of ACO. The third part of the cyclic process is updating the pheromone matrix based on the constructed solutions. In the end, once the cyclic process is finished, the algorithm will return the best solution found.

Algorithm 1 Ant Colony Optimization general algorithm

```
1: procedure ACO
2:     Initialize Pheromone
3:     Initialize Colony of Ants
4:     while End Condition Not Satisfied do
5:         Construct Set of Solutions
6:         Perform Local Search
7:         Update Pheromone
8:     return Best Solution
```

3.1.1 Colony of Ants

As with its real life counterpart, ACO often uses a colony of ants. Each ant is responsible for constructing a candidate solution during every cycle.

The number of ants in the colony varies per application, with some applications relating the number of ants to the problem size while others use a set number regardless of problem size. Generally though, having too few ants results in not enough solutions being constructed each cycle, which decreases the chance of the ants being able to fully exploit the pheromone before it changes for the next cycle. Having too many ants can lead to not enough pheromone updates (i.e. cycles) during the running of the algorithm, again leading to poor results.

There are two ways ants can construct solutions for each cycle. This can be done either sequentially, with each ant constructing a full solution prior to the next ant starting, or in parallel, with all ants taking turns adding a component to their solution. In the case of the latter, each ant's solution is still independent of the other ants solutions. For most applications, either form of construction will lead to the same results. The exception is in the case of using local pheromone updates as done with ant colony system (ACS) [13]. Local pheromone updates causes the pheromone matrix to change after every step of the construction solution process, thus having the ants go in sequential or parallel order can impact the solution quality. Local pheromone updates are further explained in Section 3.1.5.

3.1.2 Pheromone

The key artifact of ACO is the pheromone matrix τ, with the rows and columns indicating the pheromone value between two components. This influences the ants' decisions when constructing solutions, and without it the algorithm would essentially be performing a pseudo-random search.

The pheromone represents the desirability of an ant making a certain choice. Using the TSP as an example, assume an ant is currently at a city. When deciding the next city to visit, the ant will take into account any heuristic information, as explained in Section 3.1.3, along with the pheromone. The pheromone will have higher values for cities which are a good choice in past solutions and low values for cities that have either not been connected before with the current city or have led to poor solutions. The pheromone will help direct the ant to make a better choice, even if such a choice contradicts the heuristic information.

The pheromone is updated at the end of each cycle. How it is updated varies with different implementations. Some implementations will use all ants and weigh each ant's contribution to the pheromone upon the quality of their solution compared to the other ants. Other implementations will use an elitist approach, where only one ant is used to update the pheromone matrix. In the second case, there are three types of ants that can be used for pheromone updating: iterative best ant, global best ant, and restart best ant. The iterative ant is the best ant seen during the current cycle. The global best ant is the best seen ant since the beginning of the running of the algorithm. The restart best ant is the best ant seen since a pheromone reinitialization, which is explained in more detail in Section 3.1.5.

Another important feature of the pheromone matrix is pheromone evaporation. At the end of each cycle, some of the pheromone is evaporated prior to being reinforced with new solutions. This is done using a decay rate of ρ in the range of [0..1], with larger values resulting in a slower decay. The purpose of the evaporation is to help the ants forget past solutions, many of which are poor in the beginning. This allows for more exploration, especially as newer, better solutions are being found. If the pheromone does not evaporate, then poor choices in the past will still have influence on the current solution construction. But with the evaporation, poor choices will slowly be forgotten while choices which are used for good solutions will be reinforced.

Combining the ideas of ants applying pheromone and pheromone evaporation are the keys to pheromone updates. While different applications will use different detailed approaches for this, the one which concerns this work is the one used by ant system (AS) [14] using an elitist approach. We will explain this with the TSP example, as that is how it is commonly described in the literature [15]. Assuming the pheromone matrix is two dimensional and the problem involves a minimization goal, the pheromone update is then defined as:

$$\tau_{ij} \leftarrow \rho \cdot \tau_{ij} + \frac{1}{f} \tag{3.1}$$

where f is the cost of the solution of the ant being applied which used the arc i, j in its solution, or:

$$\tau_{ij} \leftarrow \rho \cdot \tau_{ij} \tag{3.2}$$

if the arc i, j is not used, which results in only the pheromone being evaporated.

3.1.3 Solution Construction

While constructing solutions is problem dependent, most approaches share common features. Many applications use a proportional-random method which uses both pheromone and heuristic information about the problem when choosing values. The purpose of the proportional-random method is to create a pseudo-random choice for the ant. This allows the ants to reuse past information, while injecting randomness in the solution construction process which encourages exploration of the solution space.

Heuristic information, η, is information gained from the problem to help make a choice. For example, with the TSP, the heuristic values are the distances between cities. Thus the heuristic value between cities i and j is then defined as:

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{3.3}$$

with d_{ij} being the cost of the path going from city i to city j.

There is no predefined proportional-random method for picking values designed for ACO, as different implementations use different systems. The two common rules relevant to this work are AS's random proportional rule and ACS's pseudorandom proportional rule.

Continuing with the example of applying ACO to the TSP, let i be the current city the ant is at and $\mathcal{D}$ be the set of cities the ant still has to visit. AS's random proportional rule is then defined as:

$$p_{ij} = \begin{cases} \frac{[\eta_{ij}^{\alpha}\tau_{ij}^{\beta}]}{\sum_{k \in \text{allowed}_{\mathcal{D}}}[\eta_{ik}^{\alpha}\tau_{ik}^{\beta}]} & \text{if } j \in \mathcal{D} \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

where p_{ij} is the probability of choosing city j following city i. There are also two scalars involved, α and β with both having positive real values, which help determine how strong each of the two elements are. Having a larger α value results in the heuristic values having more influence while having a higher β value results in the pheromone values having more influence.

Using similar notation, ACS's rule is defined as:

$$s = \begin{cases} \text{argmax}_{j \in \mathcal{D}}\{[\eta_{ij}^{\alpha}\tau_{ij}^{\beta}]\}, & \text{if } q \leq q_0 \\ S, & \text{otherwise.} \end{cases} \tag{3.5}$$

where s is the resulting choice, S is the random proportional value p_{ij} chosen with Equation 3.4, q is a random value in [0–1], and q_0 is a constant in [0–1].

The general difference between these two rules is that AS's rule is more exploratory compared to ACS's rule, especially if q_0 is set to a high value. These rules can have different impacts for different problems, with some problems benefiting from more exploration while others benefiting from more exploitation.

3.1.4 Local Search

Local search can help to improve the constructed solutions after each cycle and has been used for many applications of ACO [15]. It is usually applied for a short period of time, such as until the solution cannot be improved any further. While the local search will have an associated overhead, this is often mitigated by the improvement in solution quality, allowing for better solutions to be found faster and giving the pheromone updates better solutions with which to update the pheromone.

An example of this is again applying ACO to the TSP. A local search procedure that can be used is 2-opt, which goes through the solution and swaps any two cities which would lead to a better solution. The local search is applied to all ants after they have constructed their solutions, and run for a specified amount of time. This will often then result in improved candidate solutions and helps the algorithm converge faster.

The local search can be seen as complementary to the workings of ACO. The ants are constructing new solutions, or starting points, to be fed into the local search. The local search is then improving these solutions which are used to update the pheromone and help create even better new solutions.

3.1.5 Algorithms

While there are many algorithms that fall under ACO, there are two which are relevant to this work. The first is $\mathcal{MAX}-\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) [44] and the second is ACS.

$\mathcal{MAX}-\mathcal{MIN}$ Ant System

$\mathcal{MM}$AS builds off the original ACO algorithm AS. AS introduced the core concepts, such as the pheromone matrix and the random proportional rule. It was applied to the TSP, but suffered problems of stagnation. $\mathcal{MM}$AS improved on this by introducing new concepts, most importantly the pheromone limits which set maximum and minimum limits that the pheromone values can take. These pheromone limits are fluid in that they change as better solutions are found. They help to reduce stagnation since the limits make sure there are no pheromone values which dominate the choosing of values. The upper limit τ_{max} is defined as:

$$\frac{1}{1-\rho}\frac{1}{f(s^{gb})}$$

with $f(s^{gb})$ the cost of the best seen solution, s^{gb}, since the algorithm began. Using the same reasoning for τ_{min} when $\mathcal{MM}$AS is applied to the TSP [44], τ_{min} is set to:

$$\tau_{max} \frac{(1 - \sqrt[n]{p_{best}})}{((avg - 1) \cdot \sqrt[n]{p_{best}})}$$

where avg is the average number of choices an ant will see, n is the size of the problem instance, and p_{best} is a variable in the range of [0..1].

Another difference between $\mathcal{MM}$AS and AS is that pheromone is initialized to the maximal pheromone value instead of an estimate of the solution cost. The purpose of this is to encourage more exploration at the beginning. Having the pheromone values start at a higher value instead of a lower value causes a smaller difference in pheromone values through pheromone evaporation instead of through pheromone reinforcement, and the smaller difference helps reduce the impact of the pheromone at the beginning of the algorithm. This then leads to greater exploration of the solution space.

$\mathcal{MM}$AS introduced another concept called pheromone reinitialization. After certain conditions are met, such as a certain number of cycles with no improvements, the pheromone is reinitialized to the initial pheromone value. This allows the algorithm to start fresh after it has begun to stagnate or get stuck in a local minimum.

Ant Colony System

ACS is less related to AS, having introduced its own choice rule along with different ideas to deal with stagnation. Its choice rule was more aggressive than the rule used by AS and the aggressiveness of this rule was complemented by the design of how the pheromone is changed during the running of ACS.

ACS first used a different way of doing pheromone updates. Instead of applying pheromone updates to the whole pheromone matrix, only values which were used by the ant updating the pheromone are changed. This includes the pheromone evaporation. Second, during solution construction, the algorithm will perform a local pheromone update after every choice made by an ant. This is to force future ants within the same cycle to take different paths. Since the first ant will probably take the path on the pheromone with the highest values, the local pheromone updates will lower these pheromone values so the path becomes less appealing for the next ant.

3.2 Constraint Processing

The FC-CBJ algorithm is for solving constraint satisfaction problems, which are problems that possess constraints which must be met in order for a solution to be feasible. FC-CBJ is a depth-first search algorithm which allows the combination of both looking ahead to future assignments and looking back at past assignments for constraint satisfaction, combining the best of both aspects. Using Dechter's nomenclature [11], constraint satisfaction deals with three sets: the set of variables X, their associated domains D, and the set of constraints C.

3.2.1 Looking Ahead

The looking ahead, specifically forward checking (FC), allows an algorithm to propagate constraints related to an assignment of a value. This then allows an algorithm to know ahead of time if the current assignment is infeasible or not.

During the depth-first search, once a value has been selected for a variable $x \in X$, the algorithm will then propagate constraints for all unassigned variables and their associated domains. For each unassigned variable $u \in X$, the algorithm will go through all values in u's domain, D_u, and check that the value can be assigned with regards to the constraints in C which relate to x and u. If the current assignment of x causes a constraint conflict with a candidate value in D_u, then that candidate value will be removed from D_u, otherwise the algorithm continues on with checking the next value or unassigned variable.

If the assignment were to cause a domain to become empty, then the algorithm knows that the current assignment cannot lead to a feasible solution. It will undo the assignment and try a different assignment if the current domain is not empty.

Otherwise, it will have to backtrack to a previous variable and try a different value assignment for that variable.

3.2.2 Looking Back

The looking backwards, specifically conflict-directed backjumping (CBJ), allows an algorithm to jump further back than would be possible with backtracking. Regular backtracking allows an algorithm to take one step backwards to the previous assignment while backjumping allows an algorithm to jump back even further than possible with backtracking. In order for conflict-directed backjumping to work, it uses a conflict set J for each variable.

Conflict-directed backjumping allows the algorithm to jump back to the most recent assignment which caused the current conflict. This is in regards to the constraints. During the value selection process for variable $x \in X$, after a candidate value has been chosen, the algorithm will go through previously assigned variables and check to make sure the value is consistent with the constraints. If it is, then the algorithm will be able to move forward to the next variable. If it is not, then the algorithm will add the most recent variable which conflicts with x to J_x.

When a variable x has exhausted its domain, it needs to move backwards. With conflict-directed backjumping, the algorithm will pick the most recently assigned variable in J_x and jump back to that variable. If J_x is empty, then this indicates there are no feasible solutions.

The reason this allows for all feasible solutions to be found is that the algorithm is jumping over variables whose assignment has had no impact on the values of x. Changing their values wont allow for x to be assigned a value which does not conflict with any past assignments. Thus by jumping back to the most recent variable whose assignment conflicts with x, the algorithm is then able to save multiple backtracks and try a value for a variable which may allow x to be assigned a feasible value.

3.2.3 Forward Checking and Conflicted-Directed Backjumping

Combining the two past approaches results in the FC-CBJ algorithm. The two aspects combined can be seen as compliments of each other, since the forward checking helps to build the conflict sets that are used for the conflicted-directing backjumping.

When performing the forward checking for a variable x, the algorithm will update the conflict sets for the future, unassigned variables. This happens when a current assignment causes a value to be removed from an unassigned variable's domain D_u. That unassigned variable will then add x to its conflict set. When undoing an assignment, the changes made to all conflict sets caused by the assignment are undone, thus the conflict set will always reflect the current state of assignments.

Since the conflict sets are created in the past, when the algorithm needs to move backwards, it looks at its conflict set to find the most recent variable to jump back to. This conflict set allows FC-CBJ to use the conflict-directed backjumping as described earlier. And as with before, if the current variable's conflict set is empty and the algorithm needs to backjump, this then indicates that there is no feasible assignment.

3.3 Integrating ACO with Forward Checking and Conflicted-Directed Backjumping

We integrate ant colony optimization with forward checking and conflict-directed backjumping (AFC) to create an ACO approach for constrained combinatorial optimization problems. All the changes made for its integration into ACO have been in the solution construction process, as FC-CBJ has no impact on the pheromone. New aspects for ACO are the usage of variable domains, constraint propagation, and backjumping. Other new ideas are the usage of unsafe backjumping and a new approach of applying ant restarts when too many backjumps have taken place.

3.3.1 Relation To Past Work

The motivation for this approach is from the difficulties of applying ACO to the TTP. As mentioned before, Crauwels and Van Oudheusden[10] have done the only direct approach of applying ACO to the TTP and it showed poor results. The difficulties lay with their approach's reliance on only backtracking search. The constraints of the problem caused it to go through too many backtracks, which caused the algorithm to spend too much time constructing a single solution. This in turn leads to fewer solutions being constructed during the course of running the algorithm and very few cycles, which limits the time available for the algorithm to explore the solution space and leads to poor results in the end.

This new work builds off two past approaches of combining ACO with constraint processing techniques. The first is by Meyer and Ernst [34], which combined ACO with constraint programming to allow the constraints of the problem to be propagated. They had taken ACS as the base algorithm, and then modified the solution construction to integrate the CP for whenever a value was chosen. They applied this approach to a machine scheduling with sequence-dependent setup time problem. It had been found to work better in terms of solution quality and failure rate than an ACS approach which did not use constraint programming.

The second approach is the work by Uthus et al. [48], which combined ACO with backjumping search. This work was for the application of ACO to an easier round robin tournament problem. The work showed how ACO integrated with backjumping could construct solutions faster than when integrating only with backtracking search.

AFC can then be seen as the combination of both past works, looking forward with constraint propagation and moving backwards with backjumping.

3.3.2 AFC Description

Algorithm 2 describes the solution construction procedure of AFC. It follows the pseudocode as described by Dechter [11].

The algorithm takes in a set of variables X, their associated domains D, and the set of constraints C. It begins by first initializing the domains and conflicts sets along with choosing the first variable. The order in which variables are chosen is problem dependent. Following the preliminary steps, the algorithm enters the constructive phase. It will first try to select a value for the current variable using the select value method of FC-CBJ. The method has been modified so that values are chosen using the pseudo-random nature of ACO and is described further in Section 3.3.3.

Algorithm 2 AFC

```
1:  procedure ConstructSolution($X, D, C$)
2:      $i \leftarrow 1$
3:      SelectVariable
4:      $D'_i \leftarrow D_i$ for $1 \leq i \leq n$ // Initialize domains
5:      $J_i \leftarrow \emptyset$ for $1 \leq i \leq n$ // Initialize conflict sets
6:      while $1 \leq i \leq n$ do
7:          $x_i \leftarrow$ SelectValueACO
8:          if $x_i =$ null $\wedge$ #backjumps = limit then
9:              RestartAnt
10:         else if $x_i =$ null then
11:             $i' \leftarrow i$
12:             $i \leftarrow$ latest index in $J_i$
13:             if SafeBackjump then
14:                 $J_i \leftarrow J_i \cup J_{i'} - \{x_i\}$
15:             Undo changes to $D'$ and $J$ from $i'$ to $i$
16:         else
17:             $i \leftarrow i + 1$
18:             SelectVariable
19:     return $X$
```

If the algorithm is unable to find a value, which is when the domain becomes empty, then one of two things will happen. If it has already reached the limits of the number of backjumps for this current solution construction, it will then restart the ant as described in Section 3.3.5. If not, it will perform a backjump. If using safe backjumping, this process will then be the same as regular FC-CBJ. Otherwise, it will perform unsafe backjumping, described in Section 3.3.4.

If the algorithm is able to find a value, it will then progress to the next variable. Once it has reached n, the number of variables, the algorithm knows it is finished and returns the assignment X.

We note that the algorithm omits two lines from the original work that would have caused the algorithm to work incorrectly in its original presentation. These two lines take place after Line 18 and they would have both caused the domains to become inconsistent and for the conflict sets to be reset when they should not be.

This algorithm is designed for problems which have large, feasible solution spaces. If the feasible solution space is small, we would expect the algorithm to perform poorly. Additionally, if ant restarts are used and there are no feasible solutions, this algorithm would then enter an infinite loop. For problems with no feasible solutions, such as when the goal is to minimize constraint violations, it is then better to use an ACO algorithm designed for such a problem [43].

3.3.3 Value Selection

Choosing values for AFC requires most of the changes to the integrated algorithm. With ACO, all that is needed is to use ACO's proportional-random method to choose a value. But with AFC, there is additional work once a value is chosen. This is described in Algorithm 3. The algorithm takes in the current index of the solution construction process, and will return either a valid value or null if there are no feasible values.

The value selection algorithm begins by first choosing a candidate value a using ACO's proportional-random method. It then removes it from the variable's domain and begins the constraint propagation process. It will look at all values of all domains of unassigned variables and check that the value a does not conflict with any other values based on the constraints of the problem. If a value b does conflict with a, then b is removed from the associated domain. In addition, the current index in the depth first search is put into the future conflict set of the unassigned variable.

After all values have been checked, one of two things happens. If no domains became empty by assigning a, then a is returned. If a domain did become empty, then the algorithm undoes all changes made to future domains by assigning a. It will then try a different value if the current domain is not empty, or it will have to return

Algorithm 3 Value Selection

```
procedure SelectValueACO($i$)
    while $D'_i \neq \emptyset$ do
        $a$ = Value chosen with ACO's proportional-random method
        $D'_i \leftarrow D'_i \backslash a$
        $emptydomain \leftarrow false$
        for all $k, i < k \leq n$ do
            for all $b \in D'_k$ do
                if $a$ not consistent with value $b$ then
                    $J_k \leftarrow J_k \cup i$
                    $D'_k \leftarrow D'_k \backslash b$
            if $D'_k = \emptyset$ then
                $emptydomain \leftarrow true$
        if $emptydomain$ then
            Reset changes to $D'_k$ and $J_k, i < k \leq n$
        else
            return $a$
    return null
```

null.

3.3.4 Unsafe Backjumping

A backjumping algorithm is considered safe if it does not jump far enough back such that any solutions would be missed [11]. This is not a concern with ACO, as it is only trying to create a feasible solution. A new idea looked at is using unsafe backjumping to help reduce the overall number of backjumps. This unsafe backjumping is designed for problems which have a large solution space, like the TTP. It would be a hindrance for problems which are heavily constrained and have a small solution space.

When compared with safe backjumping, unsafe backjumping as described here will sometimes jump back far enough that some solutions may be missed. This happens when the algorithm first jumps back from index i to index j, and then makes a second backjump from index j to another index further back. Unsafe backjumping can jump back further than it needs to, allowing the algorithm to get out of an unfeasible partial solution faster. This then allows it to continue constructing from a feasible solution sooner, reducing the time needed to find a feasible solution. Again, this is only practical if the feasible solution space is large. For a small feasible solution space, then this might cause the algorithm to miss the only possible feasible solutions.

Unsafe backjumping is done by omitting line 14 of Algorithm 2. When this happens, the algorithm will no longer combine conflict sets when jumping back, as required by conflict-directed backjumping [11]. The computing of the union of the conflict sets prevents the algorithm from jumping over any assignments which, if changed, could allow the algorithm to find a feasible solution. Unsafe backjumping can then be seen as a backjumping form which will sometimes ignore assignments when jumping backwards.

3.3.5 Ant Restarts

Another feature we have added to AFC is ant restarts. Once the algorithm has gone through too many backjumps, the algorithm will restart the ant's solution construction process. This is useful for problems in which it is difficult to propagate all or some of the constraints along with being general enough that allowing one to avoid having to hardcode for every situation of constraint conflict. They are intended to give the algorithm sufficient time to construct a solution, but put a limit on the number of backjumps on cases where too many backjumps take place.

The limit on the number of backjumps prior to an ant restart is set to $b \cdot n$, with b being a scalar and n the number of variables. Once this limit has been reached, the algorithm will then restart the ant at index 1 and reset the domains and variables along with the backjump counter.

Ant restarts are similar in nature to other works by Meyer and Ernst[34] and by Beck[4]. With the first of the two, when they combined ACO with constraint propagation, they implemented an idea called single level backtracking, which limited how far back the algorithm could backtrack. Should a domain become empty after backtracking for that domain, then the algorithm would use a death penalty and restart the process. With Beck's approach, which was an originally designed constructive search algorithm, they would restart after a certain number of backtracks had taken place. How this differs from the ant restarts here is that they used a dynamic restart policy along with a short restart schedule.

