

Honours Year Project Report

**Constraint Programming  
for the  
Travelling Tournament Problem**

**By  
Gan Tiaw Leong**

Department of Computer Science  
School of Computing  
National University of Singapore

2002/2003

Honours Year Project Report

**Constraint Programming  
for the  
Travelling Tournament Problem**

**By  
Gan Tiaw Leong**

Department of Computer Science  
School of Computing  
National University of Singapore  
2002/2003

Project No: H41070  
Advisor: Dr Martin Henz

Deliverables:  
Report: 1 Volume  
Program: 1 Diskette

# Abstract

The objective of this project is to find the optimal solution to the TTP benchmark instances more efficiently so that we can tackle previously intractable instances. In order to achieve this goal, we made use of a constraint model to generate feasible timetables, branch and bound search to prune the search tree and a propagator produced a lower bound for team travelling distance by reducing the finite domain representation into a graph representation. Our approach is focused on producing tighter lower bound than the simple minimum distance travelled independent of team constraints suggested by the team who first proposed the TTP. While we managed to achieve a reduction in search space size with the new lower bound, the time delay resulting from the bound propagator causes the execution to take longer. It appears that the current lower bound has to be made even tighter before large instances can be handled.

## **Subject Descriptors:**

G.2.2 Graph Theory

G.1.6 Optimization

D.1.3 Concurrent Programming

## **Keywords:**

Constraint Programming, Graph modelling, Optimisation

## **Implementation Software and Hardware:**

Linux Server, LEDA, MOZART-OZ and C++

## Acknowledgement

I would like to thank Dr Martin Henz for all his help and patience during the project.

## Table of Contents

|  |     |
|--|-----|
| Title  | I   |
| ABSTRACT   | II  |
| ACKNOWLEDGEMENT  | III |
| CHAPTER 1  | 1   |
| INTRODUCTION   | 1   |
| 1.1 <i>Aim of the project</i>                          | 1   |
| 1.2 <i>Definition of Travelling Tournament Problem</i> | 1   |
| 1.3 <i>Previous Approaches and Experiment Results</i>  | 2   |
| CHAPTER 2  | 6   |
| SOLUTION METHODS, TECHNIQUES AND TOOLS                 | 6   |
| 2.1 <i>Our general approach</i>                        | 6   |
| 2.2 <i>Constraint Programming Model</i>                | 7   |
| 2.3 <i>The Mozart – Oz Platform</i>                    | 14  |
| 2.4 <i>Branch and Bound</i>                            | 15  |
| 2.5 <i>Lower Bounds for the travelling distances</i>   | 18  |
| 2.6 <i>The LEDA programming library</i>                | 23  |
| CHAPTER 3  | 25  |
| COMPUTATIONAL RESULTS                                  | 25  |
| 3.1 <i>Experimental results with timing</i>            | 25  |
| CHAPTER 4  | 28  |
| DISCUSSION   | 28  |
| 4.1 <i>Graph Model 1</i>                               | 28  |
| 4.2 <i>Graph Model 2</i>                               | 29  |
| 4.3 <i>Artificial bounds</i>                           | 31  |
| 4.4 <i>Discussion of results</i>                       | 32  |
| CHAPTER 5  | 34  |
| CONCLUSIONS  | 34  |
| 5.1 <i>Conclusion on the lower bounds</i>              | 34  |
| CHAPTER 6  | 36  |
| FUTURE WORK  | 36  |
| 6.1 <i>Possible approach 1</i>                         | 36  |
| 6.2 <i>Possible approach 2</i>                         | 36  |
| 6.3 <i>Possible approach 3</i>                         | 37  |
| References   | 38  |

# Chapter 1

## Introduction

### 1.1 Aim of the project

The aim of this project is to study the relatively new benchmark problem, the travelling tournament problem. The two main subcomponents of this problem are the constraint satisfaction problem and the optimisation of travelling distance problem. The problem of generating feasible timetables that satisfy all constraints had been studied in an earlier project with good results (Henz, Muller, Thiel, 2002). Our focus is on the optimisation aspect of the problem. We will develop graph algorithms for the efficient search for the optimal solution.

### 1.2 Definition of Travelling Tournament Problem

The Travelling Tournament Problem (TTP) is a tournament scheduling problem with a strong optimisation component (Trick, 2003). The problem was first proposed by Easton, Nemhauser and Trick.

“Given  $n$  teams with  $n$  even, a double round robin tournament is a set of games in which every team plays every other team exactly once at home and once away. A game is specified by an unordered pair of opponents. Exactly  $2(n - 1)$  slots or time periods are required to play a double round robin tournament. Distances between team sites are given by an  $n$  by  $n$  distance matrix  $D$ . Each team begins at its home site and travels to play its games at the chosen venues. Each team then returns (if necessary) to its home base at the end of the schedule.” (Easton, Nemhauser, Trick, 2001)

The formal definition of the TTP is as follows:

**Input:**  $n$ , the number of teams;  $D$ , an  $n$  by  $n$  integer distance matrix;

**Output:** A double round robin tournament on the  $n$  teams such that

- All constraints on the tournament are satisfied, and
- The total distance travelled by the teams is minimized.

Our research is on computing the optimal total travel distance in the intermural even double round robin travelling tournament problem. The benchmark instances (Trick, 2003) were used as our problem instances.

The constraints for these benchmark instances are as follows:

1. Double round robin (A at B and B at A):  $n$  teams need  $2(n-1)$  slots.
2. No more than three consecutive home or three consecutive away games for any team
3. No repeaters (A at B, followed immediately by B at A)
4. Objective is to minimize distance travelled (assume teams begin in their home city and must return there after the tournament)
5. NL $x$  takes the first  $x$  cities in the distance matrix.

In addition to the constraints above, there are some basic assumptions about the tournament. Firstly, the distance from locations A to B is equals to the distance from locations B to A. The distances between team sites are given by a symmetrical  $n$  by  $n$  distance matrix  $D$ . We also assume that all teams must be at home at the start and end of the tournament. Finally, a team's travel distance is the summation of all direct distances between venues in consecutive rounds of play.

These benchmark instances were selected for good reasons. Since we are working on an optimisation problem with emphasis on time and space efficiency, we would like to have existing benchmarks to compare our results against. Much of the research on the TTP has been centred on these instances making them a logical starting point for us. The instances were formulated with data and constraints from a real tournament, thus it gives us an indication of the kind of constraints that we may encounter for real-world instances of the TTP. It is important to take note that the complexity of the TTP increase so rapidly in relation to the number of teams in the tournament that all instances greater than NL4 cannot be solved by a straightforward search algorithm due to space and time constraints. In other words, all instances with the exception of NL4 are intractable for current optimal solution approaches.

### **1.3 Previous Approaches and Experiment Results**

In previous research, Dr Henz and his team had shown that graph algorithms can significantly reduce the size of search trees for the constraint satisfaction portion of the

problem (Henz, Muller, Thiel, 2002). They were able to use graph algorithms to implement arc-consistent constraint propagators for the global constraints *all-different* and *one-factor*. Other attempts at the full problem of finding a feasible timetable with optimal travel distance employed non-exhaustive approximation methods that were able to reach optimality for small instances and near optimality for large instances. Note that all results presented below are from research based on approximation approaches. The results are compiled from Michael Trick's website (Trick, 2003) on the TTP.

Using a combination of constraint programming and Lagrange relaxation, Rottembourg's team (Benoist, Laburthe and Rottembourg, 2001) was able to obtain optimal solutions for the NL4 and NL6 instances and fairly good solutions for the larger instances. However, the percentage of deviation between the known lower bound and their solution appears to increase dramatically as the instances get larger. Table 1 presents the best results achieved for the benchmark instances (Trick, 2003).

Another group of researchers (Easton et al, 2001. Trick, 2002) used Linear Programming (LP) to solve the same instances. The best results achieved for the benchmark instances (Trick, 2003) are presented in Table 2. Their results are much better than Rottembourg's especially for NL16, however they seemed to have no solution for NL10 – NL14.

Cardemil (Trick, 2003) also achieved good results for the data set, i.e. NL4-16. The best results achieved for the benchmark instances (Trick, 2003) are presented in Table 3.

Another HYP student Zhang Xing Wen's approach utilizes three different models working together. A constraint programming model generates a feasible timetable, a simulated annealing model mutates the initial solution and a hill-climbing model evaluates the costs of the mutated timetables. The best results from his approach are presented in Table 4.

The best results from all previous approaches are shown in Table 5. The lower bounds for Instance NL4-16 are from Easton and Waalewijn (Trick, 2003).



| Instance | Lower Bound | Best Results | Deviation |
|----------|-------------|--------------|-----------|
| NL4      | 8276        | 8276         | 0%        |
| NL6      | 23916       | 23916        | 0%        |
| NL8      | 39479       | 42517        | 7.70%     |
| NL10     | 57500       | 68691        | 19.5%     |
| NL12     | 107483      | 143655       | 33.7%     |
| NL14     | 182797      | 301113       | 64.7%     |
| NL16     | 248852      | 437273       | 75.7%     |

Table 1 Rottembourg's Best Results

| Instance | Lower Bound | Best Results | Deviation |
|----------|-------------|--------------|-----------|
| NL4      | 8276        | 8276         | 0%        |
| NL6      | 23916       | 23916        | 0%        |
| NL8      | 39479       | 41113        | 4.14%     |
| NL10     | 57500       | -            | -         |
| NL12     | 107483      | -            | -         |
| NL14     | 182797      | -            | -         |
| NL16     | 248852      | 312623       | 25.6%     |

Table 2 Easton's Best Results

| Instance | Lower Bound | Best Results | Deviation |
|----------|-------------|--------------|-----------|
| NL4      | 8276        | -            | -         |
| NL6      | 23916       | -            | -         |
| NL8      | 39479       | 40416        | 2.37%     |
| NL10     | 57500       | 66037        | 14.8%     |
| NL12     | 107483      | 124803       | 16.1%     |
| NL14     | 182797      | 216108       | 18.2%     |
| NL16     | 248852      | 308413       | 23.9%     |

Table 3 Cardemil's Best Results

| Instance | Lower Bound | Best Results | Deviation |
|----------|-------------|--------------|-----------|
| NL4      | 8276        | 8276         | 0.00%     |
| NL6      | 23916       | 24073        | 0.66%     |
| NL8      | 39479       | 39947        | 1.19%     |
| NL10     | 57500       | 61608        | 7.14%     |
| NL12     | 107483      | 119012       | 10.73%    |
| NL14     | 182797      | 207075       | 13.28%    |
| NL16     | 248852      | 293175       | 17.81%    |

Table 4 Zhang's Best Results

| Instance | Lower Bound | Best Results | Deviation |
|----------|-------------|--------------|-----------|
| NL4      | 8276        | 8276         | 0%        |
| NL6      | 23916       | 23916        | 0%        |
| NL8      | 39479       | 39947        | 1.19%     |
| NL10     | 57500       | 61608        | 7.14%     |
| NL12     | 107483      | 119012       | 10.73%    |
| NL14     | 182797      | 207075       | 13.28%    |
| NL16     | 248852      | 293175       | 17.81%    |

Table 5 Previous Best Results

## Chapter 2

### Solution Methods, Techniques and Tools

#### 2.1 Our general approach

Our task can be split into two distinct components; the discrete search for feasible timetables that satisfy all constraints defined and the combinatorial search for the optimal solution. The first component can be solved by encoding the intermural double round robin tournament scheduling problem as a constraint satisfaction problem (CSP). The problem is modelled as a set of decision variables representing integers with constraints limiting their range of values. Finite domain constraint programming is used to generate feasible timetables through repeated steps of propagation and distribution. Once we get an initial solution to the CSP, we can conduct branch and bound search for the optimal solution. In order to improve the performance of the branch and bound algorithm, we need to generate tighter bounds. Since the upper bound of any new travelling distance is constrained to be lower than the best solution so far, we turn our attention to improve the lower bound instead.

For our problem solver, we adopted a three phase approach:

##### *Phase 1 Generating Team Home/Away Play Patterns*

In the first phase, we create a set of team patterns for the teams in the tournament. By pre-determining a set of home and away patterns which will produce feasible timetables, we limit the search space for the second phase. A set of boolean (0/1) finite domain variables is used to represent the pattern of play for a team in the tournament. The constraints on the length of home stands and road trips are imposed on the patterns created. Other simple constraints like the actual number of home and away games a pattern must contain to be feasible are also enforced here. This step is very important since the next phase will perform distribution on the pattern set. The size of the pattern set generated in this phase has a direct effect on the complexity of the search for feasible timetables.