Chapter 4

Ant Colony Optimization and the Traveling Tournament Problem

Ants are so much like human beings as to be an embarrassment. They farm fungi, raise aphids as livestock, launch armies into war, use chemical sprays to alarm and confuse enemies, capture slaves, engage in child labor, exchange information ceaselessly. They do everything but watch television.

-Lewis Thomas

This chapter shows the application of AFC to the TTP (AFC-TTP). It also introduces the new idea of how to use pattern matching for constraint propagation.

4.1 Applying AFC to the TTP

Our approach of applying AFC to the TTP uses a combination of ACS and $\mathcal{MM}$AS. From ACS, we use its pseudorandom proportional rule for choosing values. We do not use the local pheromone updates due to the overhead that would arise from the backtracking. From $\mathcal{MM}$AS, we use its pheromone limits and reinitialization.

4.1.1 Variables and Values

In AFC-TTP, teams are treated as both variables and values. Each team will have a domain associated with every time slot. Within the domain will be all the opponents of the team which can be chosen for this time slot, both for games at home and games away.

AFC-TTP constructs solutions in a first-to-last time slot order, similar to solution construction by the first ACO approach for this problem [10]. Beginning with the first time slot, it will pair all teams for the time slot prior to moving to the subsequent time slot. The reasoning behind this is that it is easier to propagate constraints and enables the usage of pattern matching, as described in Section 4.2.

When first choosing the variable, AFC-TTP uses a dynamic variable ordering [11]. It picks the first team, t_1, of the current time slot by looking for the team which has the least amount of teams left in its domain, which is essentially a first-fail strategy. In the cases where there is a tie, then it will pick the team in numerical order starting from the ant's number. Each ant will have an associated number. If the number of ants and teams is equivalent, then each ant will have a distinct number. If the number of ants differs from the number of teams, then each ant's number will be random in the range of $[1..n]$ for each cycle, where n is the number of teams. This numbering of ants and choosing tied teams in numerical order from the ant's number helps to ensure the search space is being adequately searched and that no team is being favored. This is similar to both the first approach of applying ACO to the TTP [10] and also when ACS is applied to the TSP [13]. In that application, ACS will choose a random city to begin constructing the tour from for each ant.

After choosing the first team as the variable, AFC will then have to go through t_1's domain and choose a second team, t_2, as the value. When applying AFC to the TTP, AS's rule is defined as:

$$p_{ijk} = \begin{cases} \frac{[\eta_{ijk}^{\alpha}\tau_{ijk}^{\beta}]}{\sum_{k \in \text{allowed}_k}[\eta_{ijk}^{\alpha}\tau_{ijk}^{\beta}]} & \text{if } j \in D_{ik} \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

while ACS's rule is defined as:

$$s = \begin{cases} \text{argmax}_{j \in D_{ik}}\{[\eta_{ijk}^{\alpha}\tau_{ijk}^{\beta}]\}, & \text{if } q \leq q_0 \\ S, & \text{otherwise.} \end{cases} \tag{4.2}$$

with the pheromone value τ_{ijk} being the desirability of team i playing at home against team j during time slot k and the heuristic information η_{ijk} being set to $(d_{ijk})^{-1}$, with d_{ijk} representing the distance added to the partial schedule with the assignment of i at home against j during time slot k. We define AS's rule here since it is used within the ACS rule.

After choosing a candidate pairing of teams, constraint propagation takes place. The algorithm propagates the constraints to make sure the two teams do not play again during that time slot, that the pairing of the two in regards to which is home and away does not repeat in future time slots, and that for the following time slot the AMC and NRC are not violated. Also propagated is the pattern matching, as described in Section 4.2.

4.1.2 Pheromone

As stated before, pheromone is the key concept of ACO. For AFC-TTP, pheromone is defined as τ_{ijk}, which represents the desirability of team i playing at home against team j during time slot k.

When taken into context of the AFC-TTP approach for choosing variables and values, it will go through both the pheromone values when t_1 is the home team, treating it as team i in τ_{ijk}, and when t_1 is the away team, treating it as team j in τ_{ijk}. Thus, it checks for all possibilities of t_1 playing at home or away for time slot k when choosing a team to pair with t_1.

AFC-TTP will use different ants for different cycles when updating the pheromone matrix. For this approach, prior to pheromone reinitialization, AFC-TTP alternates between the iterative best ant and the global best ant. After a pheromone reinitialization, AFC-TTP alternates between the iterative best ant and the restart best ant.

This approach for pheromone updates differs from other ACO approaches, as it is more common to use the global best ant throughout the running of the algorithm. As seen later in Section 5.1.3, this was found to be the best approach for AFC-TTP.

4.1.3 Local Search

AFC-TTP uses a general TS approach [20] for the local search updates. As mentioned earlier, TS is a local search metaheuristic. It traverses through the solution space by changing a candidate solution through picking the best possible move from its neighborhood. In some cases, this will result in a move which worsens the solution quality, such as in a case when it is in a local minimum. To help prevent it from cycling, the metaheuristic employs a tabu list. This list keeps track of a certain number of previous neighborhoods, which are considered tabu and cannot be returned to. By doing so, TS is then able to escape from local minimum and search the solution space.

This TS implementation uses the neighborhood definitions described in the earlier SA approach called TTSA [2], see Section 2.3.1. These neighborhoods involve either swapping the home/away locations between a pair of teams, swapping the whole tours of two teams, partially swapping the tours of two teams, swapping all of two time slots, or partially swapping two time slots. When using these neighborhood moves, it takes into account the neighborhood overlaps as described by the earlier TS approach CNTS [12], see Section 2.3.1. TTSA's definitions are used instead of those for the CNTS approach since they are easier to check for feasibility and require less overhead to use. In addition, this allows our TS approach to be as general as possible since CNTS uses a more specialized neighborhood.

How this TS application differs from other local search approaches to the TTP is that it will only search through the neighborhood of feasible solutions. Infeasible solutions, for this problem, are solutions which meet the constraints of the double round robin schedule, but violate the AMC and NRC. Other approaches need to explore both feasible and infeasible solutions because it is unknown if the solution space of feasible solutions is fully connected [12]. This is not a problem with our approach since AFC-TTP is using it only to improve the candidate solutions created by the ants, and ACF-TTP is able to explore the whole solution space since it is constructing new solutions every time.

One of the problems of using TS for local search is that it is very slow in exploring the whole neighborhood for the TTP for each move. To help mitigate this, our TS implementation uses an elite candidate list [20]. To do so, it creates a master list of length $4 \cdot n$ of the best ranked moves. With each iteration of TS, the moves in the master list are re-evaluated to make sure they are still feasible and to re-rank them. It takes the best move which is not tabu and then repeats the process. There are two conditions in which the master list is remade: every $\frac{n}{2}$ iterations or when there is no feasible, un-tabooed moves left in the master list that improves the current solution. These parameters for the elite candidate list were chosen arbitrarily.

4.2 Pattern Matching

A new idea for the TTP is using pattern matching for constraint propagation. The idea behind it is to apply patterns in certain situations to help propagate the AMC. The AMC causes the most backtracking, thus it is desirable to reduce the conflicts caused by it. During the construction of a solution, these patterns will not be applied often, but it is those rare cases in which they need to be applied that cause some of the worst backtracking.

The general idea of these patterns is to look for certain combinations of remaining home and away games that indicate what locations can be played at in future time slots. For example, assume that a team has six remaining away games and one remaining home game. Then, with respect to the AMC, it can only play a home game in the fourth to last slot, while all other time slots must be away games. The pattern would then have a combination of symbols to indicate that domains need to be restricted to away games for all but the fourth to last time slot, while that time slot would have a symbol indicating it needs to be a home game.

There are four symbols used with the pattern matching: $\mathcal{U}, \mathcal{B}, \mathcal{A}, \mathcal{H}$. A symbol of $\mathcal{U}$ indicates the current time slot assignment is unknown, $\mathcal{A}$ and $\mathcal{H}$ indicates the current time slot assignment is restricted to away or home respectively, and $\mathcal{B}$ indicates the current time slot assignment can be either home or away. If the variable X represents any of these symbols, then $X \cup X = X$, $\mathcal{U} \cup X = X$, $\mathcal{B} \cup X = \mathcal{B}$, and $\mathcal{H} \cup \mathcal{A} = \mathcal{B}$.

4.2.1 Creating Patterns

Creating the patterns is an exhaustive process, but it is quick as there are only n^2 possible patterns, with half of the patterns mirror images of the other half. These patterns are created at the start of the algorithm, and then used throughout the running of the algorithm.

Algorithm 4 describes the process of creating these patterns. The main process will go through every possible combination of remaining home and away games possible, creating a pattern for each combination. This process takes in P, the three-dimensional matrix of patterns to be created, and κ, which is the length of the AMC. While the TTP requires AMC to be set with $\kappa = 3$, these patterns were designed to work with any length.

The three-dimensional nature of P is due to the nature of the patterns. The first two indices, i and j, correspond to the remaining home games and the remaining away games. The third index, k, corresponds to the index within a pattern located at i, j. The range of the indices i and j is $[0..n-1]$. The range of k for index i, j is $[0..i+j]$. Using this index notation, P_{ij} refers to a whole pattern located at i, j while P_{ijk} refers to the k^{th} element for the pattern at i, j.

Creating an individual pattern is a recursive process, trying every possibility of assignments that correspond to the combination of remaining home and away games. The process takes in P, V which is the particular schedule of home and away being created, h and a which are the total remaining home and away games, h_r and a_r which are the remaining home and away games in the recursive process, and d, the

Algorithm 4 Pattern making

```
1: procedure MakePatterns($P, \kappa$)
2:     for $i \leftarrow 1, n$ do
3:         for $j \leftarrow i \cdot \kappa + 1, n$ do
4:             for $k \leftarrow 1, i + j$ do
5:                 $P_{ijk} \leftarrow \mathcal{U}$
6:             PatternMaker($P, V, i, j, i, j, 1$)
7:             for $l \leftarrow 1, i + j$ do
8:                 $P_{jil} \leftarrow !P_{ijl}$

9: procedure PatternMaker($P, V, h, a, h_r, a_r, d$)
10:     if $h_r = 0 \wedge a_r = 0$ then
11:         for $i \leftarrow 1, h + a$ do
12:             $P_{hai} \leftarrow P_{hai} \cup V_i$
13:     if $h_r > 0 \wedge at\_most$ = feasible then
14:         $V_d \leftarrow \mathcal{H}$
15:         PatternMaker($P, V, h, a, h_r - 1, a_r, d + 1$)
16:     if $a_r > 0 \wedge at\_most$ = feasible then
17:         $V_d \leftarrow \mathcal{A}$
18:         PatternMaker($P, V, h, a, h_r, a_r - 1, d + 1$)
```

depth of the recursive process. A pattern is initially assigned all $\mathcal{U}$. After it has created a feasible tour of $\mathcal{A}$s and $\mathcal{H}$s while respecting the AMC, it can then combine the tour with the current pattern. It will try all possible tours in an exhaustive process.

The resulting pattern will then represent the restrictions needed for certain time slots given the current combination of remaining home and away games. For example, if $a = 2$ and $h = 1$, then this will result in the pattern $\mathcal{B}, \mathcal{B}, \mathcal{B}$, since it can create schedules of $\{\mathcal{H}, \mathcal{A}, \mathcal{A}\}$, $\{\mathcal{A}, \mathcal{H}, \mathcal{A}\}$, and $\{\mathcal{A}, \mathcal{A}, \mathcal{H}\}$. But if $a = 5$ and $h = 1$, this will then result in the pattern of $\{\mathcal{A}, \mathcal{A}, \mathcal{B}, \mathcal{B}, \mathcal{A}, \mathcal{A}\}$. This is because it can only create two feasible schedules, $\{\mathcal{A}, \mathcal{A}, \mathcal{H}, \mathcal{A}, \mathcal{A}, \mathcal{A}\}$ and $\{\mathcal{A}, \mathcal{A}, \mathcal{A}, \mathcal{H}, \mathcal{A}, \mathcal{A}\}$.

There are two possible ways to cut down the number of patterns that need to be made. The first is in regards to the AMC. With κ being the limit of the AMC, patterns are then only created when either the number of remaining home or away games is more than κ times larger than the other. The purpose of this is to reduce the number of patterns needing to be made to only situations where it will be able to propagate constraints. This then helps to speed up the creation of the set of patterns. These reduction of patterns needing to be created is due to the following theorem. Without loss of generality, we restrict ourselves to $h \leq a$ and leave the symmetric case of $a \leq h$ for the reader.

Lemma 1. *Let h and a be the number of remaining home and away games such that $h = 1$ and $h \leq a \leq \kappa$. All patterns that meet these characteristics will be composed of all $\mathcal{B}$s.*

Proof. Suppose to the contrary there exists a pattern with $h = 1$ and $h \leq a \leq \kappa$ that is not composed of all $\mathcal{B}$s. Then there is a value for a for which this is true. For every value of a, create all possible schedules by initially creating a schedule of the one $\mathcal{H}$ followed by a $\mathcal{A}$'s. Shift the $\mathcal{H}$ right from one slot to the next until it is in the last slot:

$a = 1 : \mathcal{H}_1\mathcal{A}_1|\mathcal{A}_1\mathcal{H}_1$

$a = 2 : \mathcal{H}_1\mathcal{A}_1\mathcal{A}_2|\mathcal{A}_1\mathcal{H}_1\mathcal{A}_2|\mathcal{A}_1\mathcal{A}_2\mathcal{H}_1$

...

$a = \kappa : \mathcal{H}_1\mathcal{A}_1...\mathcal{A}_\kappa|\mathcal{A}_1\mathcal{H}_1...\mathcal{A}_\kappa|...|\mathcal{A}_1...\mathcal{H}_1\mathcal{A}_\kappa|\mathcal{A}_1...\mathcal{A}_\kappa\mathcal{H}_1$

For all values of a, the composition of the set of schedules will result in a team being able to play either at home or away for each round, thus resulting in a pattern of all $\mathcal{B}$s. This contradicts the assumption that there is a pattern with $h = 1$ and $h \leq a \leq \kappa$ that is not composed of all $\mathcal{B}$s. $\square$

Lemma 2. *Let h and a be the number of remaining home and away games such that $h > 1$ and $h \leq a \leq h \cdot \kappa$. All patterns that meet these characteristics will be composed of all $\mathcal{B}$s.*

Proof. Suppose to the contrary there exists a pattern with $h > 1$ and $h \leq a \leq h \cdot \kappa$ that is not composed of all $\mathcal{B}$s. Let the multiplier $c = \lceil \frac{a}{h} \rceil$ and the remainder $d = a - c \cdot (h - 1)$. If $d = 0$, create an initial schedule such that there are h sets composed of one $\mathcal{H}$ followed by c $\mathcal{A}$s. If $d \neq 0$, create an initial schedule such that there are $h - 1$ sets composed of one $\mathcal{H}$ followed by c $\mathcal{A}$s with a final set of the last $\mathcal{H}$ followed by the final d $\mathcal{A}$s.

Beginning with the first set, enumerate from there a set of feasible schedules by rotating the $\mathcal{H}$ through the set of $\mathcal{A}$s one slot at a time as done in Lemma 1. Once the first set is done, continue on with the next set and do so until all sets have had their $\mathcal{H}$ rotated through their $\mathcal{A}$s. Due to the rotating of the $\mathcal{H}$s and $\mathcal{A}$s into every slot at some point in time, the composition of all the schedules will result in the possibility of a team playing either at home or away for every slot. This is as a result of Lemma 1. Each set created can be treated as a case of Lemma 1, and the rotating will lead to that set being a set of all $\mathcal{B}$s. This will then result in a full pattern of all $\mathcal{B}$s. This contradicts the assumption that there is a pattern with $h > 1$ and $h \leq a \leq h \cdot \kappa$ that is not composed of all $\mathcal{B}$s. □

Theorem 1. *Let h and a be the number of remaining home and away games. Then the smallest ratio of $h : a$ games in which a pattern will be created that is not all $\mathcal{B}$s is greater than $1 : \kappa$.*

Proof. Suppose to the contrary there exists a pattern with a ratio smaller than or equal to $1 : \kappa$ that is not composed of all $\mathcal{B}$s. Then there is a value for h and a for which this is true. If $h = 1$, then $h \leq a \leq \kappa$ and this will create a pattern of all $\mathcal{B}$s due to Lemma 1. If $h > 1$, then $h \leq a \leq h \cdot \kappa$ and this will also create a pattern of all $\mathcal{B}$s due to Lemma 2. Since in both cases every pattern will be composed of all $\mathcal{B}$s, this contradicts the assumption that there is a pattern with the ratio smaller than or equal to $1 : \kappa$ that is not composed of all $\mathcal{B}$s. □

In the case where there is no pattern created, it then leaves the pattern as $\mathcal{U}$, which will indicate in the future that there is no pattern for the current combination of home and away games. This will indicate that either the current pattern would be composed of all $\mathcal{B}$s, resulting in no constraints needing to be propagated, or that the current pattern is infeasible with respect to the AMC. In the case of the latter, this situation will never arise as long as the patterns are being applied, as they help prevent any infeasible combination of remaining home and away from coming up in the future.

The second possible way to cut down the number of patterns needing to be created is due to the patterns being mirrors of each other in terms of the number of remaining home and away games. A pattern for a specified combination of h and a will be a mirror of the opposite combination values of h and a. Any symbols of $\mathcal{H}$ and $\mathcal{A}$ will be swapped with the other, as done on Line 8. An example is when $a = 5$ and $h = 1$, which as described before will result in the pattern of $\{\mathcal{A}, \mathcal{A}, \mathcal{B}, \mathcal{B}, \mathcal{A}, \mathcal{A}\}$. If $a = 1$ and $h = 5$, this will result in a pattern of $\{\mathcal{H}, \mathcal{H}, \mathcal{B}, \mathcal{B}, \mathcal{H}, \mathcal{H}\}$. As can be seen, these two patterns are mirrors of each other if we swap the symbols $\mathcal{H}$ for $\mathcal{A}$. Thus, we only have to create half of the possible patterns, and swap the two symbols to create the other half of the patterns.

4.2.2 Using Patterns

Algorithm 5 shows how these patterns are applied during the constraint propagation. It takes in the set of patterns, the current team, the remaining home and away games, and the current time slot r_c.

Algorithm 5 Applying patterns for constraint propagation

```
1: procedure ApplyPatterns($P, t, h, a, r_c$)
2:     if $P_{ha}$ exists then
3:         for $i \leftarrow 1, h + a$ do
4:             $r \leftarrow r_c + i$
5:             if $P_{hai} = \mathcal{H}$ then
6:                 for $j \leftarrow 1, n$ do
7:                     if $@j \in D_{tr}$ then
8:                         $D_{tr} \leftarrow D_{tr} \setminus @j$
9:                         $D_{jr} \leftarrow D_{jr} \setminus t$
10:            else if $P_{hai} = \mathcal{A}$ then
11:                for $j \leftarrow 1, n$ do
12:                    if $j \in D_{tr}$ then
13:                        $D_{tr} \leftarrow D_{tr} \setminus j$
14:                        $D_{jr} \leftarrow D_{jr} \setminus @t$
```

It will first check if there is a pattern for the current combination of h and a. If there is no pattern, such as the case when all symbols would be $\mathcal{B}$s, it will then exit the procedure. If there is a pattern, the algorithm will then go through the team's future domains and restrict those domains to either home games if the pattern for the time slot is $\mathcal{H}$ or to away games if the pattern for the time slot is $\mathcal{A}$. If the symbol is a $\mathcal{B}$, it then does nothing to the domain being checked.

Here is an example of using patterns. Assume the domains of the last five time slots of a team, which has four away games and one home game remaining, are composed of:

$$\{\{+1, -2, -4\}\{-2\}\{+1, -5\}\{-2, -3\}\{+1\}\}$$

The combination of home and away games results in the pattern $\{\mathcal{A}, \mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{A}\}$. Applying this pattern would then remove all home games from the first and last time slot, reducing the team's domains to:

$$\{\{-2, -4\}\{-2\}\{+1, -5\}\{-2, -3\}\{\}\}$$

As can be seen, the pattern has caused the domain of the final time slot to become empty since it had intially only contained a home game but no away games. The algorithm then knows that the current partial solution is infeasible: no schedule can be created with the remaining games that would respect the AMC and the double round robin constraints. Thus four time slots in advance, the algorithm will know that it cannot satisfy the AMC.

The time for applying these patterns is $\mathcal{O}(n)$ since this is a linear process. As mentioned before, these patterns are not applied very often because its a rare case that these are needed. But as we will see later in the results in Section 5.1.1, these patterns can make a significant difference in reducing the time needed to construct solutions.

Chapter 5

Ant Colony Optimization Results

The purpose of computing is insight, not numbers.

-Richard Hamming

The Art of Doing Science and Engineering

This chapter looks at the results of AFC-TTP. It is split into two sections. The first section looks at the new ideas and components of AFC-TTP. The second section compares AFC-TTP with past approaches to the TTP, which are both the past ACO approaches along with the best TS and SA approaches for the TTP.

Unless otherwise specified, experiments were run with a colony of five ants. Pheromone reinitialization took place if $20 \cdot n$ cycles had passed since the last improvement to the best known solution and at least the same number of cycles had passed since the last pheromone reinitialization. For the pheromone, AFC-TTP used the standard definitions of τ_{max}. For τ_{min}, it used the same value with $\mathcal{MM}$AS for the TSP, with $\tau_{min} = \frac{\tau_{max}}{2n}$. For ACS's pseudorandom proportional rule, $q_0 = 0.9$. With the local search, it was applied to all ants, and was run for $5 \cdot n(n-1)$ iterations. The $n(n-1)$ represents the number of pairings in a schedule. Pheromone decay was set to $\rho = 0.8$. Most of the choices for these values were taken from past approaches of ACS and $\mathcal{MM}$AS being applied to the TSP, or in the case of local search, chosen arbitrarily. The reason for this was that one of the current issues with ACO is there is too many possible variables that can be optimized, and trying to optimize all is infeasible. As such, we used many of the values used for past approaches to similar problems, as they fitted in the range of commonly used values [15]. This also allowed us to focus on optimizing the variables for the new ideas presented along with the variables we believed would have had the largest impact on solution quality.

All tests with ACO were run using single cores of Intel Dual Core Processors running at 2.3GHz. We list all timings in seconds. When applicable, statistical significance tests were performed using t tests, with P values reported within the respective results.