##### *Phase 2 Generating Feasible Timetables*

In this phase, we take the pattern set from phase 1 as input. Our instances do not have special constraints that limit assignment of teams to specific patterns or opponents. Hence

the constraint store will reach an initial stable state without much trimming of the finite domain variables. We want to ensure that our possible search space is kept minimal; therefore we carry out distribution on the choice of team patterns for our teams. By determining the pattern of play for teams as early as possible, we can reduce the number of variations in team assignments dramatically. For example, team 1 is assigned the pattern {H, H, A, H, A, A} and team 2 is assigned the pattern {H, H, H, A, A, A}, we immediately know that these two teams can only play against each other in the 3<sup>rd</sup> and 4<sup>th</sup> rounds. However, such an arrangement would violate the *no repeater constraint* and immediately lead to failure of the current branch.

### *Phase 3 Branch and Bound Search for the optimal travelling distance*

The optimisation of the total travelling distance of a tournament is clearly a NP-Hard search problem. We adopt the branch and bound search algorithm to try and remove as much of the search space as possible; hopefully to a point where previously intractable problems become solvable. Branch and Bound search ignores a particular branch of the search tree as soon as it becomes impossible for the branch to get a better solution. However, just finding the optimal solution isn't good enough, we need to be able to terminate our search once we get the optimal solution. The decision problem of determining whether a given solution is indeed optimal is a problem in the NP-Complete class. A simple solution is to exhaust all remaining alternative solutions and compare the solutions. A better way is to try and develop a tight lower bound of the optimal solution. This lower bound will then act as a stop sign for the search, terminating it once the upper bound of the partial solution being investigated overlaps the lower bound of the optimal solution. This significantly reduces the number of timetables to be explored.

We recognised the similarity between computing the minimum travel distance for a partially constrained timetable to graph problems such the travelling salesman problem. Hence, we implemented a graph-based propagator to compute the lower bound for distance travelled by a team given its binding to a home-away pattern and finite domains of possible opponents at each round.

## **2.2 Constraint Programming Model**

### **2.2.1 Constraint Programming**

Constraint programming is an emergent software technology for declarative description and effective solving of large, particularly combinatorial, problems especially in areas of planning and scheduling. Much of the following introduction to constraint programming is sourced from Dr Henz's overview of constraint programming (Henz, 2000).

Constraint programming is the study of computational systems based on constraints. The main idea behind constraint programming is the solving of a problem by the stating of constraints (conditions, properties) which must be satisfied by the solution. A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can hold; it represents some partial information about the variables of interest. For instance, "the circle is inside the square" relates two objects without precisely specifying their positions, i.e., their coordinates. Now, one may move the square or the circle and he or she is still able to maintain the relation between these two objects. Also, one may want to add other object, say triangle, and introduce another constraint, say "square is to the left of the triangle". From the user (human) point of view, everything remains absolutely transparent.

One important type of constraint programming is known as *Finite Domain Constraint Programming*. A finite domain as suggested by its name is a variable that holds a finite number of possible non-negative integer values that represent its current domain or set of possible values. These finite domain variables are stored in the constraint store. A *constraint* is a formula of predicate logic. It restricts the values a finite domain variable in the constraint store can take.

- A *domain constraint* takes the form  $x \in D$ , where  $D$  is a finite domain. Domain constraints can express constraints of the form  $x = n$  since  $x = n$  is equivalent to  $x \in \{n\}$ .
- A *basic constraint* takes one of the following forms:  $x = n$ ,  $x = y$  or  $x \in D$ , where  $D$  is a finite domain.

More formally, the constraint store is a conjunction of constraints of the form  $x \in S$ , where  $S$  is a set of integers. These constraints are called the basic constraints. Computation starts with an initial domain for each variable as given in the model.

Some constraints can be directly entered in the constraint store by strengthening the basic constraint on a variable. For example, the constraint  $x \neq 5$  can be expressed in the constraint store by removing 5 from the domain of  $x$ .

Other more complex constraints are represented by computational agents called propagators. Each propagator observes the variables given by the corresponding constraint in the problem. Whenever possible, it strengthens the constraint store with respect to these variables by excluding values from their domain according to the corresponding constraint. For example, a propagator for the constraint  $x \leq y$  observes the upper and lower bounds of the domains of  $x$  and  $y$ . A possible strengthening consists of removing all values from the domain of  $x$  that are greater than the upper bound of the domain of  $y$ . A finite domain variable becomes a singleton when the domain is reduced to a single value. When all variables become singletons without violating any constraints, a solution is found. Notice that a finite domain problem has at most finitely many solutions, provided we consider only variables that occur in the problem.

The process of propagation continues until no propagator can further strengthen the constraint store. The constraint store is said to be stable. At this point, many problem variables typically still have non-singleton domains. Thus the constraint store does not represent a solution yet, and a search phase becomes necessary.

Since the constraint store is now stable and no propagation is possible, we need to discretely determine the value of some of the non-singleton domains. This is known as the distribution phase. A new branching constraint  $c$  is generated to alter the domain of a chosen variable. This branching constraint produces two new generally non-stable stores. In other words,  $c$  and  $\sim c$  modifies the constraint store to trigger a new wave of propagation within the store. After stability is reached again, this branching process is continued recursively on both sides until the resulting store is either inconsistent or represents a solution to the problem.

Finite domain constraint programming is best seen as a software framework for combining software components to achieve problem-specific tree search solvers. These software components can be organized into three families.

1. **Propagation algorithms** implement individual constraints by describing how the constraints can be employed to strengthen the constraint store.
2. **Branching algorithms** select branching constraints at each node of the search tree after all propagation has been done. Branching algorithms define the size and shape of the search tree.
3. **Exploration algorithms** describe which part of a given search tree is explored and in which order. In this project, exploration algorithms are mostly ignored.

Constraint propagation is an efficient inference mechanism obtained with concurrent propagators accumulating information in a constraint store. Constraint distribution splits a problem into complementary cases once constraint propagation cannot advance further. By iterating propagation and distribution, propagation will eventually determine the solutions of a problem. Constraint distribution can easily lead to an exponential growth of the number of sub-problems to be considered. Fortunately, this potential combinatorial explosion can often be contained by combining strong propagation mechanisms with problem specific heuristics for selecting distribution steps.

One class of problems in which finite domain programming excels, is the Constraint Satisfaction Problem (CSP). The Constraint Satisfaction Problem (CSP) is a problem where one is given:

- a finite set of variables,
- a function which maps every variable to a finite domain,
- a finite set of constraints.

Each constraint restricts the combination of values that a set of variables may take simultaneously. A solution of a CSP is an assignment to each variable a value from its domain satisfying all the constraints. The task is to find one solution or all solutions.

The CSP is a combinatorial problem which can be solved by search. There exists a trivial algorithm that solves such problems or finds that there is no solution. This algorithm generates all possible combinations of values and, then, it tests whether the given combination of values satisfies all constraints or not (consequently, this algorithm is called generate and test). Clearly, this algorithm takes a long time to run

so the research in the area of constraint satisfaction concentrate on finding algorithms which solve the problem more efficiently, at least for a given subset of problems.

### 2.2.2 Constraints of the TTP

In this section, we examine the constraints of the travelling tournament problem. The first component of the problem is the task of scheduling a double round robin tournament. The most important constraints for round robin tournaments are

- The *all-different constraint*, which expresses that the rows in a tournament contain every team only once, and
- The *one-factor constraint*, which expresses that every column groups the teams into matches.

In the case of double round robin tournaments, the first constraint needs to be replaced by

- The *double round-robin constraint* states that each team must appear in every row exactly twice. Once at A's home site and once at B's home site. Thus there are total  $2(n-1)$  rounds, and during each round  $n/2$  games are played.

| NL4   |   | Rounds |   |   |   |   |   |
|-------|---|--------|---|---|---|---|---|
|       |   | 1      | 2 | 3 | 4 | 5 | 6 |
| Teams | 1 | 2      | 4 | 3 | 4 | 2 | 3 |
|       | 2 | 1      | 3 | 4 | 3 | 1 | 4 |
|       | 3 | 4      | 2 | 1 | 2 | 4 | 1 |
|       | 4 | 3      | 1 | 2 | 1 | 3 | 2 |

Table 6 A double round robin tournament timetable

In our implementation, the *double round-robin constraint* is enforced by constraining the sums of a particular team playing away and at home in a particular round to be equals to one. Then we further constrain the sum of the home and away sums to be exactly two. The one-factor constraint is implemented with the built-in OZ pairwise distinct propagator. The propagator ensures that all opponent assignments in a given round are pairwise distinct.



The benchmark instances that we are working on include 2 additional constraints. (Easton, Nemhauser and Trick, 2001. Trick, 2002):

1. *Consecutive constraint*: No more than three consecutive home or three consecutive away games for any team.
2. *No repeater constraint*: For any pair of teams, namely A and B, match A at B must not immediately follow match B at A.

The *consecutive constraint* is enforced during team pattern generation. We generate home - away patterns that fulfil this constraint with the “NoMoreThan” procedure. The *no repeater constraint* is implemented during team assignment by adding a “not equals” constraint between the “against” variables of the current and immediately preceding rounds. There are also many redundant constraints in the implementation to increase the rate of propagation.

The second component of the problem is the process of finding the timetable with the minimal distance. Before we can proceed to this step, we need a special constraint to dynamically link the travel distances stored in our distance matrix to the timetable being generated.

### **2.2.3 The special travelling distance constraint**

The *travelling distance constraint* links the travel distances from the Distance matrix with the timetable. The timetable’s total distance is constrained to the domain of the sum of all team distances using the finite domain sum propagator. For the purpose of computing the travel distances, we have an extra dummy round for the constraint that all teams must go back home at the end of the tournament. The travel distance of any team is similarly constrained with the sum propagator imposed on the travel distance of that team in each round. In determining the correct distance to assign, we always have one of three cases. It is the first round, the last round or a round in between. If we are extracting the distance in the first round, the team is travelling from the start node to either a home node (in which case, distance equals zero) or an away node (distance equals the distance between our team’s home and the opponent’s home). If we are looking at the end of the tournament, we examine the last match. If the last match was a home game, we assign a value of zero for distance. If we had an away game, the distance equals the cost of returning home from the opponent’s location.

For the final case (not first or last round), we need to consider four possible scenarios for our travelling distance. We could be travelling

- from an away game to a home game. Distance from opponent location to team home.
- from a home game to an away game. Distance from team home to opponent location.
- from an away game to another away game. Distance from opponent 1's location to opponent 2's location.
- from a home game to another home game. It is not explicitly handled in the code below since all instances not matching the previous three will have the same effect as distance equal zero.

```
%% Code fragment for determining distance travelled for this
round
%% when round is the first or the last.
%% R-1 away; R home
D1 = {FD.decl}
Index1 = {FD.decl}
S1 = {FD.decl}
Index1 =: InRound.(R-1).playsTeam.T.against + TeamN * T - TeamN
Switch1 = {FD.decl}
{FD.conj InRound.(R-1).playsTeam.T.away
InRound.R.playsTeam.T.home Switch1}
{FD.times D1 Switch1 S1}
{FD.element Index1 FlatMatrix D1}
%% R-1 home; R away
D2 = {FD.decl}
Index2 = {FD.decl}
S2 = {FD.decl}
Index2 =: InRound.R.playsTeam.T.against + TeamN * T - TeamN
Switch2 = {FD.decl}
{FD.conj InRound.(R-1).playsTeam.T.home
InRound.R.playsTeam.T.away Switch2}
{FD.times D2 Switch2 S2}
```

```

{FD.element Index2 FlatMatrix D2}
%% R-1 away; R away
D3 = {FD.decl}
Index3 = {FD.decl}
S3 = {FD.decl}
Index3 =: InRound.(R-1).playsTeam.T.against + TeamN *
InRound.R.playsTeam.T.against - TeamN
Switch3 = {FD.decl}
{FD.conj InRound.(R-1).playsTeam.T.away
InRound.R.playsTeam.T.away Switch3}
{FD.times D3 Switch3 S3}
{FD.element Index3 FlatMatrix D3}
in
    X =: S1 + S2 + S3
End

```

## 2.3 The Mozart – Oz Platform

Our main development tool is the Mozart implementation of the Oz programming language. The following introduction to the Oz programming language is taken from the Mozart online documentation.