5.1 AFC Components and Configuration

The first set of tests look at AFC-TTP. There are three aspects: the components of AFC, the difference between using AS's and ACS's rule for choosing values, and looking at the pheromone update schedule.

5.1.1 AFC Components

We first tested the components of AFC-TTP. We begin this with comparisons of the safe and unsafe backjumping, with and without ant restarts and pattern matching. These experiments were run on team sets of sizes 6 up to 20. Each test was done by having one ant construct 1000 solutions, with the tests averaged over 100 runs. For these particular tests, pheromone, heuristic information, and local search are not used, which allows the tests to focus solely on the components of AFC. When ant restarts are used, b is set to 100.

Figure 5.1 shows the results of these initial experiments. We do not display the results for running without ant restarts, as the timings were such that the experiments could not finish within a week for 10 teams. As can be seen, using unsafe backjumping always resulted in a smaller time needed to construct a solution compared to using safe backjumping. This helps confirm that unsafe backjumping can get out of an infeasible solution space faster, which reduces the number of backjumps and also

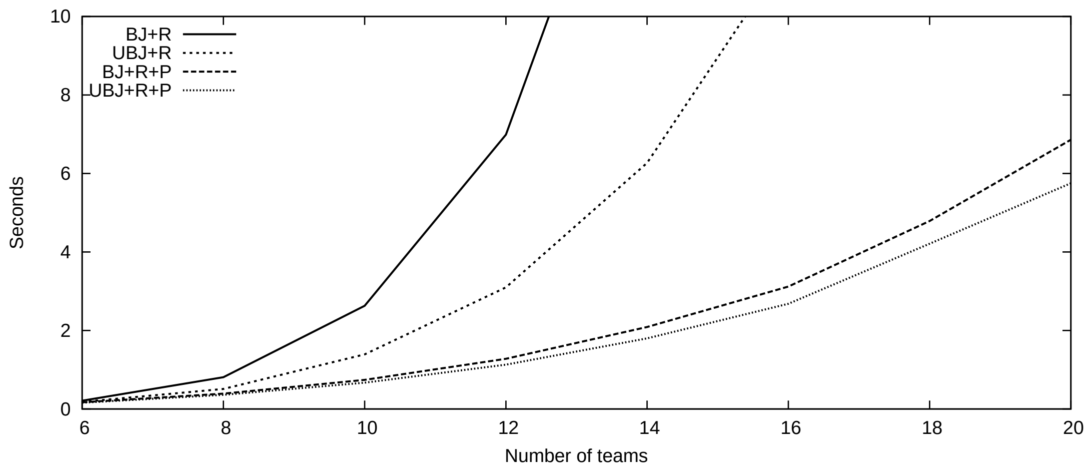


Figure 5.1: Comparison of using backjumping(BJ) and unsafe backjumping(UBJ) with and without ant restarts(+R) and pattern matching(+P). Timings represent the number of seconds for one ant to construct 1000 solutions averaged over 100 trials.

reduces the number of ant restarts. As will be seen later in this section, this savings in time is without any significant cost to the solution quality.

What had an even bigger impact than unsafe backjumping was the pattern matching. It significantly reduced the time needed to construct solutions, especially for team sets greater than 8. The reason pattern matching had such a profound impact is due to it being able to mitigate the conflicts caused by the AMC. It can find in advance when a solution would become infeasible, thus greatly decreasing the need for backjumping. This also the led to less ant restarts, thus overall reducing the time needed to construct solutions.

As we stated before, we do not show the results for running without ant restarts, due to the fact that the algorithm could not finish the tests for ten teams within a week of running time. The reason for this is that there are many various causes for a solution to become infeasible due to the various constraints of the problem. Pattern matching helped to reduce the conflict from the AMC, but there are also conflicts from the NRC along with the double round robin structure. Instead of trying to create constraint propagation for every possibility, the ant restarts allows us to bypass this. The added benefit is that it reduces the amount of overhead of

Table 5.1: Comparison of solution quality using safe backjumping against unsafe backjumping, both using ant restarts and pattern matching. Values are average distances found, with standard deviations displayed in brackets.

Set	Time	BJ+R+P	UBJ+R+P	P value
NL6	20	24433.7 (282.3)	24493.7 (213.6)	0.3571
NL8	40	42365.4 (637.0)	42309.7 (547.7)	0.7178
NL10	80	70281.9 (810.8)	70006.0 (908.8)	0.2197
NL12	160	135669.3 (1106.5)	135836.9 (1044.1)	0.5486
NL14	320	245393.6 (2157.2)	245181.2 (2408.9)	0.7203
NL16	640	343678.5 (2740.7)	343110.7 (2588.7)	0.4128

propagating all possible constraint possibilities, which is beneficial for larger team sets. Besides this is the fact that there are many feasible solutions for the TTP, thus the ideas of ant restarts and unsafe backjumping are very applicable to this sort of problem.

Returning again to safe and unsafe backjumping, it is also important to look at the quality of the solutions, specifically to ensure that unsafe backjumping does not cause the solution quality to degrade. We tested this with short runs averaged over 30 trials on the NL set, using heuristic information and pheromone for these set of tests. As can be seen in Table 5.1, the unsafe backjumping did not lead to a significant degradation of solution quality, with slightly better average quality for four of the six team sets. When constructing a solution, unsafe backjumping will sometimes miss a better solution and end up with a poorer solution, while at other times pass over a poorer solution and end up with a better solution. But since more solutions can be constructed with unsafe backjumping, this then gives a greater chance of finding a better overall solution, thus resulting in four the six team sets having slightly better solution quality.

The next set of tests concern the ant restarts and the parameter b, the time until ant restarts. Values tested for this were 50, 100, 200, and 500. These tests were on team sets between 6 and 16 while using unsafe backjumping and pattern matching. They used the same testing procedure of each ant constructing 1000 solutions, with results averaged over 100 runs. As can be seen in Table 5.2, the minimal time for constructing solutions coming when b is set to 100. The differences in time when

Table 5.2: Comparison of time needed to construct solutions for different values of ant restart's parameter b. These tests were done with using unsafe backjumping and pattern matching. All average timings are in seconds with standard deviations in brackets. The second table records the P values when comparing the results for each of the values for b.

Teams	50	100	200	500
6	0.16 (0.02)	0.16 (0.01)	0.18 (0.01)	0.16 (0.00)
8	0.38 (0.03)	0.36 (0.01)	0.36 (0.01)	0.35 (0.01)
10	0.67 (0.01)	0.67 (0.01)	0.76 (0.04)	0.67 (0.02)
12	1.25 (0.10)	1.13 (0.02)	1.17 (0.03)	1.18 (0.04)
14	1.90 (0.13)	1.80 (0.02)	1.86 (0.10)	1.89 (0.07)
16	2.91 (0.18)	2.68 (0.04)	2.77 (0.10)	2.84 (0.08)

Teams	50 v 100	50 v 200	50 v 500	100 v 200	100 v 500	200 v 500
6	1.0	< 0.0001	1.0	< 0.0001	1.0	< 0.0001
8	< 0.0001	< 0.0001	< 0.0001	1.0	< 0.0001	< 0.0001
10	1.0	< 0.0001	1.0	< 0.0001	1.0	< 0.0001
12	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.0469
14	< 0.0001	0.0156	0.4990	< 0.0001	< 0.0001	0.0148
16	< 0.0001	< 0.0001	0.0005	< 0.0001	< 0.0001	< 0.0001

comparing 100 to the other three values was generally significant, indicating that have a value which is too small or large can significantly impact the ability to quickly construct solutions.

A final test with b was setting it to 1, which then leads to a configuration similar to past works [4, 34]. This resulted in a much poorer performance for constructing solutions. The time needed to construct solutions for team sets 6 to 16 were 0.18 (0.01), 0.41 (0.01), 0.82 (0.02), 1.52 (0.04), 2.59 (0.06), and 4.11 (0.17). Compared with the results presented in Table 5.2, these times are longer in average length have a larger standard deviation. This helps show it is more beneficial to give the ants more time to construct solutions instead of restricting the number of possible backtracks or backjumps to a small number.

5.1.2 ACS's and AS's Rules

The next comparison is looking at whether to use AS's or ACS's rule for choosing values. Also we must look at if heuristic information should be used or not. In order to not use heuristic information, α is set to 0. These were longer tests, thus were only performed on NL8, NL10, NL12, and NL14 with times of 600, 1800, 3600, and 5400 seconds respectively. We did not test on NL6 since the instance is too easy to solve, with all runs resulting in the optimal solution being found. All tests were averaged over 30 trials.

Table 5.3 shows the results of these tests. As can be seen, using ACS's rule for choosing values outperformed that of AS's rule, both with and without heuristic information. This is due to ACS's more aggressive nature, and is explored further later in this section. What is more surprising though is the use of heuristic information. When using AS's rule, we see the expected result of better performance, with lower average distances. But with ACS's rule, it performs worse when heuristic information is used. We believe this is due to the aggressiveness of ACS. When using the heuristic information, it sacrifices the long-term gain of the pheromone for the short-term gain which is caused by the heuristic information. While the short-term gain is beneficial when AS's rule is used as it has more emphasis on exploration, this short-term gain is instead detrimental for ACS's rule.

We further ran more experiments on NL8 and NL12 to better understand why ACS's rule performs better than AS's rule. Figure 5.2 shows the results of these experiments, with the average of the best current ant compared in similarity with the best ant seen so far. The average similarity was measured by comparing schedules with respect to the pairings within each time slot. For example, if both schedules had pairings located in the same time slots, then they would have 100% similarity, while schedules with half of the pairings in the same time slot would then have 50% similarity. As can be seen, using ACS's rule allows for more similarity, which in turn allows for the ant to reuse more of the information gained from previous cycles. With AS's rule, there was very little similarity, showing that the algorithm was spending too much time exploring and not enough using exploitation. This helps explain why ACS's rule is more beneficial for this particular application.

Table 5.3: Comparison of solution quality using AS's and ACS's rules for choosing values, with and without heuristic information. Experiments on NL8 used 600 seconds, on NL10 used 1800 seconds, on NL12 used 3600 seconds, and on NL14 used 5400 seconds. The results are averaged over 30 trials, with averages and standard deviations shown. The following table shows the relevant P values when comparing between the four aspects.

Set	ACS	ACS+H
NL8	39772.9 (129.7)	39753.0 (87.3)
NL10	61004.7 (633.4)	61213.3 (804.4)
NL12	116158.5 (1217.9)	118875.6 (914.3)
NL14	204765.1 (3049.1)	209043.7 (1984.4)
Set	AS	AS+H
NL8	39872.9 (107.9)	39778.3 (91.1)
NL10	63083.9 (347.3)	62659.8 (356.4)
NL12	120876.8 (431.3)	120408.3 (531.6)
NL14	214763.3 (1006.2)	212814.8 (1163.0)

Teams	ACS v ACS+H	ACS v AS	ACS v AS+H
NL8	0.4885	0.0019	0.8526
NL10	0.2690	< 0.0001	< 0.0001
NL12	< 0.0001	< 0.0001	< 0.0001
NL14	< 0.0001	< 0.0001	< 0.0001
Teams	ACS+H v AS	ACS+H v AS+H	AS v AS+H
NL8	< 0.0001	0.2766	0.0005
NL10	< 0.0001	< 0.0001	< 0.0001
NL12	< 0.0001	< 0.0001	0.0004
NL14	< 0.0001	< 0.0001	< 0.0001

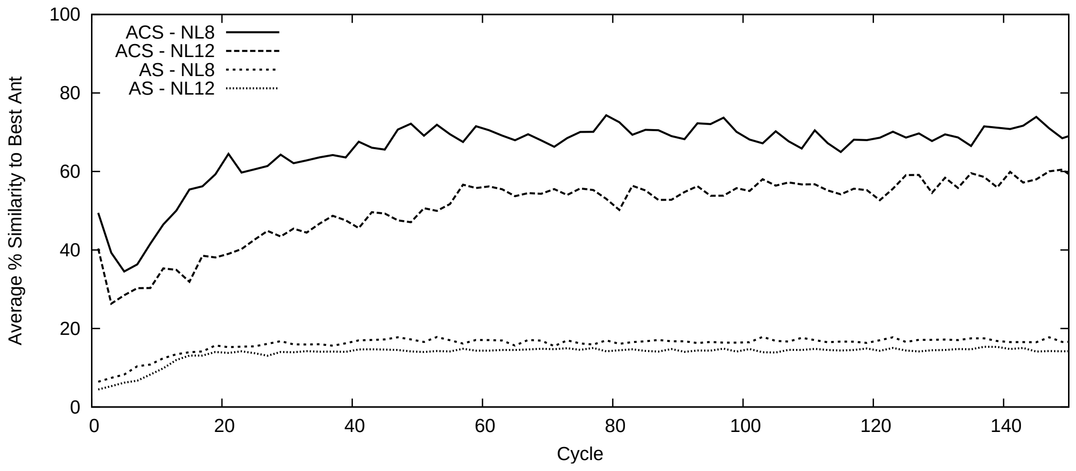


Figure 5.2: Comparison of using AS's and ACS's rules. Average similarity is measured through comparison of an ant's schedule with that of the best ant seen so far in terms of pairings for each time slot. For example, 100% similarity indicates both schedules are identical and 50% similarity indicates half of the pairings are identical within the time slots while the other half of pairings are found in different time slots.

5.1.3 Pheromone Update Schedule

The next set of experiments deal with the pheromone update schedules, specifically whether to use the global best, restart best, or iterative best ant for updating the pheromone matrix at the end of a cycle.

Before a pheromone reinitialization has taken place, AFC-TTP alternates between the global best and and the iterative best ant. For cycles where a pheromone reinitialization has taken place, we look at different combinations of ants to alternate between. The first combination is to continue with the global and iterative best ants. The second combination is to use the restart and iterative best ant. And the final combination uses the restart best ant and iterative best ants for the first half of possible cycles after a pheromone reinitialization, then alternating between the global and iterative best ant afterwards. These experiments were done by testing on NL8, NL10, NL12, and NL14, with solutions averaged over 30 trials. We tested on NL14 due to the close similarity in solution quality. Having the NL14 helps to better determine the choice for updating.

Table 5.4 shows the results of these experiments. As can be seen, the best schedule

Table 5.4: Comparison of pheromone update schedules. GB+IB is a combination of global and iterative best ants; RB+IB is a combination of restart and iterative best ant; and RB/GB+IB is a combination of restart and iterative best ant for the first half after pheromone reinitialization, then a combination of global and iterative best ant. The times used were 600 seconds for NL8, 1800 for NL10, 3600 for NL12, and 5400 for NL14. The second table reports the P values when comparing between each approach.

Set	GB+IB	RB+IB	RB/GB+IB
NL8	39772.9(129.7)	39722.8(9.9)	39724.7(13.7)
NL10	61004.7(633.4)	60687.6(367.8)	60607.9(483.0)
NL12	116158.5(1218)	116075.1(642)	116414.7(684)
NL14	204765(3049)	202814(1919)	203541(1760)
Set	GB+IB v RB+IB	GB+IB v RB/GB+IB	RB+IB v RB/GB+IB
NL8	0.0392	0.0476	0.5405
NL10	0.0211	0.0084	0.4750
NL12	0.7413	0.3193	0.0521
NL14	0.0044	0.0618	0.1316

was the one which used a combination of restart best ant and iterative best ant for alternating cycles, with some significance being shown that using a global best ant is not the best approach for this application. We believe this schedule resulted in the best average solution due to it allowing for more exploration than if the global best ant were used during the later process. Only using the restart best ant allows for more exploration since the global best ant can anchor the algorithm to a local minimum in the solution space.

5.2 Comparison With Past Approaches

There are two sets of comparisons which were done with past approaches. The first is with past ACO approaches and the second is with past TTSA and CNTS approaches.

5.2.1 Past ACO Approaches

The first comparisons of AFC-TTP are with past ACO approaches to the TTP: the direct approach by Crauwels et al. and the hyper-heuristic by Chen et al. The problem with comparing with these two approaches is that neither approaches gave

Table 5.5: Comparison of AFC-TTP with previous ACO approaches.

Set	Crauwels	Chen	AFC-TTP		
	Best	Best	Time	Avg	Best
NL4	8276	8276	10	8276	8276
NL6	23916	23916	20	23916	23916
NL8	40797	40361	40	39785.5	39721
NL10	67871	65168	80	61951.3	60399
NL12	128909	123752	160	118609.1	115871
NL14	240445	225169	320	208146.7	203205
NL16	346530	321037	640	296220.75	292214

timings for all of their experiments. Another problem was that for their experiments, they only gave their best results and not any averages. All timings we used for these were shorter than any timings they listed when taking into consideration of processor speed, and display both the best results and the average results.

Table 5.5 shows the results of these first comparisons. As can be seen, AFC-TTP had results that greatly improved upon the past ACO approaches. For all team sets greater than six, AFC-TTP had better averages than the past approaches best results. This became more significant as the team set became larger.

When comparing with Crauwels et al., we believe AFC-TTP performed better largely due to it being able to construct solutions quickly, allowing for more cycles even with a shorter running time. This leads to more solutions being constructed, which in turn allows for more exploration of the solution space and more possibility of exploiting past solutions.

Unfortunately, it is more difficult to explain why ours greatly outperformed Chen et al. due to the different uses of ACO, with theirs using ACO as a hyper-heuristic and ours being a direct approach. Theirs is also the only approach which has attempted to use any hyper-heuristic for solving this problem, thus it is unknown whether their performance would be typical of a hyper-heuristic being applied to this problem.

5.2.2 Past TS and SA Approaches

The next set of experiments compares with the best past TS [12] approach, CNTS, and the best SA [2] approach, TTSA. We ran two separate experiments, running ACO-TTP in the same amount of time as the average time for each approach. When comparing with CNTS, we used the same experimental setup, which consisted of 10 trials per problem instance. When comparing with TTSA, we averaged over 30 trials instead of the 50 that they had used. This is due to the long running times they used and time constraints we had.

Table 5.6 shows the results of these comparisons. The average solution of AFC-TTP was lower than the other approaches by a few percentage points, with some being considered significant, but for some problem instances it did better than CNTS. While the averages may have been lower, these results are very promising, as they show ACO can perform at a similar level to other metaheuristic approaches on the TTP.

5.3 Limits of the Results

With the results presented, there are limits of this work in terms of the pheromone representation, constraint propagation and pattern matching, and parameter optimization.

5.3.1 Pheromone Representation

The pheromone representation used in this approach is not the only possible choice. The previous ACO approach for this problem used a successor relationship between teams [10]. We have also briefly explored alternative pheromone representations which were more aligned to the successor relationships. They were not explored in depth mostly due to initial experimental results had shown our approach performing worse with such a representation. As a caveat, this may be due to the fact that we had constructed this approach with the pheromone representation presented here as the

Table 5.6: Comparison of AFC-TTP with CNTS and TTSA.

Set	CNTS		AFC-TTP				
	Avg (StdDev)	Best	Time	Avg (StdDev)	Best	Avg % Dif	P value
NL10	60424.2 (823.9)	59876	4969.51	59928.3 (155.47)	59634	-1.0	0.0778
NL12	114880.6 (948.2)	113729	7660.07	114437.4 (895.7)	112521	-0.4	0.2968
NL14	197284.2 (2698.5)	194807	20870.07	198950.5 (1294.43)	196849	0.8	0.0953
NL16	279465.8 (3242.4)	275296	35931.27	285529.6 (3398.57)	278456	2.2	0.0007
CIRC10	259.2 (2.3)	256	3131.47	254 (3.13)	248	-2.0	0.0005
CIRC12	440.4 (1.7)	438	8167.18	436 (4)	430	-1.0	0.0049
CIRC14	694.4 (6.4)	686	10183.59	692.8 (9.25)	674	-0.2	0.6582
CIRC16	1030.0 (8.7)	1016	18896.9	1039.6 (5.56)	1034	0.9	0.0087
CIRC18	1440.8 (10.5)	1426	39010.77	1494.8 (7.61)	1486	3.7	< 0.0001
CIRC20	1998.4 (17.7)	1968	47316.76	2061.4 (8)	2046	3.1	< 0.0001

Set	TTSA		AFC-TTP				
	Avg (StdDev)	Best	Time	Avg (StdDev)	Best	Avg % Dif	P value
NL8	39721 (0)	39721	1188.32	39721 (0)	39721	0.0	1.0
NL10	59605.96 (53.36)	59583	29190.02	59773.5 (131.34)	59583	0.3	< 0.0001
NL12	113853 (467.91)	112800	49658.27	114427.4 (465.81)	113523	0.5	< 0.0001
NL14	192931.86 (1188.08)	190368	169316.89	197656.6 (1079.01)	195627	2.4	< 0.0001
NL16	275015.88 (2488.02)	267194	139240.2	283637.4 (1865.52)	280211	3.1	< 0.0001

foundation, thus changing to a different representation may cause worse performance. For future considerations then, one could look at alternative representations beyond what is used here.

5.3.2 Constraint Propagation and Pattern Matching

We used a simple constraint propagation technique here in the form of forward checking. More advanced techniques, such as arc consistency, were initially considered but later dropped. This was due to the overhead associated with these advanced techniques. With this problem, the feasible solution space is large, and the ants can construct solutions quickly most of the time using the basic techniques presented. It is only in the rare case where the algorithm will go through excessive backtracking. Thus using techniques like arc consistency would create unnecessary overhead for most of the constructed solutions, and it is questionable whether they would help to mitigate the rare occurrences.

The motivation behind the pattern matching idea was to create a strong propagation technique that was fast to implement with minimal overhead. As the results showed, for this particular application, it did help to mitigate the amount of backtracking caused by the AMC. No formal comparison was done to compare it with more advanced techniques, and could be possible future research.

5.3.3 Parameter Optimization

One aspect that was not performed in this research is the optimization of parameters. In many cases of past applications of ACO to various problems, the parameters were optimized. We unfortunately had to forgo this due to computation and time constraints. One of the difficulties of working with the TTP is that it requires long running times, which makes it difficult to optimize parameters. As such, we were limited to reusing parameter settings from similar problems such as the TSP. We believe that were the parameters optimized, the approach may have exhibited results that exceed those seen with TTSA and CNTS.

Chapter 6

Concurrent Iterative-Deepening-A*, Forced Deepening, and Elite Paths

We are like dwarfs standing upon the shoulders of giants, and so able to see more and see farther than the ancients.
-Bernard of Chartres

This chapter starts describing the second of two approaches for the TTP. No longer will we be looking at finding good, but not necessarily optimal, solutions with metaheuristics but instead looking at finding optimal solutions with heuristic search algorithms.

This chapter is split into four sections. The first revisits A* [22, 23] and Iterative-Deepening-A* [29] along with past parallel approaches to IDA*. The second introduces Concurrent IDA* (CIDA*), our new approach for parallelizing IDA*. The third introduces two additional new ideas, forced deepening and elite paths. The final section looks at applying these ideas to the Traveling Salesman Problem, a simpler problem for a more thorough testing of these ideas.

6.1 Heuristic Search, A*, and IDA*

The focal area of research for these following chapters is in heuristic search for finding optimal solutions to the TTP with particular focus on IDA*. This chapter first begins with the A* search algorithm and the later-derived IDA* algorithm. It then discusses past approaches to parallelizing IDA*. Finally, it delves into our new approach which ties past concepts together along with introducing a few new ideas of our own.

6.1.1 A*

A* is a best-first search algorithm for finding optimal solutions. It solves a problem expanding the minimal number of nodes needed, which means that no other systematic search algorithm can solve the same problem expanding fewer nodes if it were to use the same heuristic function as A*. It does this by keeping in memory a sorted queue of all unexpanded nodes, and expands the next node which has the minimal f-value.

The $f(n)$ value for a node n is a calculation of the expected cost for the whole solution based on the cost of the current partial solution from the root node up to n, $g(n)$, and a heuristic estimate of what the rest of the solution from n until the end will cost, $h(n)$. For A* and derived algorithms, $f(n) = g(n) + h(n)$.

The first solution found by A* is optimal. This is due to the best-first approach, as each node expanded will always be the node with the minimal expected cost. Thus when a solution is found, the algorithm knows that all other solutions will either have the same cost as the current solution or have a higher cost.

A* is able to prove a solution is optimal as long as a certain requirement is met: the heuristic estimate used is admissible. This means the heuristic estimate cannot overestimate the remaining cost. If it were to overestimate, then the optimal solution may not be found with a lesser-quality solution being found instead.

6.1.2 Iterative-Deepening-A*

Even though A* can find the optimal solution with the minimal number of nodes being expanded, it does have a shortcoming in that it has to store all unexpanded nodes within a priority queue. While for a small problem this is not an issue, with larger problems A* will quickly exhaust the computer memory before a solution can even be found.

To overcome this, the algorithm IDA* was created that works in linear-space. This is accomplished by going through multiple iterations of depth-first search with a limiting threshold instead of doing a best-first search. Each successive threshold increases by the minimal amount based on the f-values found the previous iteration. By increasing the limiting threshold by the minimal amount, this gives IDA* the same optimality guarantee as A* for the first solution found. In addition, IDA* will not expand any nodes not seen by A*.

The limiting threshold, $f^t(I)$, of iteration I is the minimal f-value from the previous iteration which exceeded the previous limiting threshold. Thus while an iteration is being performed, the algorithm keeps track of the limiting threshold for the next iteration, $f^t(I + 1)$.

Algorithm 6 shows the basic process of IDA*. After initializing the appropriate f-values, the algorithm goes through an iterative process of increasing limiting thresholds. For each iteration, it will perform a depth-first search. Whenever a node is expanded during the search, the algorithm will check that the node's f-value is within the limit of $f^t(I)$. If the f-value exceeds the limit, then the algorithm closes the node and expands a different, untried node in the search. The search continues until either a solution is found or when all nodes have been expanded that are within the $f^t(I)$-cost limit.

Once the depth-first search for a given iteration is complete, the algorithm will perform one of two actions. If no solution has been found, it will then update the appropriate f-values and start the next iteration. If a solution has been found, it will then return the solution and finish the iterative process.

Algorithm 6 IDA*

```
1: procedure $IDA*$
2:     $I \leftarrow 1$
3:     $f^t(I) \leftarrow InitialValue$
4:     $f^t(I+1) \leftarrow \infty$
5:     while $SolutionNotFound$ do
6:         DepthFirstSearch($f^t(I)$)
7:         if $SolutionFound$ then
8:             return $Solution$
9:         $f^t(I) \leftarrow f^t(I+1)$
10:        $f^t(I+1) \leftarrow \infty$
11:        $I \leftarrow I+1$
```

While IDA* will re-expand more nodes than A* due to the iterations, in most applications it is the final iterations which dominate the time spent by the algorithm. This is due to the number of expanded nodes growing at an exponential rate each iteration. Therefore, the extra cost in time is less of an important factor, especially compared to the trade off of being able to run the algorithm in linear-space.

6.1.3 Parallel Approaches to Iterative-Deepening-A*

There have been multiple approaches for parallelizing IDA*. Many of the common features they had to deal with is finding ways to spread the work amongst the different processors along with dealing with load balancing, which makes sure that no processor is left idle for too long. These past approaches can be put into two groups, those which used tree decomposition [21, 35, 38] and those which used window parallelization [1, 9, 36].

The first approach to use tree decomposition was work by Rao et al. [38]. Their approach to tree decomposition was to use local stacks for each processor, with the stacks containing nodes needing to be expanded. Should a processor's stack become empty, it would then request from another processor for a portion of their work to work on. The general outline was that one processor would begin with the root node in its stack for an iteration while the other processors' stacks would be empty. These processors would then request work from the processor which contained the root. At the end of an iteration, the processors would then communicate with one another to determine the new limiting threshold for the next iteration, or if a solution had been found, then cease operation. Their approach was able to exhibit strong speed-up performance, scaling well with the number of processors used.

The next approach using tree decomposition was work by Powley et al. [35]. This approach differed from other approaches presented here in that it was designed for a single-instruction, multiple-data machine. They accomplished this through partitioning the search space and giving each processor a different subtree to work upon. This partitioning was done so that the number of subtrees was equivalent to the number of processors. They approached the issue of load balancing from three points. The first was through the initial distribution, which tried to assign frontier nodes with all the same f-values. The second was between iterations, where information gained from the previous iteration was used to help distribute the work at the start of a new iteration. The last was during an iteration: when a certain number of processors had become idle, the search will stop temporarily so that work can be redistributed amongst the processors. This last aspect used the notion of triggers to determine when the processors should stop and redistribute the work. This approach was able to reach speed-ups in the range of 53% – 69%, with the efficiency decreasing as the number of processors increased.

The third approach using tree decomposition was work done by Hafidi et al. [21]. As with the previous approach, this approach also distributed the work amongst the processors with a one-to-one ratio of frontier nodes and processors. What differed though is that this approach was designed for distributed memory using a master-slave paradigm. The master process would distribute the work amongst the processors, and the slaves would do the depth-first search for each successive iteration. In dealing with load balancing, when a processor became idle, it would communicate to the master process for more work. The master process would then request the work load from all of the slaves, determining which one has the heaviest load to work through. The master process would then request the slave with the heaviest workload to send some of its work to the idle slave. The results of this work showed an efficiency in the range of 89% and 98%.

The first approach to use parallel window search was by Powley and Korf [36] which combined parallel window search with node ordering. Each processor would be given an iteration and perform a search on it, with each processor given an iteration that had a different threshold limit. This is possible for problems like the sliding puzzle, which they worked with, since the f^t-value increases monotonously for each iteration. When a processor finished its search on an iteration, if no solution had been found, it would jump ahead to the next, untried f^t-value. Once a solution was found, all processors working on iterations with greater threshold values would stop, while those working on iterations with smaller threshold values would continue until they either found a better solution or completed their search. To improve the performance, the authors also implemented node ordering. A set of frontier nodes would be sorted so that the processors would first search down the paths of the frontier nodes more likely to contain an optimal solution. In doing so, the optimal solution would hopefully be found at the beginning of a search through its corresponding iteration, which would reduce the extraneous node expansion of later iterations by cutting off their searches sooner.

The next approach to use parallel window search was by Diane et al. [9]. As with the previous approach, they also implemented a form of node ordering, but this implementation differed. Instead of creating a frontier set and going through the set, the authors would change the order that nodes were expanded. This change of ordering would effect all levels of the search tree. Using the sliding puzzle as an example, IDA* during the first iteration may try expanding node n's children in the order of left, right, down, and then up. In subsequent iterations, the node ordering might cause the algorithm to expand n's children in a different order, for example down, right, up, and then left. The advantage of this approach over the previous approach was the it reduced the space needed to implement this idea.

A more recent approach [1] also used window parallelization, but on a cluster of workstations. The significant difference though with this approach and past approaches is that it relaxed the optimality guarantee of IDA* since the algorithm would stop as soon as a solution was found. Their purpose was to evaluate the number of nodes expanded and the quality of the solutions against the size of the computing cluster along with the time needed for the first solution to be found.

6.2 Concurrent IDA*

CIDA* is a new approach to parallelizing IDA*. The key difference between CIDA* and past approaches is a new way of using subtrees which combines the concepts of tree decomposition and node ordering, allowing for parallelization, while introducing a new idea of subtree skipping. We note that CIDA* was not designed to compete with past approaches, but instead was designed to bring together past concepts while creating a framework which allows for subtree skipping, an idea not possible with past approaches.

6.2.1 Subtrees and Subtree Forest

CIDA* uses subtrees to parallelize IDA*. These subtrees decompose the search tree up to a set of frontier nodes all at the same depth in the search tree, and the subtrees are used throughout the running of the algorithm for node ordering and parallelization. Unlike past approaches which used tree decomposition to create a set of frontier nodes with a one-to-one ratio to the number of processors, the subtrees here will generally outnumber the processors by a significant amount. This allows for the implementation of ideas of node ordering and subtree skipping, as explained in this section.

Subtree and Subtree Forest Definitions

A subtree s represents a partition of the search space down to a depth of s^d. This then represents the first s^d nodes of a search tree along with the search tree that builds off of the tail node of the subtree. A subtree forest, $\mathcal{S}$, is the union of all subtrees which

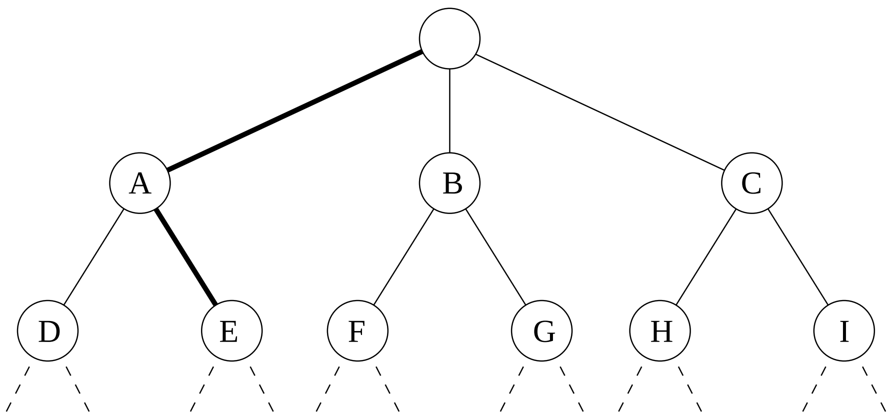


Figure 6.1: A partial view of a tree with only the first two levels of the tree shown. A subtree s of depth 2, (A,E), is highlighted.

create all feasible combinations of the first s^d nodes of the search tree. Figure 6.1 represents a search tree up to a depth of two. Using the figure as a guide, if $s^d = 1$, then $\mathcal{S} = \{(a), (b), (c)\}$, and if $s^d = 2$, then $\mathcal{S} = \{(a, d), (a, e), (b, f), (b, g), (c, h), (c, i)\}$. If $s^d = 3$, then this would extend the permutations found when $s^d = 2$ down to the next level of the search tree.

What differentiates a subtree forest from a collections of paths is that each subtree will contain additional information which allows for it to be manipulated as needed for the algorithm. Along with the permutation a subtree represents, the subtree stores with it an s^f value which is the minimal f-value that it found the previous iteration it was worked upon which exceeded the limiting f^t-value. It also keeps the depth at which this value was found. The reason for keeping these two values is so that $\mathcal{S}$ can be sorted by a criteria which fits the problem that is being looked at. This allows for implementation of node ordering, explained in the proceeding section, and load balancing, explained later in Section 6.2.2.

Subtree Forest Creation and Usage

The subtree forest is created at the beginning of the algorithm and is then used throughout the running of the algorithm. This is done through a breadth-first search up to the specified subtree depth s^d. Once created, these subtrees do not change in size and can be viewed as static.

During an iteration, the processor or set of processors will go through the subtree forest, constructing solutions based on the subtree being processed. A processor will treat the frontier node of the subtree as the root of a search tree instead of using the root of the whole problem's search tree as done with IDA*. While running the search on each subtree, it acts the same as done with IDA* in terms of f-values and thresholds.

Between each iteration, the subtrees in the subtree forest are sorted by a problem-specific criteria. They are sorted in order that the subtree most likely containing the optimal solution will be worked upon first. This allows the subtree forest to incorporate the notion of node ordering, which has been shown to help reduce the number of nodes expanded in the final iteration [36]. This node ordering will also be useful later on when working with Forced Deepening, as explained in Section 6.3.1.

Reducing the Search Space with Subtree Skipping

There are two aspects where the search space can be reduced with subtrees. The first has already been discussed, which is the usage of node ordering for the final iteration. The second aspect is an idea called subtree skipping, which impacts the iterations prior to the final iteration.

Under IDA*, the whole search space needs to be explored each iteration up to the limiting f^t-value to ensure that the f^t-values are updated correctly for each iteration. When using subtrees, partitions of the search space can be skipped. Subtrees whose s^f values exceed that of $f^t(I+1)$ can be skipped since it is known ahead of time that they will have no affect on updating $f^t(I+1)$ or its own s^f value. This is summarized with the following theorem.

Theorem 2. *Let s^f be the f-value that a subtree found which exceeded the previous $f(I)^t$ value. If $f(I+1)^t$ is the next iteration's limiting f-value and if $s^f > f(I+1)^t$, then the subtree can be skipped without losing guarantee of optimality.*

Proof. A subtree's s^f is the minimal f-value that it can find which is equivalent or exceeds $f^t(I)$. For any iteration that subtree s is processed, its s^f value will either stay the same or become larger. If $s^f > f(I+1)^t$, then it cannot find a f-value in which $f(I)^t \leq f < f(I+1)^t$ is true. Thus processing that subtree would not change s^f, nor would it change the value of $f(I+1)^t$. Because of this, subtree s can safely be skipped while still guaranteeing optimality for any solution that is found. □

This skipping of subtrees is only applicable when all the subtrees do not have the same s^f values. For example, when applying IDA* to the sliding puzzle [36], all the frontier nodes contain the same f-values. Thus, for problems like the sliding puzzle, no subtrees would be skipped, but for other problems such as those looked at in this paper, not all subtrees contain the same s^f values.

6.2.2 Parallelization

CIDA* uses the subtree forests to parallelize IDA*. It does so in a master-slave paradigm [45] similar to the work by Hafidi et al. [21]. Algorithm 7 explains how this is done.

Algorithm 7 Concurrent IDA*

```
1: procedure CIDA*
2:     $\mathcal{S} \leftarrow$ CreateSubtreeForest
3:     $SubtreePriorityQueue \leftarrow \mathcal{S}$
4:     $I \leftarrow 1$
5:     while $SolutionNotFound$ do
6:         while $SubtreePriorityQueue$ not empty concurrently do
7:             Obtain Subtree $s$ from $SubtreePriorityQueue$
8:             if $s^f \leq f(I+1)^t$ then
9:                 Construct $Solution$ using $s$
10:                if $SolutionFound$ then
11:                    return $Solution$
12:                Update values of $s$
13:            $ProcessedSubtreesSet \leftarrow ProcessedSubtreesSet \cup s$
14:        $f(I)^t \leftarrow f(I+1)^t$
15:        $f(I+1)^t \leftarrow \infty$
16:        $I \leftarrow I+1$
17:        $SubtreePriorityQueue \leftarrow ProcessedSubtreesSet$
```

CIDA* first begins with the master process creating $\mathcal{S}$ and initializing I. It then enters an iterative process until a solution has been found. At the beginning of the iteration, the slaves goes through $\mathcal{S}$, working with the subtrees and constructing solutions off of the frontier nodes contained in the subtrees. It will skip any subtrees whose s^f values exceed $f^t(I+1)$. The process of working through the subtrees in $\mathcal{S}$ is a concurrent process, as the start of one subtree is not dependent of the finish of another subtree. This is also then the parallel aspect of the algorithm, where different processors will work with different subtrees. After a subtree has been processed or skipped, it is placed in a set containing processed subtrees, which will be reused for the next iteration.

After all subtrees have either been processed or skipped, the master process will then update the appropriate f^t-values and move the processed subtrees back into the priority queue. This iterative process continues until a solution has been found. At this point, all slave processes cease operation and the optimal solution is returned.

Communication

Working with subtrees requires three points of communication between the master process and slaves when running CIDA* across multiple processors: distribution of subtrees, updating of $f^t(I+1)$ during an iteration, and the finding of a solution. How this is dealt with matters on what type of machine CIDA* is being run on.

If using a shared-memory approach with multiple processors, then the master process will only run at the beginning and between iterations. During an iteration, all control will be between the slaves. They will all have access to $\mathcal{S}$, taking turns obtaining subtrees when needed, and can update $f^t(I+1)$ as needed. In the final iteration, the first processor to find a solution returns that solution, and all slaves then cease operation.

If using a distributed-memory approach, the master process will have to run at all times. It will be responsible for communication between itself and the slave processes and will have sole-control of the priority queue holding $\mathcal{S}$, sending out subtrees to the slaves when needed. The master process also deals with communication for updating

$f^t(I+1)$ between iterations. For the final point, when a solution has been found, the master process will then inform all slaves to end their search.

Load Balancing

A necessary point to address when working with parallelism is load balancing. It is undesirable for any processor to be idle while waiting for other processors to finish their work. This problem is addressed by using the subtree forest.

The first aspect of dealing with the load balancing is having more subtrees than processors, or more importantly, having significantly more subtrees than processors. By having more subtrees than processors, this results in a finer partitioning of the search space, which helps reduce the time needed for a processor to work with a single subtree. This then reduces the maximal amount of idle time a processor can have at the end of an iteration.

The second aspect is with the sorting of $\mathcal{S}$. By keeping it sorted, the subtrees with larger trees expanding from them will more likely be tried first. Even though they are not sorted by the number of nodes expanded, one of the criteria we have used for sorting has been the depth that f-values were found. The deeper IDA* can go in a tree, the more likely the tree will be large. Thus by having the subtrees sorted, the processors will first work with the larger subtrees, and in the end work with the smaller subtrees, leading to a minimal amount of idle time at the end of an iteration.

It is possible that for some problems, CIDA* will have processors which go idle. This would be for cases in which only a few of the subtrees have most of the search tree expanding from them. For cases like this, one could then split the necessary subtrees to create a more balanced partitioning. As this was not a significant issue for the problems we applied CIDA* to, this has not been further investigated and is considered for future research.

6.3 Forced Deepening and Elite Paths

This next section describes two ideas which are not explicitly tied to CIDA*. The first of the two, Forced Deepening, is applicable to the class of combinatorial optimization problems which have solutions who all are permutations of one another. Examples of this are the quadratic assignment problem, the TSP and the TTP. The purpose of FD is to reduce the number of iterations IDA*-like algorithms go through for real-world distance problems. The second of the two, Elite Paths, was originally designed to improve the performance of FD, but has also been found to help find the optimal solution more quickly in the final iteration.

6.3.1 Forced Deepening

One of the problems when applying IDA*-like algorithms to problems which use real-world distances is that they can go through too many iterations. These algorithms work best when the number of iterations is few and the number of nodes being expanded each iteration grows exponentially.

To overcome this problem, we present a new technique, FD, which can be used to reduce the number of iterations. The general idea behind FD is to force the algorithm to find the next $f^t(I+1)$ value at a depth greater than the previous iteration. This helps drive the algorithm to skip iterations, cutting down the total number of iterations and can significantly reduce the running time to find the optimal solution.

Using the TSP as an example, when IDA* was applied to this problem [40], the algorithm would go through too many iterations for certain problem sets. These problem sets exhibited the same performance: very few new nodes were being expanded each iteration. If FD were used in conjunction with IDA*, then this would not happen. With a problem set composed of 20 cities for example, instead of going through the numerous iterations that IDA* would go through, FD would force IDA* to go through at most 20 iterations.

Implementation

Implementing FD requires few changes to IDA*. To begin with, a couple of new variables are needed. The first is the FD limit $\mathcal{L}$, which is the depth limit that a new $f^t(I+1)$ value must come from for the current iteration. The second is λ, which determines the rate that $\mathcal{L}$ grows.

In terms of $\mathcal{L}$ and λ, $\mathcal{L}$ is the minimal limit needing to be exceeded based on the depth that $f^t(I)$ was found at plus λ. λ is a positive integer which forces the depth to grow each iteration.

Generally with IDA*, when a node is expanded, it checks if its f-value does not exceed $f^t(I)$. If it does, it then updates $f^t(I+1)$ if $f(n) < f^t(I+1)$. With FD, this updating only happens if the depth of node n is greater than or equal to the current FD limit $\mathcal{L}$. If the depth of n does not equal or surpass $\mathcal{L}$, but it is less than $f^t(I+1)$, the algorithm will actually continue to construct the solution. It will only backtrack in the cases of where the f-value exceeds $f^t(I+1)$ or when the depth of n is at least equivalent to $\mathcal{L}$ and $f(n)$ is less than $f^t(I+1)$, allowing for $f^t(I+1)$ to be updated.

If $\mathcal{F}$ is the set of limiting thresholds seen by regular IDA* and $\mathcal{F}'$ is the set of limiting thresholds seen by IDA* with FD, then $\mathcal{F}' \subseteq \mathcal{F}$. The reason for this is that all limiting thresholds found by FD are the same ones that would be found by IDA*, as they are all based on f-values of the nodes. The difference is that for some applications, many of the thresholds will be skipped by FD. For some problem instances, it is possible that there will be no difference between $\mathcal{F}'$ and $\mathcal{F}$, such as if all distance costs are constant. It is then problem like these were FD would not be beneficial.