“Oz is a multi-paradigm language that is designed for advanced, concurrent, networked, soft real-time, and reactive applications. Oz provides the salient features of object-oriented programming including state, abstract data types, objects, classes, and inheritance. It provides the salient features of functional programming including compositional syntax, first-class procedures/functions, and lexical scoping. It provides the salient features of logic programming and constraint programming including logic variables, constraints, disjunction constructs, and programmable search mechanisms. It allows users to dynamically create any number of sequential threads. The threads are dataflow threads in the sense that a thread executing an operation will suspend until all operands needed have a well-defined value.”

For our project, Oz served as a finite domain constraint programming language with a C++ native interface for implementations of special constraint propagators. All propagation, distribution and search functions are built-in and automatically handled. Our

programming task involved declarations of input data, finite domain variables, constraints on those variables and the ordering function for branch and bound search. Everything else including flow of execution, memory allocation and de-allocation is handled automatically by the system. We developed a constraint programming solver for the double round robin tournament scheduling component using the finite domain constraint functions provided and extended it to incorporate travelling distances. The main bulk of the project lay in the special constraint propagator we implemented using the C++ extensions provided by Oz.

## 2.4 Branch and Bound

### 2.4.1 Branch and Bound Search

Once we extended the constraint model to include travel distance, we could use brute-force search to try and find the optimal solution. The most obvious way is to just generate all feasible timetables, compare their travel distances and return the timetable with the lowest total travelling distance. However, the TTP is an NP-hard optimisation problem and an exhaustively search of all possibilities will have a space requirement that will overwhelm any computer and take a longer time than the age of the universe for any but the most trivial instances. We clearly need a smarter way of searching.

*Branch-and-bound* is an approach developed for solving discrete and combinatorial optimization problems. The discrete optimization problems are problems in which the decision variables assume discrete values from a specified set; when this set is set of integers, we have an integer programming problem. The combinatorial optimization problems, on the other hand, are problems of choosing the best combination out of all possible combinations. Most combinatorial problems can be formulated as integer programs.

The major difficulty with these problems is that we do not have any *optimality conditions* to check if a given (feasible) solution is optimal or not. There are no global optimality conditions in discrete or combinatorial optimization problems. The only way to guarantee a given feasible solution's optimality is "to compare" it with every other feasible solution. To do this *explicitly*, amounts to *total enumeration* of all

possible alternatives which is computationally prohibitive due to the *NP-Completeness* of integer programming problems. Therefore, this comparison must be done *implicitly*, resulting in *partial enumeration* of all possible alternatives. The branch-and-bound approach provides such a means for finding the optimal solution.

The essence of the branch-and-bound approach is the following observation: in the total enumeration tree, at any node, if I can show that the optimal solution cannot occur in any of its descendants, then there is no need for me to consider those descendent nodes. Hence, I can "*prune*" the tree at that node. If I can prune enough branches of the tree in this way, I may be able to reduce it to a computationally manageable size. Note that, I am *not* ignoring those solutions in the *leaves* of the branches that I have pruned, I have left them out of consideration *after* I have made sure that the optimal solution cannot be at any one of these nodes. Thus, the branch-and-bound approach is not a heuristic, or approximating, procedure, but it is an exact, optimizing procedure that finds an optimal solution.

It is always possible to find a feasible solution to a combinatorial or discrete optimization problem. If available, one can use some heuristics to obtain, usually, a "reasonably good" solution. Let us call this solution *the incumbent*. Then at any node of the tree, if we can compute a "bound" on the best possible solution that can expected from any descendent of that node, we can compare the "bound" with the objective value of the incumbent. If what we have on hand, the incumbent, is better than what we can ever expect from any solution resulting from that node, then it is safe to stop branching from that node. In other words, we can discard that part of the tree from further consideration.

With this solution we can safely discard any part of the tree which we know it cannot result in solution better than this. What we need now is a way of computing a "*lower bound*" on the value of the makespan when a partial schedule at an intermediate node is completed.

Two points should be clear: better ("tighter") the lower bound, the more we can prune off the tree, and regardless of the lower bounding scheme used, how much we can prune is highly data dependent. A same branch-and-bound procedure that behaves very well with a specific set of data can do very poorly with a different set of data.

How well a branch-and-bound algorithm solves a specific discrete or a combinatorial optimization problem depends on how branching is going to take place and which bounding scheme is to be used. Starting at the root node, we *partition* the solution space of the problem. In different problems this partitioning or branching can take various forms. It is a major algorithm design issue to decide which one to use.

In order to compute a bound at a particular node, what we essentially do is to solve a *relaxed* problem. We do this by removing a number of *complicating* constraints. In order to be able to prune more of the tree, thus having less alternatives to explicitly evaluate, we need to have *tighter* bounds. To be able to have a tighter bound, one has to solve the problem with minimal relaxation. (Tightest bound is obtained when one solves the problem *as is*, i.e. without any relaxation.) In order to have a tighter bound, we have to spend more computational effort at each node, but we need to evaluate fewer nodes, since we would have pruned most of the tree. On the other hand, if our bounding scheme results in not very tight bounds, we would not be able to prune much of the tree, thus having to evaluate large number of the nodes, which may be worse than total enumeration of all alternatives.

#### **2.4.2 Our naïve branching approach**

Our constraint programming model branches on three sets of data. During the team pattern generation phase, when the constraint store is in a stable state but still contain non singleton domains, we make the leftmost undetermined home/away variable  $X$  in our current pattern the branching point. A new constraint  $c$  and its negation  $\sim c$  which in this case would be  $X = 0$  and  $X = 1$  respectively is separately applied to the constraint store to generate two new unstable stores. These new stores represent the left and right branches of the binary search tree. The distribution strategy of choosing the leftmost undetermined variable as the branching point is known the naïve strategy. During the team pattern assignment phase, the same naïve strategy is used to distribute on the team pattern vector. In this case, the leftmost team without a pattern assigned is selected and the smallest value in the domain of that team's pattern set becomes the branching constraint  $c$ . In the final distribution step, we need to determine all the unbounded team assignments. Therefore, we distribute on the vector

of all “against” variables. The naive strategy is still used here to resolve all the non-singleton domains from left to right, choosing the first available team in the “against” domain. The reason for using the naïve strategy is simple. If we know that giving priority to certain variables and values in our search space during distribution will lead us to a better solution or produce solutions faster, then it makes sense to try and branch on those variables and values as soon as possible. Unfortunately, we do not know of any such ordering strategy for the sequence of distribution.

## **2.5 Lower Bounds for the travelling distances**

### **2.5.1 Graph Model 1 (TSP)**

The idea behind this model comes from our realization that the optimisation component of our task is very similar to the well-known travelling salesman problem. Therefore, we designed a graph representation of the “tour” of a particular team.

#### **Graph Model 1**

**Input:** Team number (Our salesman’s id), a vector of 0/1 values representing whether the team plays at home, a vector of finite domain variables holding potential opponents for each round.

**Output:**  $G(V, E)$

$v \in V \mid v$  is a start or end node or  $v$  is a match node. If  $v$  is a match node is either a home or away match node.  $|V|$  is NoOfRounds+2.

$e \in E \mid e$  exists for  $v_1$  and  $v_2$  if and only if the team can play the match  $v_1$  and the match  $v_2$  in consecutive rounds.

**Example of a *tour* for team 1 in NL4 instance**

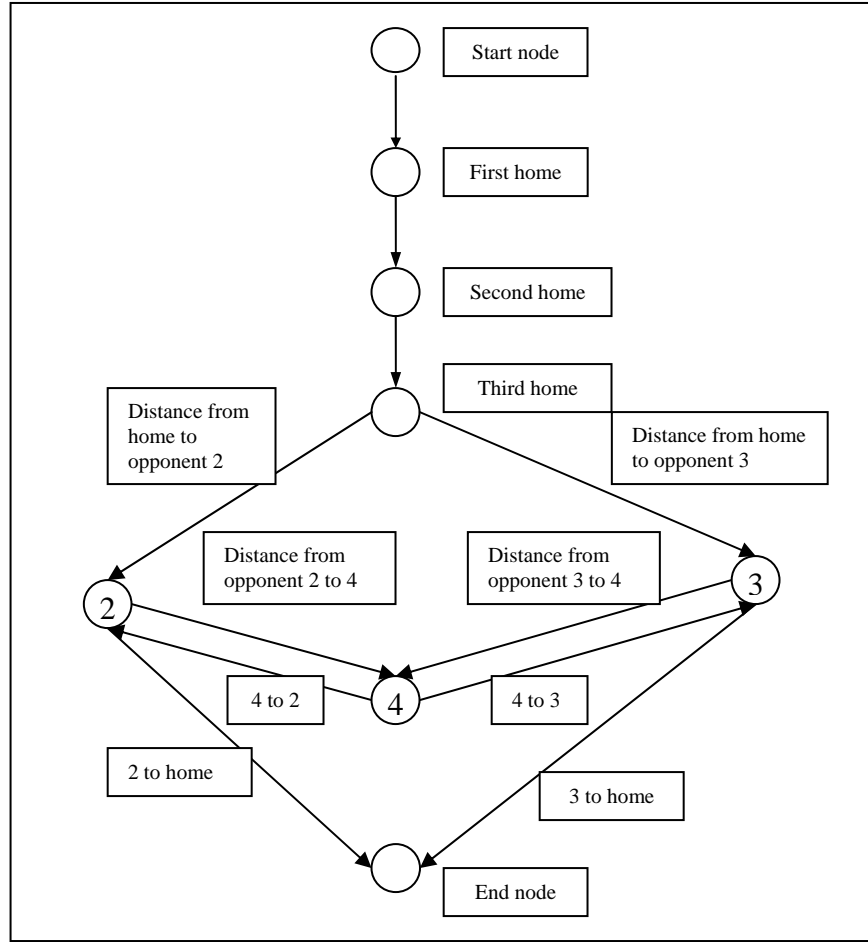


Figure 1 *Tour* based graph

Team = 1 Home Pattern = {1, 1, 1, 0, 0, 0} Against = {{2, 4}, {3, 4}, {3}, {2, 3}, {4}, {2, 3}}

1a = start node. 1e = end node, 1b – 1d = the home game nodes, 2 – 4 represents the away games at opponent 2 – 4 respectively.

The graph captures possible travel distances based on the values reflected by the current constraint store. Opponent information is ignored whenever we encounter a home game since the location is determined solely by the home value. For home nodes, we are concerned only with the sequence in which each home game/node is connected. For away games, the opponent information is needed to determine which away location to visit. The start node is a dummy node for enforcing the constraint of starting out from home, while the end node is a dummy node for enforcing the



constraint of going back to home after end of tournament. We add a dummy edge from the end node to the start node to allow a complete tsp tour.

Clearly, the cost of an optimal TSP tour of such a graph would be strictly lower than or equal to the minimum travel distance possible for the team given current opponent assignments. Unfortunately, the TSP itself is also a combinatorial optimisation problem. Being part of the class of NP-Hard problems, there is little chance we can find a polynomial-time algorithm for solving the problem. The resulting delay in time might override any possible gains from the lower bound. Therefore, we decided to find a lower bound for the TSP instead. The bound chosen was the weight of a minimum spanning tree of our graph. This constitutes our first lower bound which is twice relaxed from the original TTP constraints. In order to find the directed minimum spanning tree for our graph, I implemented Edmonds' algorithm for the directed minimum spanning tree problem.

#### **Edmonds Algorithm for directed MST**

1. Discard the arcs entering the root if any; For each node other than the root, select the entering arc with the smallest cost; Let the selected  $n-1$  arcs be the set  $S$ .
2. If no cycle formed,  $G(N,S)$  is a MST. Otherwise, continue.
3. For each cycle formed, contract the nodes in the cycle into a pseudo-node ( $k$ ), and modify the cost of each arc which enters a node ( $j$ ) in the cycle from some node ( $i$ ) outside the cycle according to the following equation.
4.  $c(i,k)=c(i,j)-(c(x(j),j)-\min_{\{j\}}(c(x(j),j)))$  where  $c(x(j),j)$  is the cost of the arc in the cycle which enters  $j$ .
5. For each pseudo-node, select the entering arc which has the smallest modified cost; Replace the arc which enters the same *real* node in  $S$  by the new selected arc.
6. Go to step 2 with the contracted graph.