A drawback of this approach is that extra nodes will be expanded, including nodes which would generally not be expanded by regular IDA*. This is due to the algorithm needing to find the minimal $f^t(I+1)$ for the next iteration. Until it is found, the algorithm will expand extra nodes. For the problems that FD is designed for, the extra expansion of nodes is greatly mitigated by the reduction of iterations since nodes are re-expanded far fewer times.

Optimality with Forced Deepening

To ensure optimality, FD is not used during any iteration which the depth of the limiting f^t-value is equivalent to the depth of the tree. If FD were used during this iteration, then it would force nodes to be expanded which are goal nodes, even if such nodes were not optimal. Also, since the f^t-value was found at the same depth as the goal nodes, this then indicates that the iteration is the final iteration, which is another reason why FD is not used.

Even though FD is not used in the situation just described, for some problems it is possible that the final iteration takes place when the $f^t(I)$ was not found at the depth of the goal nodes. This is specifically the case when the heuristic estimate is accurate, thus the optimal solution cost is found without having to construct a whole solution. For example, when working with the TSP and using the minimum spanning tree as a heuristic estimate [24], when there is one city left to visit, the heuristic estimate can correctly calculate the cost of visiting that city and then visiting the first city in the tour. For the TSP, the final iteration would then take place at a depth one less than the height of the search tree, though for some instances it can take place even earlier.

One of the questions raised when using FD is whether the first solution found is still optimal. It is optimal, which is proven with the following theorem.

Theorem 3. *When FD is applied to IDA* on problems where all solutions are of the same length, then optimality is still guaranteed for the first solution found.*

Proof. Assume to the contrary that the first solution found is not optimal. Then there exists a solution whose value is less than the found solution. Since the current solution has the same value as $f^t(I)$, then there exists an optimal solution whose value is less than $f^t(I)$. This contradicts how IDA* works, as all remaining possible solutions (open nodes) have at least cost estimate $f^t(I)$ and the estimate is admissible, i.e. it underestimates the actual cost. Therefore, this contradicts the assumption that the first solution found is not optimal. □

Similar Approaches

Two similar past approaches to FD are DFS* [50] and IDA*_CR [42]. These two algorithms, which were presented at relatively the same time, combine IDA* with branch-and-bound. These algorithms were designed so that the limiting threshold was increased by an amount greater than what would generally be done by IDA*. This was done by sampling the search each iteration, and then determining a new limiting threshold for the next iteration. To guarantee optimality, once a solution was found, the algorithms then entered a depth-first branch-and-bound. This allowed the algorithms to either prove that the found solution was optimal or that there was another solution which was optimal.

Compared to these past approaches, FD is able to make better use of the f-values. All limiting thresholds are the same thresholds that would be found with IDA*, while the thresholds used by DFS and IDA*_CR are thresholds which are not necessarily found by IDA*. This gives a benefit of that for the final iteration, the first solution found by FD is optimal, while the other approaches need to perform a depth-first branch-and-bound search. If the estimated cost is poor, this can then lead to very poor performance since many extra nodes would be expanded.

One advantage that these past approaches have compared to FD is that they can work for any type of problem that IDA* is applicable to. As mentioned before, FD is restricted to a certain class of problems, resulting in this limit.

From a parallelism perspective, FD also has another advantage over these past approaches. When working with these past approaches, for the final iteration many solutions might have to be communicated between the slave processes and master process. This is more expensive than communicating new f^t-values, which consist of only the f-value and the depth it was found at.

6.3.2 Elite Paths

The next idea presented here, EP, was originally designed to help reduce the number of extraneous nodes expanded when using FD. What EP does is that for a subtree which finds a new $f^t(I+1)$ value, it records the path to that value and stores it with the subtree. Then at the start of a new iteration, should a subtree have an EP associated with it, it will re-construct the EP and start the search having already constructed the partial solution. In doing so, it allows IDA*-like algorithms to quickly find a low value for $f^t(I+1)$, which helps reduce the number of extra nodes expanded until the minimal $f^t(I+1)$ value is found.

A side effect later noticed when using EP was that it allowed for the optimal solutions to be found quickly in the final iteration. Since EP records the path to the node which found $f^t(I)$, it then recreates that path first. Should that node be at the depth of the goal nodes, then the first node expanded will be the solution. For problems where the final iteration is earlier than this as described above, then it is likely that one of the children of the EP will be the optimal solution.

In order for this to work, the EP must be the $f^t(I+1)$ node which was found at the deepest depth when multiple nodes share the same $f^t(I+1)$ value for a given subtree. If it chose a node at a more shallow depth, then more search would be required.

6.3.3 Combining CIDA* with FD and EP

Combining CIDA* with FD and EP is beneficial in both directions. CIDA* benefits from having to go through fewer iterations when FD is applied and from being able to find the optimal solution more quickly in the final iteration with EP. FD benefits from the sorting of the subtree forest. Subtrees which are more likely to contain lower f-values are tried first, which then allows for the minimal $f^t(I+1)$ to be found sooner. This is especially true when the minimal $f^t(I+1)$ value is not found in the subtree containing the EP. Having the subtrees sorted then helps to minimize the extraneous nodes being expanded until the minimal $f^t(I+1)$ value is found.

In regards to implementation particularities when combining CIDA* and FD, the subtrees' s^f values are updated whenever a node cannot be expanded due to its f-value exceeding $f^t(I+1)$. Keeping these values updated allows for the subtrees to be correctly sorted since the subtrees will better reflect the values they found during the search.

The drawback though of combining CIDA* and FD is that it increases the amount of communication for updating the $f^t(I+1)$ value, especially in a distributed memory approach. Whenever a subtree finds a lower value, it communicates this with the master process and the master process will then have to send this value to all other slave processes. Fortunately, in the experiments we have run during this research, these values were updated infrequently, thus it had minimal impact on the performance of CIDA*. More importantly, the cost of communicating pales in comparison to the cost of running the additional iterations if FD were not used, thus it is a minor trade off for the gains from using FD.

When combining CIDA* and EP, the EP are stored with the subtrees. Any subtree which had updated the $f^t(I+1)$ value will then have an EP stored with it in the priority queue. It is possible that multiple subtrees could contain an EP, such as when the $f^t(I+1)$ is updated multiple times by various subtrees. As the subtrees are already sorted, then subtrees with an EP are more likely to be processed in the beginning, thus there is nothing detrimental if more than one subtree has an EP. But as a caveat, it is not desirable for every subtree to have an EP due to memory limitations. This would be the case of every subtree updating the $f^t(I+1)$, which would happen if the subtree forest was poorly sorted.

In terms of EP and parallelization, the EPs are sent with the subtrees. In most cases, the EP will be empty for the associated subtree and would then have no impact on communication costs. Again as with the FD, while this will add a small overhead cost for the subtrees which do have an associated EP, the overhead is minimized by the cost savings from the benefits of using EP.

6.4 Initial Analysis with the Traveling Salesman Problem

Before looking at the TTP, we analyze the performance of the new ideas presented in this chapter on the TSP. The motivation for this is that the TSP is much easier to work with since it is easy to control the various features of the problem. When looking at the TTP, 4-team instances are too easy, 6-team instances are not very difficult, and 10-team instances are too difficult for testing purposes, leaving only 8-team instances for reasonable testing. With the TSP, we can create city sets of a larger range of sizes, allowing us to choose city set sizes which are not too easy to solve nor too difficult. This then provides a strong testing ground for our new ideas.

6.4.1 CIDA* Applied to the TSP

The settings of these tests are similar to when Enhanced IDA*[40] was tested on the TSP. They used a depth-first search for assigning cities and a minimum spanning tree as the heuristic estimate.

When applying CIDA* to the TSP, the subtrees will contain all permutations of s^d cities since the only constraint of the TSP is that cities can visited just once. The subtrees in $\mathcal{S}$ are sorted first by the depth that s^f was found at in descending order and then by the s^f value in ascending order. This causes the subtrees with the best chances of finding the optimal solution or minimal f^t values to be tried first along with allowing for efficient load balancing.

6.4.2 Subtree Forest

The first set of testing used 25-city problem instances in the [0-1000] coordinate set in order to analyze the affects of the subtree forest.

The first test is to compare the nodes expanded by CIDA*+FD+EP, both with subtree skipping and without subtree skipping, and to compare both variations against IDA*+FD+EP. Figure 6.2 shows the results of this set of tests. The city

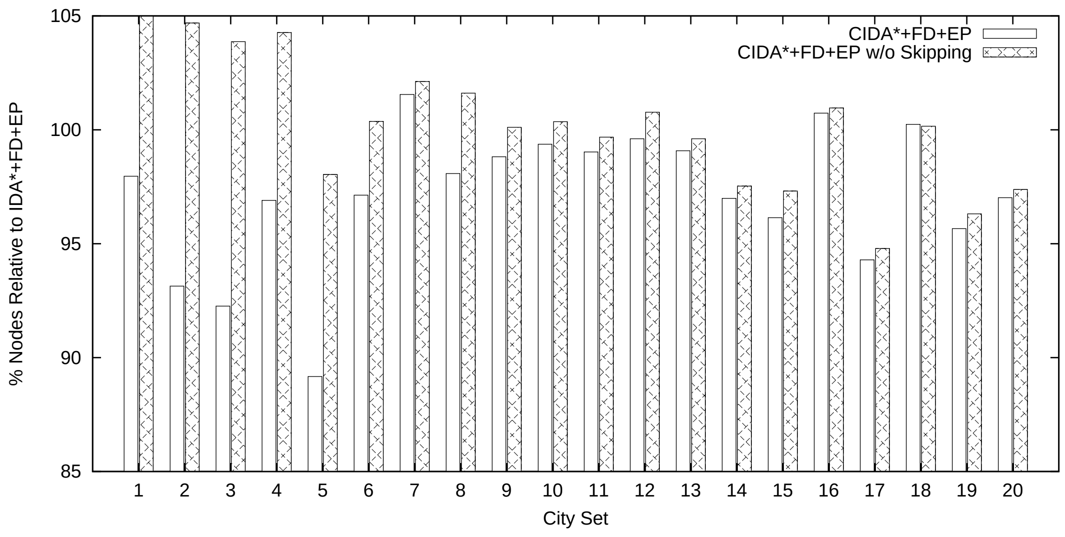


Figure 6.2: Comparing CIDA*+FD+EP, with and without subtrees skipping, against IDA*+FD+EP in terms of number of nodes expanded.

instances are sorted in ascending order of the number of nodes expanded when applying IDA*+FD+EP. With subtree skipping, CIDA* is able to expand fewer nodes than IDA* for all but three instances. The three instances where it expanded more nodes was likely due the overhead of using the subtree forest. Without skipping, the algorithm performs worse for the easier instances, but then performs relatively the same as with skipping. This is due to the more difficult instances have more uniform trees expanding across the subtrees, which gives greater chance of more subtrees having the same s^f value each iteration and lowering the chance of being able to skip subtrees.

We next look at the impact the different values of s^d can have on the performance of the algorithm. There were four tests done, using s^d values of [1-4]. The latter three values were compared with the first value on the same twenty problem instances as with the previous test. Figure 6.3 shows the results of these experiments. As seen, having a deeper depth of 2 or 3 results in fewer node expansions for most of the instances, even taking into account the additional overhead associated with deeper depths. Having $s^d = 4$ results in poor performance for the easier problem instances, with the associated overhead causing too many nodes to be expanded.

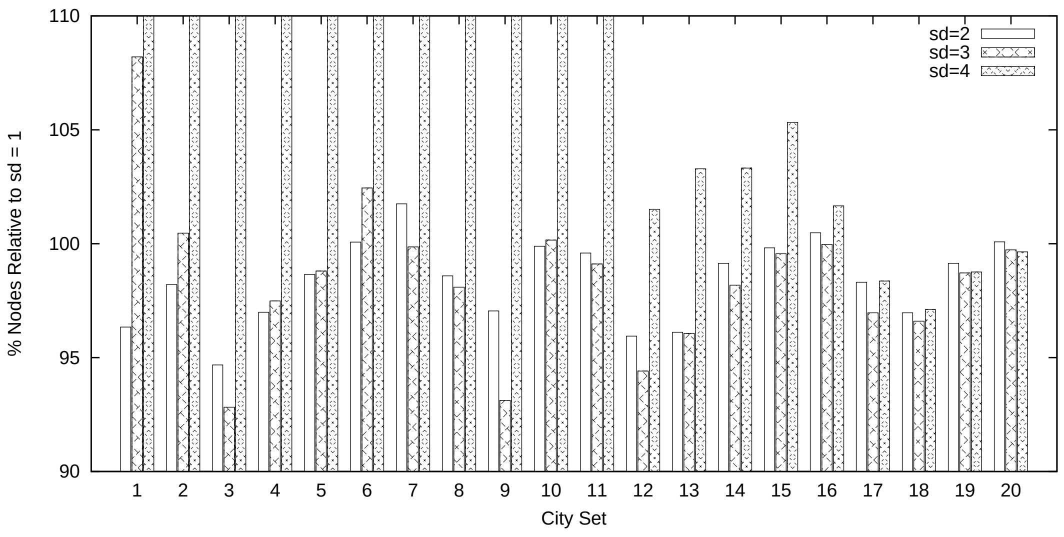
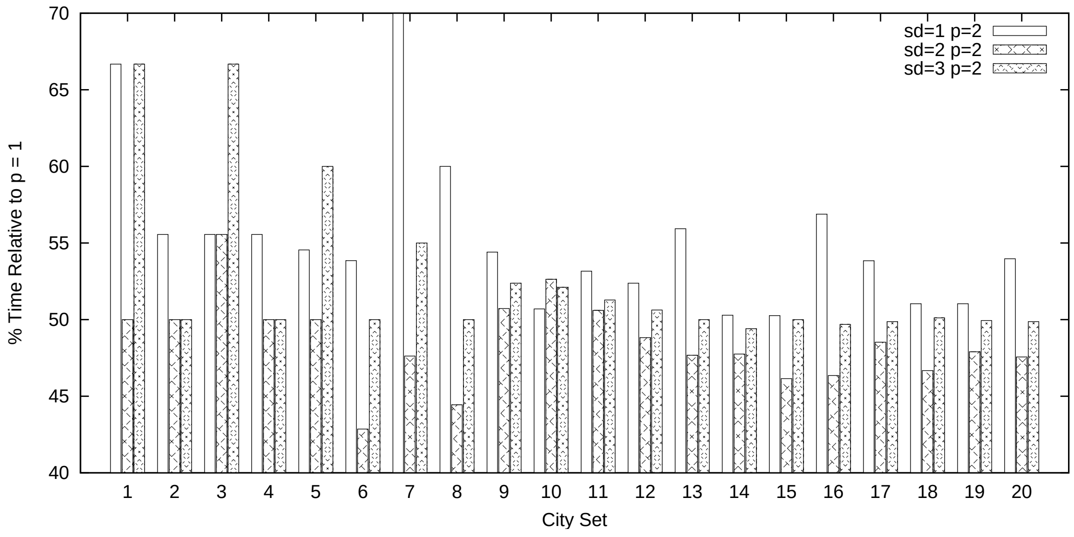


Figure 6.3: Comparing the effects of s^d with values of 2, 3, 4; compared to $s^d = 1$ in terms of number of nodes expanded.

This is mitigated as the problem instances become more difficult, since the savings from having a finer partition results in more subtrees being skipped which overcomes the extra nodes expanded from the subtree overhead.

The final testing with the subtree forest and subtree lengths is how it can influence parallelism, an important aspect of CIDA*. We did testing with s^d values in the range of [1-3] on 1, 2, and 4 processors. These tests were done again on the same twenty problem instances used for the previous two tests. Figure 6.4 shows the results of these tests. For the three values of s^d, running on 2 and 4 processors was compared with the running on 1 processor. As can be seen, for the easier problem instances, the performance varied alot due to the overhead of the subtrees and parallel processing. But as the problem instances became more difficult, the three values for s^d evened out, with reductions by about half for when 2 processors were used and a quarter for when 4 processors were used. What can also be seen is that the best results were when $s^d = 2$. This can be attributed to the best balance of problem decomposition and overhead from using subtrees, especially in regards to the overhead of communication between processors.



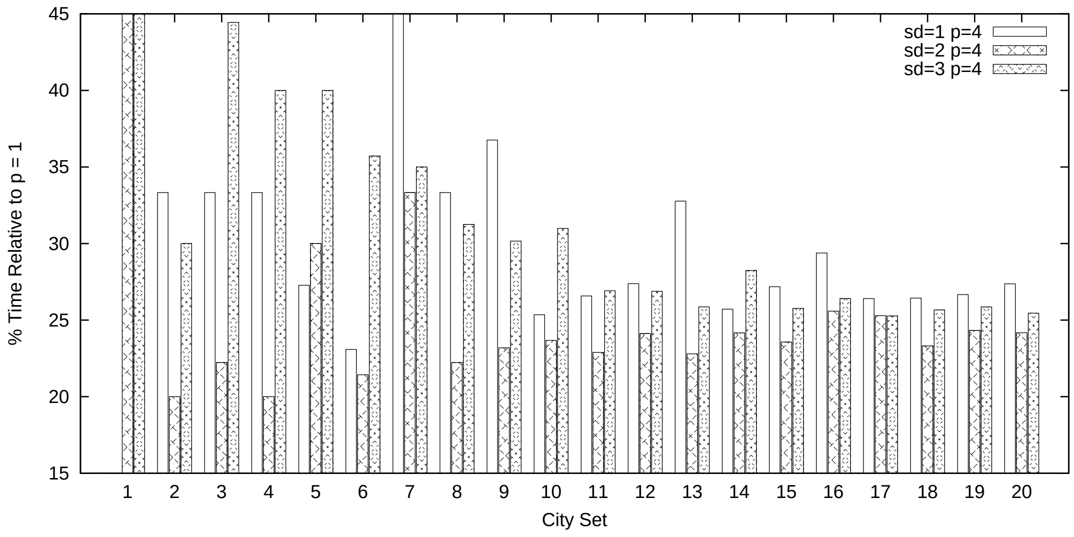


Figure 6.4: Comparisons of parallelism against 1 processor for s^d values of 1, 2 and 3, with comparisons looking at the amount of time needed to find the optimal solution. Top graph is for when 2 processors are used and the bottom graph is for when 4 processors are used.

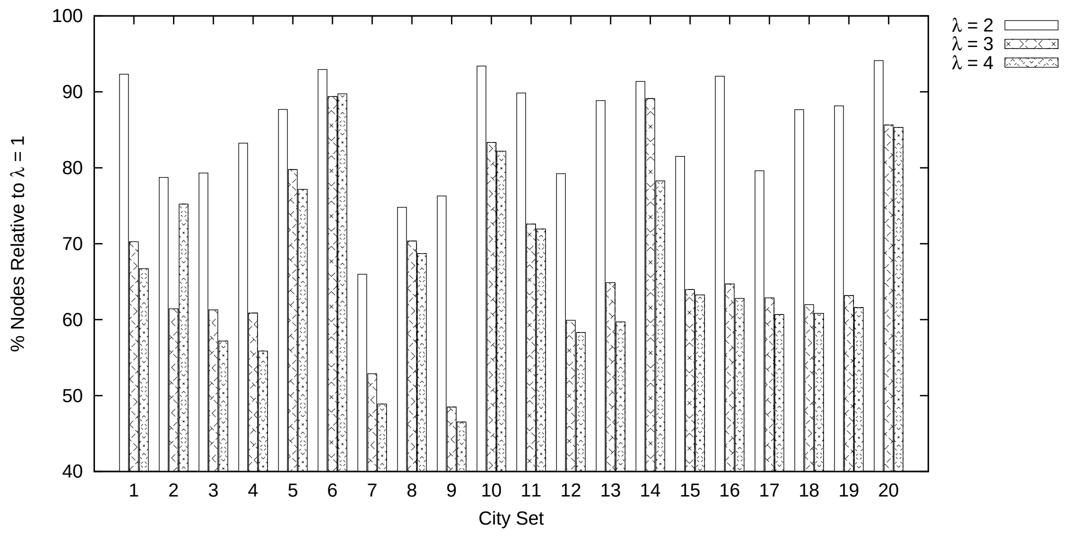


Figure 6.5: Comparisons of different values of increments for λ, compared to $\lambda = 1$ in terms of number of nodes expanded.

6.4.3 Forced Deepening and λ

The final aspect we check for is with FD and the affect of the increment of λ for calculating $\mathcal{L}$. We looked at increment values in the range of [1-4], testing on the same 20 problem instances. Figure 6.5 shows the results of these tests. The increment values of [2-4] were compared against an increment value of 1. As can be seen, increasing the increment value helped to reduce the amount of nodes being expanded, with a couple of instances of over 50% reduction. Having values of 3 or 4 showed little variation for the more difficult instances. This is due to the fact that most node expansion happens in the final iteration, thus eventually the number of lower iterations being passed over starts to have less of an affect on overall node expansion.

What is difficult to see here, but will become more evident when working with the TTP is that having too large of an increment can lead to worse performance. This is not much of an issue for these smaller TSP instances, but becomes an issue as the problem size grows.

6.4.4 Summary of Initial Results

These initial results of applying CIDA* to the TSP help to show how effective our techniques can be. CIDA* is able to reduce the number of nodes expanded when compared to IDA* for problems where subtree skipping can be exploited. It also offers an effective way of parallelizing IDA*. Looking at the size of the subtree forest, having too small or too large of a forest can decrease the effectiveness of subtree skipping and parallelization. Finally, the experiments for the increments of λ shows how larger increments can reduce the number of iterations, leading to less nodes being expanded in total. This shows the impact that FD can have on problems with real-world distances.

Chapter 7

Concurrent Iterative-Deepening-A* and the Traveling Tournament Problem

Genius is one percent inspiration, ninety-nine percent perspiration.

-Thomas Edison

This chapter discusses the application of CIDA* to the TTP. It first begins by showing the depth-first search process used for this application, which is similar to the depth-first search approach used when applying ACO to the TTP. The chapter then continues with team ordering; heuristic estimates and disjoint pattern databases; and symmetry breaking.