The key idea of the algorithm is to find the replacing arc(s) which has the minimum *extra cost* to eliminate cycle(s) if any. The given equation exhibits the associated extra cost. The following example illustrates that the contraction technique finds the minimum extra cost replacing arc (2,3) for arc (4,3) and hence the cycle is eliminated.

#### **2.5.2 Graph Model 2 (Shortest Path Algorithm)**

This model is a straightforward mapping of the constraints store (variables) into a directed graph. The graph reflects the sequential flow of the matches and the possible paths of progression through the matches. This is an attempt to create a better and more reliable model to replace the “tour” graph. We have a start and end node just like the first model. We also use one node to represent each home game to be played. However, we create a new node for every opponent we face in an away game. For example, if round two is an away game with three possible opponents, we will create exactly three new nodes. The most obvious difference between the two models in the number of nodes generated given the same input. This model preserves information about the rounds in which particular matches may occur. Therefore, it tends to be more constrained than the first model.

### **Graph Model 2**

**Input:** Team number (Our salesman’s id), a vector of 0/1 values representing whether the team plays at home, a vector of finite domain variables holding potential opponents for each round.

**Output:**  $G(V, E)$

$v \in V$  |  $v$  is a start or end node or  $v$  is a match node. If  $v$  is a match node is either a home or away match node.  $|V|$  is equals to  $(\text{NoOfRounds}/2 + 2) +$  the number of possible opponents for all away games.

$e \in E$  |  $e$  exists for  $v_1$  and  $v_2$  if and only if the team can play the match  $v_1$  and the match  $v_2$  in consecutive rounds.

### **Example of a *tour* for team 1 in NL4 instance**

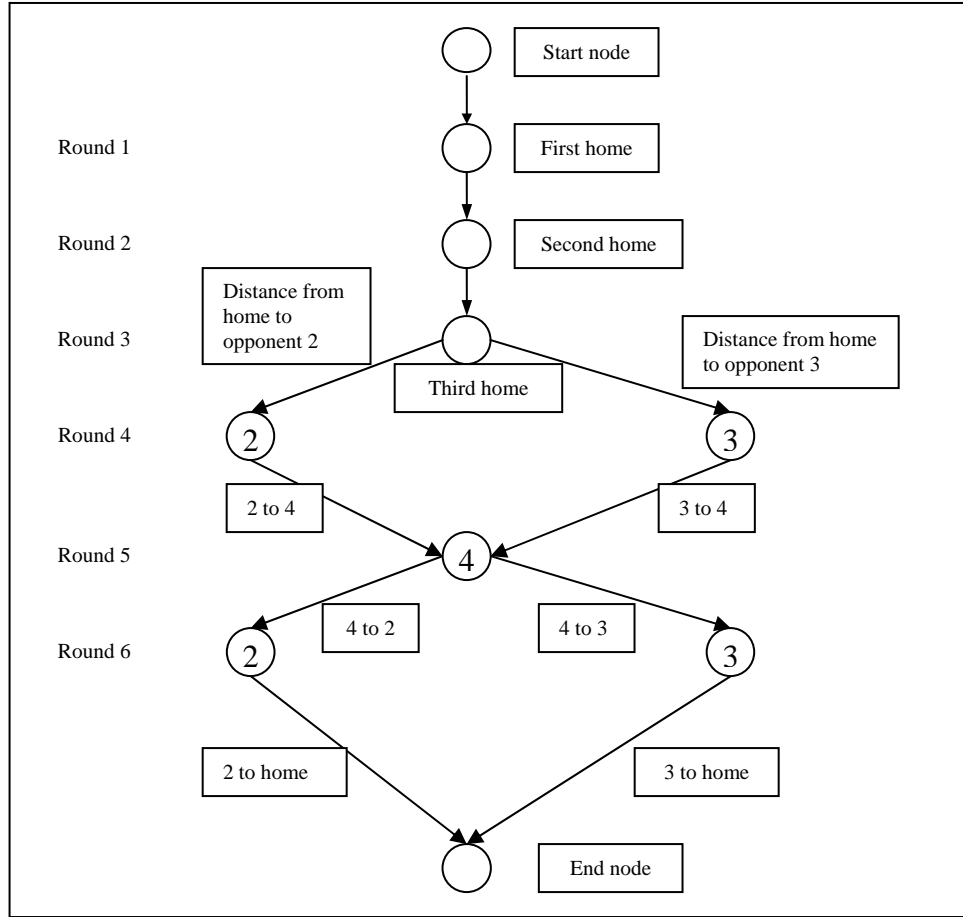


Figure 2 The Round Based Model

Team = 1 Home Pattern = {1, 1, 1, 0, 0, 0} Against = {{2, 4}, {3, 4}, {3}, {2, 3}, {4}, {2, 3}}

Each node is a start, end or game node. A game node  $v$  situated  $I$  steps away from the start node represents the possibility of the team playing to a match in round  $I$  at a specific location. i.e. nodes are match round and location specific. Each edge has a weight corresponding to the travel distance from the match location of the source node to the match location of the end node. For all edges  $e$ , if the source node of  $e$  is in round  $I$  then the target must be in round  $I+1$ . For any node  $v_i$  where  $i$  is the round number,  $v_i$  must have an out-edge to all nodes  $v_{i+1}$  in the succeeding round. The exception is the case when away games are being played in both round  $I$  and  $I+1$ . In such a case, there must not be an edge between  $v_{i,j}$  and  $v_{i+1,j}$  with  $i$  being the round number and  $j$  being the opponent team number, since a team cannot play away against

an opponent more than once. However, this constraint is not enforced when the away matches are not consecutive. The graph is simple (no parallel edges), loopless and acyclic. Since the structure of the graph ensures that any edge followed will always lead you further away from the source node and closer to the end node, a shortest path algorithm is sufficient to find a lower bound for the team “tour”. The LEDA implementation of Dijkstra’s shortest path algorithm is used. The algorithm is given here for completeness.