7.1 Depth-First Search

CIDA* applied to the TTP performs a depth-first search using backtracking search with constraint propagation. This approach is similar to our ACO approach. The key difference is that the ACO approach relied on techniques like unsafe backjumping, constraint propagation, and ant restarts, while this approach relies on just backtracking with constraint propagation. It also makes use of the subtrees idea along with implementing key concepts of IDA*.

The depth-first search used here continues to construct solutions in the same manner of pairing all teams for a given time slot prior to pairing teams in a subsequent time slot. Also, as with the ACO approach, every team $t \in T$ will have associated with it a domain for each time slot r of the schedule, D_r^t. In the beginning, each domain will contain all the opponents that a team can play both home and away, resulting in a domain size of $2(n-1)$.

Algorithm 8 explains the process of the depth-first search for applying CIDA* to the TTP. At the start, after initializing the domains, it applies the subtree being worked with and the EP if applicable. If this were IDA* instead of CIDA* being used, then the subtree would be an empty tree, allowing for the root of the tree to be the root of the search. Also initialized is i, the index of the depth-first search process. This is initialized to the depth of the subtree plus the depth of the EP. The last of the preliminary steps is to initialize the first team of a pairing for index i, t_1^i, to null, allowing the algorithm to know if it is moving forward or backward in the backtracking search.

Following these preliminary steps, the algorithm then enters the search phase, exploring the search tree within the limiting $f^t(I)$ value until either all domains have been exhausted or a solution has been found. At the start of each node, the algorithm will first calculate which time slot r is currently being worked on by applying the equation $\lceil \frac{2i}{n} \rceil$. Following this, if t_1^i is null, it will choose the first team of the pairing. How this is done is based on the ordering of teams, which is explained later in Section 7.3. Once t_1^i is not null, the algorithm will then attempt to pick a second team for the pairing, t_2^i. This will also be explained later in Section 7.1.1.

If the algorithm is able to choose a second team for the pairing, it will then progress further in the depth-first search process. If progressing would result in the index reaching $n(n-1)$, this indicates that a solution has been found. It will return this solution and finish operation, as the solution found will have been the optimal solution and no further work is needed.

If the algorithm is unable to choose a second team for the pairing, it will then

Algorithm 8 Constructing solutions

```
1: procedure $ConstructSolution$
2:     InitializeDomains
3:     ApplySubtree
4:     ApplyElitePath
5:     $solutionFound \leftarrow false$
6:     $i \leftarrow s^d + ElitePathLength + 1$
7:     $t_1^i \leftarrow \emptyset$
8:     while $i > s^d + 1 \wedge !solutionFound$ do
9:         $r \leftarrow \lceil \frac{2i}{n} \rceil$
10:        if $t_1^i = \emptyset$ then
11:            $t_1^i \leftarrow$ ChooseTeam1($r$)
12:        $t_2^i \leftarrow$ ChooseTeam2($t_1^i, D_r^{t_1^i}, r, i$)
13:        if $t_2^i \neq \emptyset$ then
14:            $i \leftarrow i + 1$
15:            if $i = n(n-1)$ then
16:                $solutionFound \leftarrow true$
17:                return $Solution$
18:            $t_1^i \leftarrow \emptyset$
19:        else
20:            $i \leftarrow i - 1$
21:            UndoPairing($t_1^i, t_2^i, i$)
22:    return $Subtree$
```

backtrack to the previous index, undoing the pairing. If backtracking results in the algorithm going to a depth lower than the depth of the subtree, or past the root of the search tree in the case of regular IDA*, the algorithm then knows it has done a complete search up to the limiting f^t-value. In the case of regular IDA*, it returns the empty subtree. In the case of CIDA*, it will return the subtree with appropriate values updated.

The reason why we constructed solutions in this manner, ignoring the programming ease with code sharing between this approach and the ACO approach, was due to simplicity of various aspects needed for CIDA*. First, this made it easy to propagate constraints, making it easier to check if the current solution was feasible or not. This includes the usage of the same pattern matching used for the ACO approach. Secondly, by constructing solutions one time slot at a time, this also made it easier to accurately calculate the current solution cost so far. Tied in with this is that it also makes it easier for calculating the heuristic estimate, as explained in Section 7.4. These two values are needed for calculating $f(n)$, so the more accurate the two are, the more accurate $f(n)$, allowing for better performance of the algorithms.

7.1.1 Choosing the Second Team

The next two algorithms presented represent how the second team, t_2^i, is chosen. The difference between the two is that the first algorithm is used for when FD is not applied and the second is for when FD is applied.

Starting with the first of the two algorithms which does not use FD, Algorithm 9, the algorithm will go through t_1's domain, trying to find a team in which a feasible solution can be constructed and whose resulting f-value is within the f^t-limit. When choosing opponents from t_1's domain, the algorithm will try all home teams prior to away teams. This was chosen arbitrarily, as experiments have shown that there is no difference between either choosing a home game or an away game first.

After a candidate team is picked, the algorithm will propagate the constraints associated with pairing the two teams. This consists of making sure the two teams

Algorithm 9 Choosing second team of pairing without Forced Deepening

```
1: procedure $ChooseTeam2(t_1, D_r^{t_1}, r, i)$
2:     while $D_r^{t_1} \neq \emptyset$ do
3:         $t_2 \leftarrow t \in D_r^{t_1}$
4:         $D_r^{t_1} \leftarrow D_r^{t_1} \backslash t_2$
5:         AddPairing$(t_1, t_2, i)$
6:         if PropagateConstraints$(i) = valid$ then
7:             $f(n) \leftarrow$ CalculateF$(r)$
8:             if $f(n) < f^t(I)$ then
9:                 return $t_2$
10:            else
11:                if $f(n) > f^t(I) \wedge f(n) < f^t(I+1)$ then
12:                    $f^t(I+1) \leftarrow f(n)$
13:                    UpdateElitePath
14:                if $f(n) > f^t(I) \wedge f(n) < s^f$ then
15:                    $s^f \leftarrow f(n)$
16:                UndoPairing$(t_1, t_2, i)$
17:    return $\emptyset$
```

are not paired again with any other teams for the current time slot; they do not play each other in regards of which is home and away for future time slots; and that for the following time slot the two teams do not violate the No Repeat Constraint and At Most Constraint. Also applied at this point is pattern matching, which is the same pattern matching as used with ACO.

If no constraints are violated and no domains are left empty, then the algorithm will update $f(n)$ and check if it exceeds $f^t(I)$. If it does not, it then returns t_2. Otherwise, if $f(n) < f^t(I+1)$ is true, it will update $f^t(I+1)$. Whether it updates it or not, the algorithm will then try another value from t_1's domain.

If the algorithm has tried all values in t_1's domain and fails to find any candidate t_2 that does not violate the constraints or $f^t(I)$, then that means the algorithm will have to backtrack and it returns null to indicate this.

The second of the two algorithms, Algorithm 10, is for when FD is applied. The key difference between this algorithm and the one just described is how f-values are checked and whether the algorithm can continue to construct a solution down the same path or not. If the f-value exceeds $f^t(I)$ but is less than $f^t(I+1)$ and the depth is less than $\mathcal{L}$, then the algorithm will return t_2 and continue constructing down this path. If the f-value exceeds $f^t(I)$ but is less than $f^t(I+1)$ and the depth is equivalent to $\mathcal{L}$, it updates the $f^t(I+1)$ and then tries a different value for t_2. For all other cases of the f-value exceeding $f^t(I)$, the algorithm will then act the same as if FD was not applied by returning null to indicate CIDA* needs to backtrack.

Algorithm 10 Choosing second team of pairing with Forced Deepening

```
1: procedure $ChooseTeam2(t_1, D_r^{t_1}, r, i)$
2:     while $D_r^{t_1} \neq \emptyset$ do
3:         $t_2 \leftarrow t \in D_r^{t_1}$
4:         $D_r^{t_1} \leftarrow D_r^{t_1} \setminus t_2$
5:         AddPairing$(t_1, t_2, i)$
6:         if PropagateConstraints$(i) = valid$ then
7:             $f(n) \leftarrow$ CalculateF$(r)$
8:             if $f(n) < f^t(I) \lor f(n) < f^t(I+1) \land i < \mathcal{L}$ then
9:                 return $t_2$
10:            else
11:                if $f(n) > f^t(I) \land f(n) < f^t(I+1)$ then
12:                    $f^t(I+1) \leftarrow f(n)$
13:                    UpdateElitePath
14:                if $f(n) > f^t(I) \land f(n) < s^f$ then
15:                    $s^f \leftarrow f(n)$
16:                UndoPairing$(t_1, t_2, i)$
17:    return $\emptyset$
```

7.2 Forced Deepening and Subtree Forest

A couple of aspects to be looked at is applying FD and the subtree forest to the TTP. With regards to FD, if $\lambda > 1$, then λ will change for the final time slot should $\mathcal{L}$ reach such a depth. There are two reasons for this. The first is so $\mathcal{L}$ does not reach a depth greater than the depth of the solutions. The second is because the heuristic estimates can correctly calculate the schedule cost by the time they have entered the final time slot without having to pair any teams in the final time slot. For the TTP, the value of λ will change so that $\mathcal{L}$ will be set to the final pairing of the second to last time slot should it exceed that, and afterwards λ will be set to one.

With regards to applying subtree forests to the TTP, the subtrees represent the first s^d feasible pairings with respect to the depth-first search used. For example, if $s^d = \frac{n}{2}$, then a subtree will consist of all n teams paired for the first time slot. The subtrees are also sorted in the same manner as with the TSP: they are first sorted in descending order by the depth that their s^f values were found at, and then ties are broken by sorting in ascending order by their s^f values.

7.3 Team Reordering

It is possible to reduce the number of nodes expanded by changing the order that the teams are chosen. This is both in terms of the first and second team of the pairing. With the second team, it will first trying go in a specified order for the possible home games, and then the possible away games.

The first choice of choosing teams is in the order that they are presented in the problem instances and the distance matrices. The second choice is by choosing the teams in order sorted by the minimal total distances to all opponents' venues. The third choice is by choosing the teams in order sorted by the maximal total distances to all opponents' venues.

With the last two possible reorderings, they are done at the start of the algorithm, rearranging the distance matrix to match the descriptions. After the algorithm is finished running, the solution is then re-sorted so that the scheduled teams match the teams of the original problem instance. By doing so, the algorithm does not have to do any extraneous computation checks during the running of the algorithm, as all computations will be done prior to the start and after the end of running the algorithm.

The reason for trying these possible reorderings is that teams which will have a larger impact can be tried first when pairing at the beginning of a time slot. This is similar in reason to the variable ordering used with the ACO approach to the TTP. Combined with ideas like FD and subtree skipping, team reordering has the potential to both allow for minimal $f^t(I+1)$ values to be found sooner, more subtrees to be skipped, and for a difference in the size of the search tree for each iteration.

7.4 Heuristic Estimate Calculations, Disjoint Pattern Databases, and Team Cache

We now look at the usage of heuristic estimates for the TTP. As required by any derived A* algorithm, the heuristic needs to be admissible, otherwise optimality cannot be guaranteed. For this application, CIDA* uses the ILB [16] for calculating heuristic estimates, which is admissible. To improve performance, instead of recalculating the heuristic estimates each time they are needed, they are kept in a disjoint pattern database [30] so they only need to be calculated once. An additional improvement is to use a team cache, which is similar to the usage of caches in processors.

7.4.1 Calculating Heuristic Estimates

The ILB is a heuristic estimate created for the TTP. It treats each team independently for calculating the estimate. It will first calculate the best possible schedule for an individual team, independent of the other teams' tours and its domains, then sum the individual estimates to obtain the total estimated cost. This estimate is used for this approach since it conforms well with the depth-first search presented in this paper. After CIDA* pairs a set of teams, it will only have to calculate the estimate for those two teams as it can reuse the estimates for the other teams due to the estimate independence.

When calculating the estimates for a team, it is possible to break symmetry within this estimate, allowing for the estimate to be calculated faster. The reason this is possible is due to the home and away trips caused by the AMC and due to the symmetrical distances. For example, working with an individual estimate, if it were to contain two away trips A and B, then there is no change in the distance traveled when either traveling through the teams in A, returning home, then traveling through the teams in B, or doing the trip in reverse such that one goes through the teams in B, return home, and then travel through the teams in A. Additionally, given an away trip of more than one game, there is no change in distance cost if traveling through an away trip in forward or reverse order of the teams listed for that trip. Both of these possible symmetry breaks are proven in the following theorems.

Theorem 4. *Let A and B be two away trips within a heuristic estimate. The order that A and B are visited will not change the distance of a heuristic estimate as long as distances are symmetrical.*

Proof. Let d_A be the distance for a team $t \in T$ traveling through the teams in A, including the distance for traveling between its home venue and the first and last teams in A, and let d_B be the distance for t traveling through the teams in B, including the distance for traveling between its home venue and the first and last teams in B. Traveling through A first and then B will result in the distance of $d_A + d_B$ while traveling through B first and then A will result in the distance of $d_B + d_A$. Since addition is associative, $d_A + d_B = d_B + d_A$. Therefore, the order that A and B are visited will not change the heuristic estimate. $\square$

Theorem 5. *Let A be an away trip composed of two or more teams for a heuristic estimate. Traveling through the teams in A in forward or reverse order will result in equivalent summed distances as long as distances are symmetrical.*

Proof. Let $t \in T$ be the team that the heuristic estimate is being calculated for, let $t_i \in A$ be teams being traversed through, and let a be the number of teams in A. Let d_F be the distance of traveling through the teams in A in forward order such that

$$d_F = d_{t,t_1} + d_{t_1,t_2} + \cdots + d_{t_{a-1},t_a} + d_{t_a,t}$$

Let d_R be the distance of traveling through the teams in A in reverse order such that

$$d_R = d_{t,t_a} + d_{t_a,t_{a-1}} + \cdots + d_{t_2,t_1} + d_{t_1,t}$$

Since distances are symmetrical, then all terms in d_F are symmetrical equivalences of the opposite terms in d_R, leading to $d_F = d_R$. Therefore, the summed distances of traveling through A in forward or reverse order will be equivalent as long as distances are symmetrical. $\square$

To break the first of the two described symmetries, teams are chosen such that the first team of each trip is in numerical order. Thus, if trip A come before trip B, then the first team in A will be numerically lower than the first team in trip B. This is only applicable to trips which are not extensions of the trip currently being scheduled in the partial-solution.

To break the second described symmetry, teams within a trip of two games or more are chosen so that the first team is numerically lower than the last team. If the trip consists of three games, there is no restriction on what team the middle team can be, since placing restrictions would prevent all possible estimates from being calculated.

Another symmetry in the heuristic estimates is with the home games. As there is no distance calculated for consecutive home games, called a home stand, it then does not matter in which order the length of home stands occur. Thus to help further break this symmetry, the length of home stands are done so that a later home stand is at most the same length or shorter than previous home stands.

7.4.2 Disjoint Pattern Databases

To improve performance when working with heuristic estimates, a disjoint pattern database is used to store all of the estimates. This is possible since each team's estimates are independent of the other team's estimates, allowing the estimates to be treated disjointly.

The disjoint pattern database used here does differ in its usage for its original application to the sliding puzzle. With its application to the sliding puzzle, the disjoint pattern database was created for puzzles of all the same size. For example, one database would be created and was then used for all possible 15-puzzle starting positions. With the TTP, the disjoint pattern database for one instance is not applicable to another instance of the same size. For example, the database for NL8 would not be applicable for CIRC8. Instead, it is possible to use the database within a problem set, since the database of one instance is the subset of the next larger instance. For example, the database for NL6 is a subset of NL8's database, which in turn is a subset of NL10's database.

Even though it is possible to build up databases, this is not explored further in this paper. The reason for this is that the creation of these databases for the TTP is not expensive in time compared to the time needed to solve a TTP instance. This is the opposite of what is seen with the sliding puzzle. The time to create such a database takes longer than solving the puzzle, and the costs of these databases were amortized across multiple puzzles.

These disjoint pattern databases are created at the beginning of the running of the algorithm. When working in parallel on a shared memory approach, there is only one database for all processors and it is possible to split up the work of creating this database amongst the various processors. When working in parallel on a distributed memory approach, then each processor will have its own database and will be required to populate its own database due to the expensive cost of communication.

What follows now is the number of estimates needed for each team's pattern database. What is stored at each index is only the heuristic estimate. These estimates are indexed by five possible dimensions. The first two is the number of remaining away games and the number of remaining home games. The next is the set of teams it has to play away, which is directly related to the remaining number of away games. Another dimension for these estimates is either the number of the previous consecutive away games or the number of previous consecutive home games. Only one of the two is used, since it is impossible for a team to have played both a previous home game and away game. The final dimension, in the case of the last game played being an away game, is the the previous team played. For the following definitions which follow, $\mathbb{N} = n - 1$ is defined as the number of opponents.

Beginning from the top, the total number of estimates for one team is defined as:

$$\mathbb{H} + \mathbb{A} \tag{7.1}$$

with $\mathbb{H}$ being defined as the number of estimates needed when the previous game was a home game and $\mathbb{A}$ being defined as the number of estimates needed when the previous game was an away game. Delving further into this, when the previous game was a home game, this then results in:

$$\mathbb{H} = \sum_{k=1}^{3} \sum_{l=1}^{\mathbb{N}-k} \mathbb{C}^h \tag{7.2}$$

with the first summation due to the possible number of previous consecutive home games, [1-3], as restricted by the AMC. The second summation is due to the possible number of remaining home games, which ties in with the previous number of consecutive number of home games. $\mathbb{C}^h$ is the total number of estimates from the possible combinations of teams it can play away after having played a home game previously. With these sets, order and duplicates are ignored. $\mathbb{C}^h$ is then further defined as:

$$\mathbb{C}^h = \sum_{i=1}^{\mathbb{N}} \binom{\mathbb{N}}{i} \tag{7.3}$$

with the summation due to the total number possible of remaining away games. The second aspect is from the choosing of i possible teams from the total number of opponents that the team being looked at can play away.

Now that we have defined when the previous game was a home game, we next define when the previous game was an away game, which gives us the equation of:

$$\mathbb{A} = \mathbb{N} \cdot \mathbb{N} \cdot \mathbb{C}^a \tag{7.4}$$

with the first of the two $\mathbb{N}$s being due to the possible teams that the team could have played in the previous time slot and the second of the two $\mathbb{N}$s being due to the possible number of home games remaining. Both of these are related to the possible number of opponents a team has. The third aspect in the equation, $\mathbb{C}^a$, is similar to $\mathbb{C}^h$, but now is the total number of estimates from the possible combinations of teams a team can play away after having played an away game in the previous time slot. Again as with before, order and duplicates are ignored. This then results in $\mathbb{C}^a$ being defined as:

$$\mathbb{C}^a = \sum_{k=1}^{3} \sum_{i=1}^{\mathbb{N}-k} \binom{\mathbb{N}-1}{i} \tag{7.5}$$

with the first summation due to the number of possible previous consecutive away games, [1-3]; the second summation from the possible number of remaining away games, which relates to the number of previous consecutive away games; and the final aspect again due to the choosing of i possible teams from the total number of possible teams the team being looked at can play away. With this last aspect, since we have already counted one of the away teams in the previous definition, this then reduces the choosing from $\mathbb{N}$ to $\mathbb{N}-1$.

When putting these estimates together, this then results in at most $\mathcal{O}(n^3 n!)$ number of estimates needed for a single teams pattern database. When looking at the whole disjoint pattern database, this would result in an upper limit of $\mathcal{O}(n^4 n!)$ estimates.

The disjoint pattern database defined here is suitable for up to 14 teams when using 2GB of memory. When working with more than 14 teams, fitting all estimates into memory would then require more relaxations of the ILB. Possible relaxations could be to relax the indexing of the remaining number of home teams.

An alternative would be to dynamically create the database while estimates are being calculated during the running of the algorithm, similar to the work by Felner and Adler [18]. Once memory has become full, then no more new estimates would be stored in the database. Any estimate not already stored would have to be recalculated every time it is seen. Fortunately, in such a case, these estimates would probably be smaller estimates which occur deeper in the search, and as such they are faster to calculate than those seen in the beginning of the search. Since the largest team set explored by CIDA* in this work is only 10 teams and with CPU being the current limiting factor, this limit of memory is not further explored and is left for future research.

7.4.3 Team Cache

The disjoint pattern databases help to eliminate the need of recalculating heuristic estimates. But one of the weaknesses of them is that it is still expensive to calculate the index of where an estimate is located, though not as expensive as calculating the heuristic estimate itself. A way to mitigate this is to keep a cache of the estimates for each team, which we call a team cache. For each time slot, a cache will be kept for each team that will include the estimates valid for their schedule up to the previous time slot. Since each team's estimates are independent of the other teams, their own estimates will not change unless the pairing of the previous time slot has changed.

The improvement in running time comes from the cheaper cost of looking up in a table instead of calculating the index in the disjoint pattern database. Calculating the index is $\mathcal{O}(n)$ while looking up in the table is $\mathcal{O}(1)$.

The information kept in the table for team t for a given time slot consists of all the estimates for all possible matches it could play in the time slot. These estimates are based on t's pairing of the previous time slot. Thus, until the previous time slot pairing for t has changed, the current time slot estimates will remain the same.

This gain is further amplified by the fact that the team orderings are fixed. During the search, there will be multiple times a team has to try all possible pairings for a given time slot before it has changed its previous time slot. This then gives it the possibility of reusing its estimates up to $\mathcal{O}((n-2)!)$ times.

The reason why this team cache is applicable to the TTP and not to problems like the TSP is simply due to the round robin structure of the TTP along with the order of the depth-first search, pairing up teams in one time slot prior to pairings teams in a subsequent time slot. Also, since the teams are picked in a predetermined order, this then allows for the estimates to be reused many times, making it worthwhile to

Team	1	2	3	4	5	6
1	+3	+2	+4	-3	-2	-4
2	+4	-1	-3	-4	+1	+3
3	-1	+4	+2	+1	-4	-2
4	-2	-3	-1	+2	+3	+1

Team	1	2	3	4	5	6
1	-4	-2	-3	+4	+2	+3
2	+3	+1	-4	-3	-1	+4
3	-2	-4	+1	+2	+4	-1
4	+1	+3	+2	-1	-3	-2

Figure 7.1: Symmetric schedules with equivalent total distances. Both are optimal for NL4.

use the team cache.

7.5 Symmetry Breaking

With the TTP, there are two types of symmetry present. One of these is found in all problem instances with symmetrical distances called reflective symmetry [25, 26]. The second is tied with the CIRC instances, in that they exhibit a rotational symmetry [26]. Both of these ideas have been adapted to work on our depth-first search approach.

7.5.1 Reflective Symmetry

With all problem instances of the TTP, there is a reflective symmetry present. This means that for all problem sets, half of the solutions in the solution space are reflections of the other half of solutions.

This reflective symmetry is due to the symmetrical nature of the distances used between teams along with the way distances are calculated for the problem set. An example of this can be seen in Figure 7.1. When looking at the tours for a single team between both schedules, they are reflective of each other. With each team, its tour can be reflected across the midpoint of the schedule. Since all distances are symmetrical, the distances for each tour will be equivalent. And since every tour for each team has equivalent distances, then both schedules have equivalent total distances.

To break this symmetry, we treat the first team of the schedule as a pivot. When half of its schedule has been constructed, the algorithm checks if the number of remaining home games is greater than the remaining number of away games, or if the opposite is true. The former check is called Symmetry-Breaking-H, while the latter is called Symmetry-Breaking-A. Only one of these checks is used throughout the running of the algorithm, since using both would result in no solutions being constructed.

The motivation behind having two possible checks is that they can have an impact on the number of nodes expanded. This is through the heuristic estimate, specifically for the pivot team. The two checks will cause the first half of the schedule to have either more home or away games played, and this can impact the accuracy of the heuristic estimate for the pivot team. Therefore, both checks are considered and are tested later in Section 8.1.5.

The reason these checks work is that when half of a solution is constructed, there will be $n - 1$ games remaining left to play. This is always odd since n is always even, which results in either the remaining number of home or away games being an odd number while the other is even. Thus by checking that one is greater than the other, we are able to eliminate half of the solution space. For example, if using Symmetry-Breaking-A, all solutions that are created will have a greater number of home games played during the first half of the schedule for the first team, while those which have a greater number of away games in the first half for the first team will be skipped.

Constraint propagation is used to further improve this check. During the first half of the schedule, after every pairing for the first team, the algorithm propagates to make sure there are still teams available in the remaining first half of time slots so that the constraint will not be violated once half of a schedule has been constructed. This then helps to reduce any extra number of node expansions while checking for this constraint.

Team	1	2	3	4	5	6
1	+3	+2	+4	-3	-2	-4
2	+4	-1	-3	-4	+1	+3
3	-1	+4	+2	+1	-4	-2
4	-2	-3	-1	+2	+3	+1

Team	1	2	3	4	5	6
1	-3	-2	-4	+3	+2	+4
2	+4	+1	+3	-4	-1	-3
3	+1	-4	-2	-1	+4	+2
4	-2	+3	+1	+2	-3	-1

Team	1	2	3	4	5	6
1	0	0	0	2	1	2 (3)
2	0	1	2	1	2	0
3	2	2	0	0	1	2 (3)
4	2	1	2	1	0	0

Team	1	2	3	4	5	6
1	2	1	2	1	0	0
2	0	0	0	2	1	2 (3)
3	0	1	2	1	2	0
4	2	2	0	0	1	2 (3)

Figure 7.2: Example of rotational schedules found in CIRC. Top schedules are the pairings, while the bottom schedules represent the distance traveled. Numbers in brackets are the total distance traveled for the last index when taking into account the distance to return home.

7.5.2 Rotational Symmetry

The second symmetry found in the TTP is in the CIRC instances. Since their distances are the minimal number of arcs to their neighbors, each team will have the same distance matrix as every other team. This then allows for rotational symmetry to take place in the CIRC instances, and by breaking this symmetry, we can significantly reduce the search space for CIRC instances. Additionally, this rotational symmetry breaking can be combined with the reflective symmetry breaking, making CIRC easier to solve.

Figure 7.2 shows this rotational symmetry. Looking at the top schedules, the opponents played in each tour remain fixed for both schedules. But what rotates is the locations those games are played in terms of playing either home or away. This is rotated down between the two schedules, with the bottom rotating back to the top tour.

When looking at the bottom two schedules, this represents the distance traveled for each time slot, taking into account the location at the previous time slot. When comparing the two, the distances are also rotated down once each tour, with the bottom distances rotated to the top. If we were to keep rotating the home and away locations, the distances traveled would also continue to rotate. This is due to how the distances are defined for this problem. Because of this, it is possible to break the symmetry by a factor of $n - 1$.

For this application, this rotational symmetry breaking is done by using the subtrees as unique starting positions for the initial time slot. After the first subtree is created, every subtree created will check to see if its set of pairing distances is a rotation of any previously created subtree. If it is, then it is ignored, otherwise it is included in $\mathcal{S}$. In the end, $\mathcal{S}$ will only contain all the unique set of pairings for the initial time slot.

An alternative to this for when subtrees are not used is creating a table containing a set of unique pairings. While constructing the first time slot, if the pairings match the pairings in the table, then continue on. Otherwise, the pairings need to be changed since the current set is a rotation of one of the unique sets.

Chapter 8

Concurrent Iterative-Deepening-A* Results

Thunder is good, thunder is impressive;
but it is lightning that does the work.
-Mark Twain

This chapter looks at the results of applying CIDA* to the TTP. It is split into two sections. The first section looks at the performance of the different aspects presented in this work. The second section looks at comparing CIDA* with the best past approach along with showing new results.

All of these experiments were done on two sets of computers. The first were a set of compute servers with 4 parallel processors running at 2.4 GHz each, sharing 3GB of memory. This then gave us a shared-memory architecture type to work with. The second set were networked computers, with groups of 60 networked computers where each computer was composed of two cores running at 2.4GHz and having 3GB of memory. This allowed for a distributed memory approach, with 1 master and 119 slave processors. Due to some difficulties with the hardware and software, the master process had its own processor, even though it was idle for most of the running time of the algorithm.

For all experiments described in this chapter, disjoint pattern databases, team cache, and pattern matching were used unless otherwise specified.

8.1 CIDA* and the TTP

This section looks at the performance results of applying CIDA* to the TTP. It is split into five sections: looking at CIDA*, FD, and EP on the TTP; looking at FD and λ; looking at subtrees and load balancing; looking at disjoint pattern databases and team cache; and finally looking at team ordering and symmetry breaking. Due to a short time window for using the large set of networked computers, all of these experiments were run on the smaller compute server.

Unless otherwise specified, experiments were performed on the six- and eight-team sets of instances from NL, CIRC, SUPER, and GALAXY. Four-team instances were not used since they are too easy to solve, and ten-team instances were not used since they are too difficult for testing purposes.

8.1.1 IDA*, CIDA*, FD, EP, and the TTP

The first tests done with CIDA* on the TTP were comparing IDA* and CIDA*, with and without FD and EP. When CIDA* was run, the subtrees' depth s^d were set to $\frac{n}{2}$, which is the number of pairings in a time slot. When FD is applied, the incremental λ value is set to one.

The results of these first experiments can be seen in Figure 8.1. For these tests, IDA*+FD, IDA*+FD+EP, CIDA*, CIDA*+FD, and CIDA*+FD+EP were compared against IDA*, examining the number of nodes expanded.

As can be seen, CIDA* alone was able to slightly improve upon the performance of IDA* for all instances, reducing the number of nodes expanded and with the best improvement seen on SUPER6 due to its subtree skipping. When looking at FD, both for its application to IDA* and CIDA*, it greatly decreased the number of nodes expanded for all non-CIRC instances. The reason it was not able to decrease the node expansion for CIRC instances is due to the uniform artificial distances used,

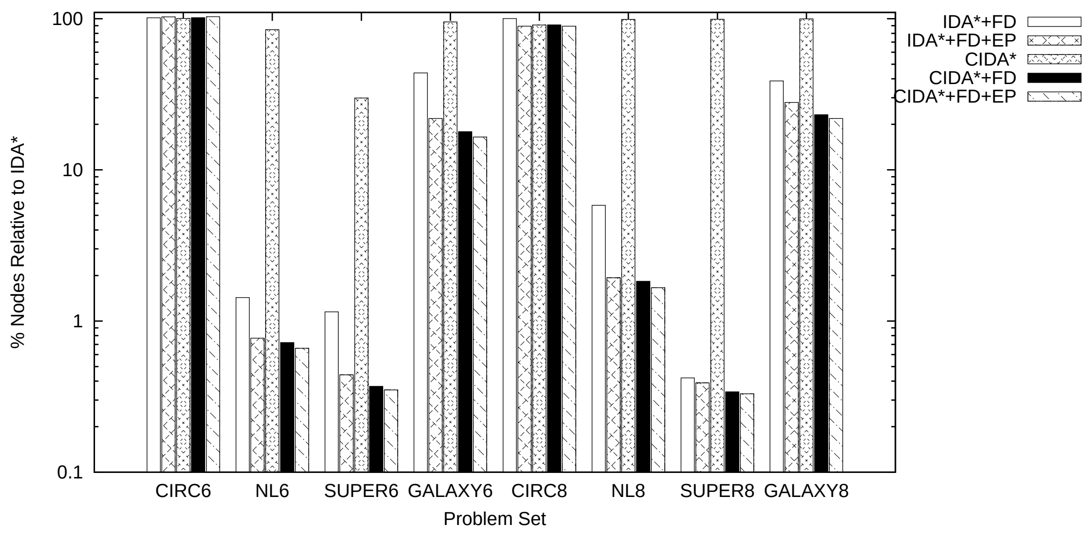


Figure 8.1: Comparison of nodes expanded using IDA* and CIDA*, both by themselves and with FD and EP. All comparisons are done against IDA* by itself.

and as such the algorithm would go through the same number of iterations both with and without FD. With the three problems sets composed of real-world distances, the savings of using FD is significant, with FD on some instances expanding less than 1% of the nodes expanded by IDA* alone. Looking further, when comparing IDA*+FD and CIDA*+FD, CIDA*+FD expanded fewer nodes for all instances. Applying EP helped to further reduce the nodes expanded for most instances when applied to either set of algorithms.

For a better understanding of CIDA* and IDA*, we compare CIDA*+FD+EP against IDA*+FD+EP, with CIDA*+FD+EP running with subtree skipping as commonly done and without subtree skipping. These results are shown in Figure 8.2. First seen is that with the CIRC instances, there is no difference between these approaches, again due to its artificial distances. But with the other instances, CIDA* was able to reduce the number of nodes when compared with IDA*. With regards to subtree skipping, the results help to show that subtree skipping does help to further reduce the number of nodes expanded. Without it, the overhead of using subtrees can actually cause CIDA* to expand more nodes than IDA*, as seen in the SUPER instances. The reason for this is that the nodes within the subtree have to be expanded

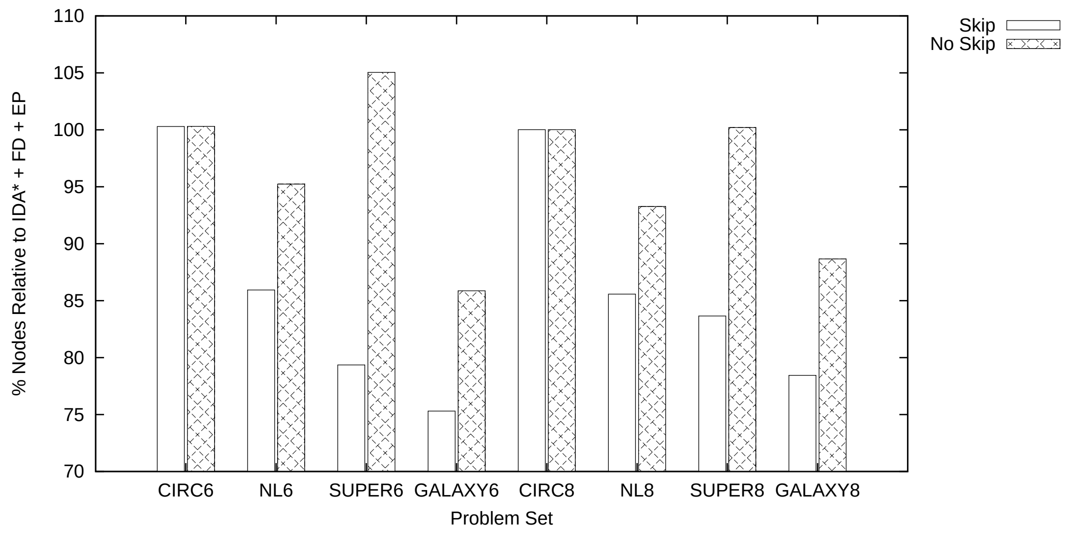


Figure 8.2: Comparison of nodes expanded with and without subtree skipping for CIDA* with FD and EP. This is compared in relation with number of nodes expanded by IDA* with FD and EP.

to ensure that constraints are propagated and the heuristic estimate can be properly calculated. Since the subtrees share many nodes, these nodes will be re-expanded more times than what is seen in regular IDA* search.

8.1.2 Length of Forced Deepening's λ

The next set of experiments look at FD and its incremental value λ. This increment was tested with value of 1, 2, $\frac{n}{2}$, n, and $\frac{2n}{3}$. All values were compared when λ was set to 1.

The results of these experiments are displayed in Figure 8.3. As can be seen, having larger values for λ allowed for more iterations to be skipped, which in turn reduced the nodes expanded greatly for all instances except CIRC. When looking at the results, the best balance of node reduction was seen when $\lambda = n$. For most instances, it expanded fewer nodes than increments of 2 and $\frac{n}{2}$. At the same time, it had similar node expansion reduction when compared to $\frac{2n}{3}$, expanding fewer nodes for NL, the same for SUPER, and did worse on GALAXY6 but did better on GALAXY8. Since 8-team sets are more difficult than 6-team sets, we then chose to go with $\lambda = n$.

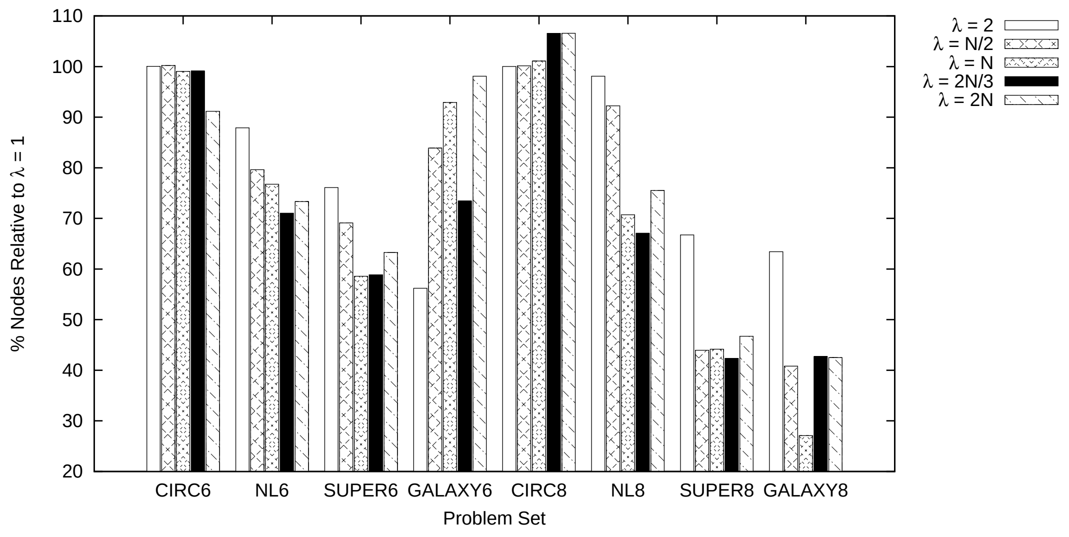


Figure 8.3: Comparing different incremental values for λ for $\mathcal{L}$ when using FD in terms of number of nodes expanded. All are compared against increment of 1.

The reason we believe that $\lambda = n$ exhibited such results was that it allowed for many iterations to be skipped, but at the same time, it did not jump ahead too much each iteration. Jumping ahead too far can make it more difficult to find the minimal $f^t(I + 1)$ value each iteration, causing extraneous node expansions.

8.1.3 Subtrees and Load Balancing

The next set of experimental testing is with the subtree forest, examining the impact that the length of s^d has on the node expansion rate of CIDA* along with its impact on parallelism.

The first set of experiments were performed with s^d values of 1, $\frac{n}{2} - 1$, $\frac{n}{2}$, and $\frac{n}{2} + 1$. The reason for these values is that they scale up as the problem size increases. The results of this initial set of tests is shown in Figure 8.4, with the latter three experimental values compared against $s^d = 1$. As shown, CIDA* performed best on average when $s^d = \frac{n}{2}$, which is the number of pairings in a time slot. Even though this value does not result in the least number of nodes expanded for all instances, it also does not have the most number of nodes expanded, for example with SUPER8. We believe this is due to it having a better balance between the overhead of using more

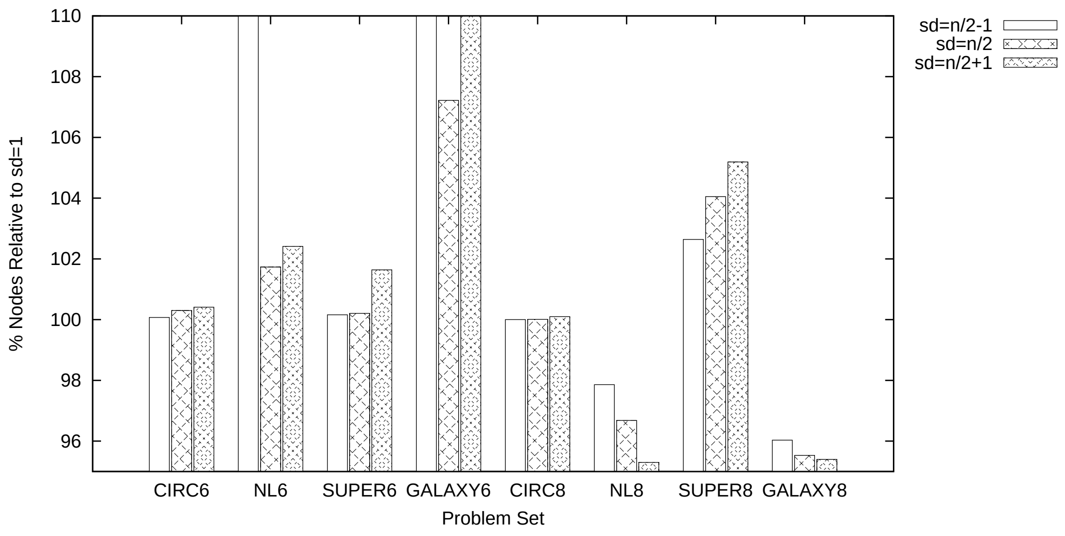


Figure 8.4: Comparing different lengths of s^d, with comparisons against $s^d = 1$ in terms of number of nodes expanded.

subtrees against the gains from partitioning the search space into finer partitions.

Still looking at Figure 8.4, it can be seen that for some instances, especially the 6-team instances, having a subtree depth of just one gave the best performance. We believe this is due to the decrease of overhead from subtrees, which are amplified for these instances since they are small instances. When working with the 8-team instances, the advantages of using a depth of one decreases significantly, with the only instance showing fewer node expansion being on SUPER8. One of the problems not shown though in this figure is that using a depth of one would would lead to poor performance in terms of parallelism, which is explored next.

The next set of experiments deal with subtree forests and load balancing when running CIDA* in parallel on four processors. This is done with subtrees depths of 1 and $\frac{n}{2}$ for comparisons. Table 8.1 shows the results of this test, with all times listed in seconds. These tests were only performed on the 8-team instances since the 6-team instances are too small for parallel testing. For all instances, CIDA* is able to find solutions faster when using a deeper subtree depth, which results in more subtrees. This is due to the load balancing. When running with $s^d = 1$, the processors would become idle, with the greater idleness coming during the second-to-last iteration. An

Table 8.1: The time needed, in seconds, to solve 8-team instances when looking at the depth of subtrees and its impact on problems in parallel.

s^d	CIRC8	NL8	SUPER8	GALAXY8
1	100	92	232	157
$\frac{n}{2}$	89	77	206	127

example of this is with SUPER8. When running with $s^d = 1$, the first processor finished at 191 seconds and the last processor finished at 232 seconds during the second to last iteration. But when running with $s^d = \frac{n}{2}$, there was much greater load balancing so no processors were left idle at the end of each iteration.

These SUPER8 timings also help highlight the strength of EP. The finishing time of the second-to-last iteration of 232 seconds is the same time needed to find the optimal solution. This is a result of the EP allowing the algorithm to quickly find the optimal solution on the final iteration. These results are also seen with the other instances tested here.

Another comparison is the efficiency of running four processors in parallel using $s^d = \frac{n}{2}$ compared to that of running with a single processor. These results are shown in Table 8.2. CIDA* running in parallel on four processors is able to reduce the running wall time down to 19.75% – 22.47% of the time needed when running on only a single processor. The fact that it took less than a quarter of the time needed by a single processor can be attributed to parallel speed-up anomalies [45]. Parallel speed-up anomalies are instances where solving a problem, with IDA* for example, can be done faster in parallel than what would be expected when dividing the time needed by the single processor by the number of processors. For example, if solving a problem with two processors, one would expect the time to be about half of that when solving with a single processor. But when there are parallel speed-up anomalies, the time will then be less than half. In the case of our results, where the instances on four processors were solved in less than a fourth of the time needed for a single processor, this then indicates the total number of nodes expanded with four processors is less than what is expanded with a single processor. This is due to the finding of minimal $f^t(I+1)$ values for each iteration sooner so fewer extraneous nodes

Table 8.2: The time needed, in seconds, to solve 8-team instances when comparing number of processors used.

Processors	CIRC8	NL8	SUPER8	GALAXY8
1	436	405	1014	653
4	89	77	206	127

are then expanded. The running time improvements seen with the results are better than what was achieved when running on the TSP, and helps to show the strength of running CIDA* on difficult combinatorial optimization problems.

We did one final testing where we ran CIDA* on a distributed computer using GALAXY8 as a test case. For this, the performance did degrade as the number of CPUs increased. We believe this is due to the small solution space for the 8-team instances. The processors are able to process a subtree quickly, which leads to the processors overloading the master processor with subtree requests. We expect this to be a non-issue for larger instances since subtrees take significantly longer to process, and as such the messaging and number of processors would not limit this aspect of parallelism.

8.1.4 Disjoint Pattern Databases and Team Cache

This set of tests begins to look more at the new ideas presented for the TTP instead of the new ideas for CIDA*. The first to be tested is that with the disjoint pattern databases and team cache. We did not do any testing per se on how much time the disjoint pattern database saves us, as this has been done before with the work on the sliding puzzle [30]. What we are more interested for this problem is how long it takes to create these disjoint pattern databases for the TTP. For 6-team instances, the time needed is too small to measure. For the 8-team instances, it takes between 2.24 – 3.42 seconds to construct with a single processor. For the 10-team instances, it takes between 241 – 452 seconds. There is a range of times due to different instances requiring different amounts of time to calculate the heuristic estimate. What this does help show though is that the time needed to construct these databases is far less than the time needed to solve the instances, which can be seen when looking at the best times reported in Section 8.2.2. This is the opposite of what is seen when working with the sliding puzzle, where the time to construct the disjoint pattern database takes longer than the time to solve a puzzle, and the time then has to be amortized across multiple puzzles.

With respect to the team cache, it generally helps to decrease the running time, with greater savings seen as the problem size increases. On the 6-team instances, using the team cache had mixed results, which can be attributed to the instances taking less than a second to solve and as such it is more difficult to get an accurate measurement. With the 8-team instances, the team cache reduced the running time for all instances, with a reduction down to between 87.95% – 89.68% of the time needed for when CIDA* is run without the team cache. We do expect these savings to increase as the instances become larger, since there is more possibility to reduce the number of lookups needed.

8.1.5 Team Ordering and Symmetry Breaking

The final set of tests to examine is those involving team ordering and symmetry breaking. These two sets of tests were combined since they have an impact on each other's performance. Symmetry-Breaking-A and Symmetry-Breaking-H were tested with the normal ordering of teams, with teams sorted by the maximal distance from the other teams, and with teams sorted by the minimal total distances from other teams. All of these tests were compared with CIDA*+FD+EP without symmetry breaking or team reordering. It is important to note that for the CIRC set, there is no difference with team reordering, as distances are the same for all teams.

The results of these tests are shown in Figure 8.5. As shown, Symmetry-Breaking-A with maximal distance team reordering on average resulted in the least amount of nodes being expanded. It does do worse for SUPER6 and GALAXY6 compared to without reordering, but better or at least equivalent for all other instances. More importantly, it does much better on the GALAXY8 instance when compared with Symmetry-Breaking-A without any team reordering.

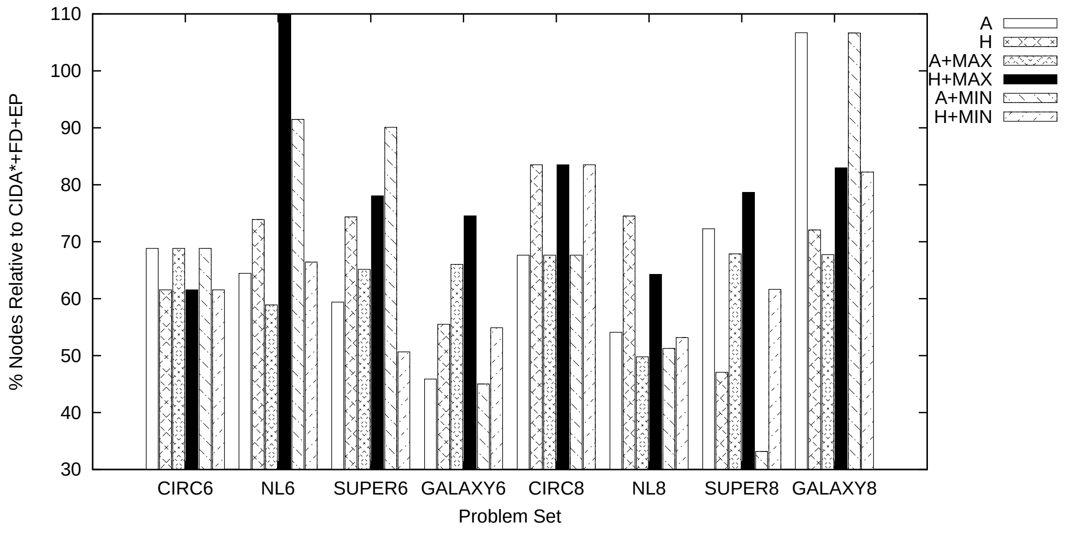


Figure 8.5: Comparison of Symmetry Breaking and Team Ordering. The various options, applied to CIDA*+FD+EP, are compared against CIDA*+FD+EP without any of these options in terms of number of nodes expanded.

An anomaly seen in these results is with SUPER8. As previously described, SUPER has the teams in two clusters distanced far apart. With SUPER8, three of the eight teams are from the smaller cluster. With the AMC restricting the maximal number of consecutive games to three, then all of the other teams will make the same three-game away trip through the three far-distance teams. This helps explain why CIDA* expanded fewer nodes on these instances when teams were sorted by the minimal distances from each other since it then ordered first by the larger cluster prior to the smaller cluster. We do not expect to see these type of results with SUPER10 and larger instances, as these instances have at least four teams from the smaller cluster.

The final symmetry breaking to concern about is the rotational symmetry breaking for the CIRC instances. When combined with Symmetry-Breaking-A, CIRC6 expanded 13.04% and CIRC8 expanded 8.38% nodes compared without using symmetry breaking, while combined with Symmetry-Breaking-H, it expanded 7.7% and 4.59%, respectively. It is rather difficult to explain why it performs better with Symmetry-Breaking-H instead of Symmetry-Breaking-A and unfortunately we cannot give a strong explanation like we can with the SUPER8 anomaly. The best guess we can come up with is that this may be due to the artificial distances of the CIRC instance. But regardless, since Symmetry-Breaking-H with the rotational symmetry breaking works better, we use this when applying CIDA* to larger CIRC instances.

One last, minor test we had performed was to see if there was any difference between choosing home or away first when picking the second team of a pairing. All tests indicate there is no significant difference between the two, thus we arbitrarily choose home teams first.

8.2 CIDA* and the Final Results

This section looks at the results of comparing CIDA* with the best past approach, Branch-and-Price with Column Generation [25]. Following this, we will present the best overall results of CIDA*, which include the first time solving of real-distance instances NL10, GALAXY10, SUPER10, and the solving of CIRC10.

8.2.1 Comparison with Branch-And-Price

The first set of comparisons deals with comparing CIDA* against Irnich's Branch-and-Price algorithm. Their approach was the first to solve NL8, and has had the best performance prior to our work. The branch-and-price algorithm was run on a 2.66GHz processor with 4GB of ram. To allow for similar testing, we ran CIDA* on one processor running at 2.4GHz with 3GB of ram.

Table 8.3 shows the results of this comparison. As shown, CIDA* was able to significantly reduce the time needed to solve all problem instances shown. Additionally, it is able to solve CIRC8 to optimality, while the branch-and-price algorithm was unable to, only finding a lower bound instead.

8.2.2 Overall Results

The final two experiments here are to look at the best results achieved with CIDA* on the TTP. The first is with single and four processors on the 4-, 6-, and 8-team

Table 8.3: Comparison of CIDA* with Irnich's Branch-and-Price algorithm. We note that CIRC8 was not solved to optimality by the Branch-and-Price approach, only a lower bound was found. All times are in seconds.

Instance	Branch-and-Price	CIDA*
NL6	509	0.8
NL8	43 321	195.0
CIRC6	10 789	0.07
CIRC8	300 087	22.25

Table 8.4: Timings, in seconds, to solve instances to optimality with a single processor and with four processors in a shared-memory configuration.

Instance	Solution	Time (P=1)	Time (P=4)
CIRC4	20	0.0	-
CIRC6	64	0.07	-
CIRC8	132	22.25	5
NL4	8276	0.0	-
NL6	23916	0.8	-
NL8	39721	195.0	42
SUPER4	63405	0.0	-
SUPER6	130365	0.49	-
SUPER8	182409	687.24	140
GALAXY4	416	0.0	-
GALAXY6	1365	0.64	-
GALAXY8	2373	428.62	92

instances. Table 8.4 shows these results, with all times in wall time in seconds. Only instances of 8 teams were run solved on four processors since the smaller team sets can be solved quickly. The real-distance problems of NL, SUPER, and GALAXY use the Symmetry-Breaking-A with maximal distance ordering while CIRC uses Symmetry-Breaking-H and rotational symmetry breaking. CIDA* was able to solve all 4-team instances in a time shorter than the smallest resolution of the timer, hence the 0.0 second listings for the 4-team instances.

The next, and final, results are with the larger 10-team instances. These experiments were the result of running CIDA* on 120 CPUs with a distributed memory approach. These used the same settings as before with one exception: with CIRC10, due to the rotational symmetry breaking greatly decreasing the number of subtrees, s^d was set to $\frac{n}{2} + 1$ to ensure that there are enough subtrees for the larger number

Table 8.5: Wall time in seconds for solving instances with 120 CPUs in parallel.

Instance	Solution	Time
CIRC10	242	95 840
NL10	59436	242 440
SUPER10	316329	7 740
GALAXY10	4535	769 331

of processors. These results are seen in Table 8.5, with all 10-team instances having been proven to optimality. This is the first approach that is able to solve CIRC10 and all real-distance instances.

An anomaly seen during the running of these final experiments on the distributed computing setup was with the load balancing. For all 10-team instances, even though most processors would finish at the same time, there would occasionally be a few processors which took longer to finish. This then caused some idleness, most noticeably in the second-to-last iteration. When looking at the point of time between the first processor finishing and the last processor finishing in the second-to-last iteration, there was a difference of 2.7% for CIRC10, 7.2% for NL10, 6.1% for SUPER10, and 6.2% for GALAXY10.

It is difficult to know how much of this anomaly was caused by our subtree forest idea and that by external factors such as some CPUs not running at their full clock speed or possible delays in communication from the network. This is more important since CIRC, NL, and GALAXY were all run on different groupings of computers due to the narrow window of time to run these experiments. When SUPER10, the only problem set which we could run on different groupings since its short running time, was run on a different group than the timing reported in Table 8.5, it had a running time of 7 311 and a difference between the first and last processor finishing of 4.4%. We had also noticed that during the course of these final experiments, some computers would crash and bring down the experiments, with 2 or 3 having crashed during the GALAXY10 experiment alone.

In a sense, this could then be seen as pushing the limits of our subtree idea for very large computing, and to work with even more processors would require taking the subtree forest idea to a further level, something which is discussed further in the conclusions.

8.3 Summary of Results

As shown, the various ideas introduced for heuristic search have had a great impact. When focusing on the TTP, CIDA* has produced many strong results. It works well by itself on CIRC instances due to the artificial distances, and when combined with FD and EP, it is able to find solutions quickly for real-world distance instances.

From a parallelism perspective, it has shown that it can take advantage of the available processing power fairly well. With the shared-memory approach, it exhibited speedup anomalies, finding solutions in less CPU time than when running sequentially. With the distributed-memory approach on the 120-CPU network, the subtrees were pushed to their limits but still allowed CIDA* to find optimal solutions to 10-team instances. We believe future work can make them more adaptable and flexible than what we have shown here.

When comparing to past approaches, CIDA* is able to find known optimal solutions in a fraction of the time needed. This can be attributed to a combination of factors: CIDA* and the subtree skipping, FD and EP improving performance on real-world distances, symmetry breaking, disjoint pattern databases, and pattern matching. CIDA* is also the first to find optimal solutions to some of the 10-team instances by taking advantage of the parallel processing capabilities of the subtrees.

Chapter 9

Conclusions

Little Strokes, Fell great Oaks.

-Benjamin Franklin

Poor Richard's Almanack

And now it is time to draw this story to a close. As we have shown, we have created two new approaches to the TTP, one based on ACO and the other based on IDA*. They have both shown good results, with the latter exhibiting results which show that it is state-of-the-art.

9.1 Summary

This work has made contributions to the fields of sports scheduling, metaheuristics, and heuristic search. With the first, we have shown the first AI-centric approach for finding optimal solutions, with CIDA* finding results that far exceed other approaches. It can find known optimal solutions in a fraction of the time needed by past approaches, and it has been able to find new solutions, being the first approach to solve any 10 team set that does not consist of constant distances.

The second contribution is from the metaheuristic perspective. The ACO approach contributes a new way of fusing ideas of ACO, which is good at optimization, and constraint processing, which has shown to handle the TTP constraints effectively.

It also presents the first use of pattern matching for constraint propagation, which has had a profound effect on reducing the constraint conflicts with the TTP.

The third contribution is from the heuristic search perspective. CIDA* builds on the IDA* algorithm and uses old ideas in new ways. This is also the first approach which does not need to search the whole search tree up to the limiting threshold during an iteration, an important aspect of heuristic search. Another part of this contribution is two new ideas which are more problem-specific. These ideas, FD and EP, help to reduce the node expansion for IDA*. The former, FD, does this by reducing the number of iterations IDA* needs for combinatorial optimization problems which require too many iterations, and the latter, EP, does this by both improving the performance of FD and by quickly finding the optimal solution in the final iteration of IDA*.

9.2 Future Work

There are multiple paths of future work that stem out of this research. From the ACO side of the work, one of the things that needs to be looked at is the heuristic values. We had looked at a simple, fast heuristic estimate, which was shown to work poorly for this application. While CIDA* was able to use the more accurate ILB, this heuristic estimate is slow to calculate as the problem size grows since one is essentially solving a TSP instance, which may make it unusable for the larger instances. Thus one area of research is to look for a heuristic estimate that is more accurate than what we used for ACO, but faster to calculate than what we used for CIDA*.

Another area for ACO is improving the balance between exploitation and exploration of the solution space. Even though ACS's rule for choosing values allowed the approach to better exploit the past solutions, leading to the improved results, further work could help to better the balance between exploration and exploitation. We do believe the current form is too focused on exploitation, which is why the performance tends to tail off a bit on the larger instances which have the much larger solution space. Possible approaches to this would be looking at further optimization of the parameters which impact the pheromone or a possible gradual shifting during the running of the algorithm from using ACS's rule to AS's rule for choosing values instead of relying on only ACS's rule.

In the area of CIDA*, one aspect to look at is with the subtree forest and making it more dynamic. As it is right now, the subtrees are static and never change during the running of the algorithm. An alternative to this is for the subtrees to be dynamic. For example, the algorithm could further split subtrees which have large trees expanding from them during the search while at the same time merging any set of subtrees which share the same $s^d - 1$ parents nodes and whom all have small trees expanding from them. There are two possible benefits from this. The first is that it could reduce the possibility of idleness when running on large-scale distributed memory approaches. The second is that it could improve the gains from subtree skipping, since some of the larger subtrees which are split may have some of their descendants skipped for the next iteration.

Another area to look at is possible hybridizations of IDA* with techniques from the operations research field. With many other problems, most notably the TSP, operations research has been in the lead of finding optimal solutions. The fact that IDA* could handily beat operations research in this aspect for the TTP is a bit surprising. But to be able to solve larger instances, such as 12-team instances, we believe that combining the strengths of the two fields is needed. Whether this requires bringing in ideas from operations research into artificial intelligence or bringing ideas from artificial intelligence into operations research, or even a new approach which draws from both areas, remains to be seen. But in the end, bringing these two fields closer together can lead to new approaches that can solve optimization problems which cannot be solved to optimality with current techniques, such as large instances of the TTP.

Bibliography

[1] A.-E. Al-Ayyoub. Performance evaluation of parallel iterative deepening A* on clusters of workstations. *Performance Evaluation*, 60(1-4):223 – 236, 2005.

[2] A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. *J. of Scheduling*, 9(2):177–193, 2006.

[3] A. Bar-Noy and D. Moody. A tiling approach for fast implementation of the traveling tournament problem. In *PATAT 2006*, pages 351–358, 2006.

[4] J. C. Beck. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research*, 29:49–77, 2007.

[5] T. Benoist, F. Laburthe, and B. Rottembourg. Lagrange relaxation and constraint programming collaborative schemes for travelling tournament problems. In *CP-AI-OR 2001, Wye College, UK*, pages 15–26, 2001.

[6] P.-C. Chen, G. Kendall, and G. V. Berghe. An ant based hyper-heuristic for the travelling tournament problem. In *IEEE Symposium on Computational Intelligence in Scheduling. SCIS '07.*, pages 19–26, 2007.

[7] K. K. H. Cheung. Solving mirrored traveling tournament problem benchmark instances with eight teams. *Discrete Optimization*, 5(1):138–143, 2008.

[8] K. K. H. Cheung. A benders approach for computing lower bounds for the mirrored traveling tournament problem. *Discrete Optimization*, 6(2):189–196, 2009.

[9] D. J. Cook, L. O. Hall, and W. Thomas. Parallel search using transformation-ordering iterative-deepening-A*. *International Journal of Intelligent Systems*, 8:855–873, 1993.

[10] H. Crauwels and D. Van Oudheusden. Ant colony optimization and local improvement. In *Workshop of Real-Life Applications of Metaheuristics, Antwerp*, 2003.

[11] R. Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, San Francisco, California, 2003.

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

[13] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1:53–66, April 1997.

[14] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[15] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, Cambridge, Massachusetts, 2004.

[16] K. Easton, G. L. Nemhauser, and M. A. Trick. The traveling tournament problem: Description and benchmarks. In T. Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 580–584. Springer, 2001.

[17] K. Easton, G. L. Nemhauser, and M. A. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In E. K. Burke and P. D. Causmaecker, editors, *PATAT*, volume 2740 of *Lecture Notes in Computer Science*, pages 100–109. Springer, 2002.

[18] A. Felner and A. Adler. Solving the 24 puzzle with instance dependent pattern databases. In *SARA*, volume 3607 of *Lecture Notes in Artificial Intelligence*, pages 248–260. Springer-Verlag, 2005.

[19] N. Fujiwara, S. Imahori, T. Matsui, and R. Miyashiro. Constructive algorithms for the constant distance traveling tournament problem. In E. K. Burke and H. Rudová, editors, *PATAT*, volume 3867 of *Lecture Notes in Computer Science*, pages 135–146. Springer, 2006.

[20] F. Glover and F. Laguna. *Tabu Search.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[21] Z. Hafidi, E.-G. Talbi, and G. Goncalves. Load balancing and parallel tree search: The MPIDA* algorithm. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *ParCo'95*, pages 93–100. Elsevier Science, 1995.

[22] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[23] P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". *ACM SIGART Bulletin*, (37):28–29, 1972.

[24] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[25] S. Irnich. A new branch-and-price algorithm for the traveling tournament problem. *European Journal of Operational Research*, 204(2):218 – 228, 2010.

[26] S. Irnich and U. Schrempp. A new branch-and-price algorithm for the traveling tournament problem, 2008. Presented at Column Generation 2008, Aussois, France, June 17-20, 2008. Available from: http://www.gerad.ca/colloques/ColumnGeneration2008/slides/SIrnich.pdf [Accessed 07 March, 2009].

[27] G. Kendall, S. Knust, C. C. Ribeiro, and S. Urrutia. Scheduling in sports: An annotated bibliography. *Computers & Operations Research*, 37(1):1 – 19, 2010.

[28] G. Kendall, W. Miserez, and G. Vanden Berghe. A constructive heuristic for the travelling tournament problem. In *PATAT 2006*, pages 443–447, 2006.

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

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

[31] D. G. Kyle. *Sport and Spectacle in the Ancient World*. Blackwell Publishing, Malden, Massachusetts, 2007.

[32] J. H. Lee, Y. H. Lee, and Y. H. Lee. Mathematical modeling and tabu search heuristic for the traveling tournament problem. In M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, editors, *ICCSA (3)*, volume 3982 of *Lecture Notes in Computer Science*, pages 875–884. Springer, 2006.

[33] A. Lim, B. Rodrigues, and X. Zhang. A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operational Research*, 174(3):1459–1478, 2006.

[34] B. Meyer and A. T. Ernst. Integrating aco and constraint propagation. In M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Stützle, editors, *ANTS Workshop*, volume 3172 of *Lecture Notes in Computer Science*, pages 166–177. Springer, 2004.

[35] C. Powley, C. Ferguson, and R. E. Korf. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, 60(2):199–242, 1993.

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

[37] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[38] V. N. Rao, V. Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-A*. In *AAAI*, pages 178–182, 1987.

[39] R. V. Rasmussen and M. A. Trick. Round robin scheduling - a survey. *European Journal of Operational Research*, 188(3):617 – 636, 2008.

[40] A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(7):701–710, Jul 1994.

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

[42] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. De Sarkar. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50(2):207–221, 1991.

[43] C. Solnon. Ants can solve constraint satisfaction problems. *Evolutionary Computation, IEEE Transactions on*, 6(4):347–357, Aug 2002.

[44] T. Stützle and H. H. Hoos. $\mathcal{MAX}-\mathcal{MIN}$ ant system. *Journal of Future Generation Computer Systems*, 16(8):889–914, 2000.

[45] E.-G. Talbi. *Parallel Combinatorial Optimization*. John Wiley & Sons, Hoboken, NJ, 2006.

[46] S. Urrutia, C. Ribeiro, and R. Melo. A new lower bound to the traveling tournament problem. In *IEEE Symposium on Computational Intelligence in Scheduling*, pages 15–18, 2007.

[47] S. Urrutia and C. C. Ribeiro. Maximizing breaks and bounding solutions to the mirrored traveling tournament problem. *Discrete Applied Mathematics*, 154(13):1932 – 1938, 2006.

[48] D. C. Uthus, P. J. Riddle, and H. W. Guesgen. Ant colony optimization and the single round robin maximum value problem. In M. Dorigo, M. Birattari, C. Blum, M. Clerc, T. Stützle, and A. F. T. Winfield, editors, *ANTS Conference*, volume 5217 of *Lecture Notes in Computer Science*, pages 243–250. Springer, 2008.

[49] P. Van Hentenryck and Y. Vergados. Population-based simulated annealing for traveling tournaments. In *AAAI*, pages 267–271. AAAI Press, 2007.

[50] N. Vempaty, V. Kumar, and R. Korf. Depth-first vs best-first search. In *Proceedings of AAAI-91*, pages 434–440, 1991.