### **Dijkstra's Algorithm**

1. Set  $i=0$ ,  $S_0 = \{u_0=s\}$ ,  $L(u_0)=0$ , and  $L(v)=\text{infinity}$  for  $v \neq u_0$ . If  $|V| = 1$  then stop, otherwise go to step 2.
2. For each  $v$  in  $V \setminus S_i$ , replace  $L(v)$  by  $\min\{L(v), L(u_i)+d_{v u_i}^u\}$ . If  $L(v)$  is replaced, put a label  $(L(v), u_i)$  on  $v$ .
3. Find a vertex  $v$  which minimizes  $\{L(v): v \in V \setminus S_i\}$ , say  $u_{i+1}$ .
4. Let  $S_{i+1} = S_i \cup \{u_{i+1}\}$ .
5. Replace  $i$  by  $i+1$ . If  $i=|V|-1$  then stop, otherwise go to step 2.

The time required by Dijkstra's algorithm is  $O(|V|^2)$ . It will be reduced to  $O(|E|\log|V|)$  if heap is used to keep  $\{v \in V \setminus S_i : L(v) < \text{infinity}\}$ .

### **2.5.3 Artificial lower bounds (testing purposes)**

As a verification of the strength of a tight lower bound in reducing the search effort, artificial lower bounds of varying tightness were tested for team distances.

## **2.6 The LEDA programming library**

The following excerpt is taken from the LEDA documentation.

“One of the major differences between combinatorial computing and other areas of computing such as statistics, numerical analysis and linear programming is the use of complex data types. Whilst the built-in types, such as integers, real numbers, vectors, and matrices, usually suffice in the other areas, combinatorial computing relies heavily on types like stacks, queues, dictionaries, sequences, sorted sequences, priority queues, graphs, points, segments. The Library of Efficient Data types and Algorithms (LEDA) is a C++ class library for efficient data types and algorithms.

LEDA provides algorithmic in-depth knowledge in the field of graph and network problems, geometric computations, combinatorial optimisation and others. LEDA is implemented following the object-oriented approach. It is available in four different packages: basic, graph, geometry and GUI.”

LEDA provides a set of easy to use, highly efficient and proven correct functions and complex data types to the average C++ programmer. Our propagator is based on the idea of reducing the TTP into a common graph problem so that we can use apply concepts of graph theory to our problem. Instead of having to implement our own graph object and Dijkstra’s shortest path algorithm, we were able to make use of the LEDA types and functions.

# Chapter 3

## Computational Results

### 3.1 Experimental results with timing

| NL4                | Time   | Explored | Solutions | Failed | Depth | Best Result |
|--------------------|--------|----------|-----------|--------|-------|-------------|
| No bounds          | 11.08s | 3301     | 5         | 3297   | 41    | 8276        |
| 1 <sup>st</sup> LB | 13.27s | 2491     | 5         | 2487   | 41    | 8276        |
| 2 <sup>nd</sup> LB | 20.84s | 2605     | 5         | 2601   | 41    | 8276        |

Table 7 Results for NL4 instances

| NL6                | Time    | Explored | Solutions | Failed | Depth | Best Result |
|--------------------|---------|----------|-----------|--------|-------|-------------|
| No bounds          | 16m 17s | 119825   | 12        | 119811 | 566   | 27039       |
| 1 <sup>st</sup> LB | NIL     | 278      | 2         | 270    | 45    | 29257       |
| 2 <sup>nd</sup> LB | 16m 21s | 53121    | 9         | 53105  | 292   | 27282       |


Table 8 Results for NL6 instances

**Note:** 1<sup>st</sup> LB propagator had to be terminated due to error for NL6

| NL4 with Special bounds    | Time   | Explored | Solutions | Failed | Depth | Bound (Team) |
|----------------------------|--------|----------|-----------|--------|-------|--------------|
| Normal                     | 7.91s  | 2451     | 5         | 2447   | 41    | LB 1500      |
| 1 <sup>st</sup> propagator | 10.63s | 2110     | 5         | 2106   | 41    | LB 1500      |
| 2 <sup>nd</sup> propagator | 16.40s | 2077     | 5         | 2073   | 41    | LB 1500      |
| Normal                     | 2.28s  | 644      | 5         | 640    | 41    | LB 2000      |
| 1 <sup>st</sup> propagator | 3.04s  | 629      | 5         | 625    | 41    | LB 2000      |
| 2 <sup>nd</sup> propagator | 4.40s  | 537      | 5         | 533    | 41    | LB 2000      |
| Normal                     | 2.23s  | 649      | 3         | 647    | 41    | UB 2200      |
| 1 <sup>st</sup> propagator | 3.10s  | 639      | 3         | 637    | 41    | UB 2200      |
| 2 <sup>nd</sup> propagator | 4.55s  | 533      | 3         | 531    | 41    | UB 2200      |
| Normal                     | 7.04s  | 1677     | 5         | 1673   | 41    | UB 3000      |
| 1 <sup>st</sup> propagator | 8.68s  | 1620     | 5         | 1616   | 41    | UB 3000      |
| 2 <sup>nd</sup> propagator | 12.09s | 1531     | 5         | 1527   | 41    | UB 3000      |

Table 9 Results for tests with artificial bounds

The timing given is the time OZ spends on exploration alone. Overheads such as garbage collection are not reported. Therefore, the actual run time involved is significantly longer. The propagator based on the first graph model and the minimum spanning tree does not work for the NL6 instance. The results from the two other cases were recorded after three hours of computation by halting the execution. Both searches were resumed after the results were taken. The normal branch and bound model will eventually terminate due to segmentation error. The 2<sup>nd</sup> propagator is capable of obtaining the best result of the normal model. However, it takes very much longer. The longest time the propagator has been left running non-stop was more than 48 hours. Unfortunately, the client running Oz got disconnected from the Linux server and no results beyond 12 solutions have been recorded.



```

[timetable{chosen:chosen(1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1)
  inRound:inRound(round{date:1
    playsTeam:playsTeam(team{against:4 away:1 home:0}
      team{against:3 away:0 home:1}
      team{against:2 away:1 home:0}
      team{against:1 away:0 home:1}))
    round{date:2
      playsTeam:playsTeam(team{against:2 away:1 home:0}
        team{against:1 away:0 home:1}
        team{against:4 away:1 home:0}
        team{against:3 away:0 home:1}))
    round{date:3
      playsTeam:playsTeam(team{against:3 away:1 home:0}
        team{against:4 away:1 home:0}
        team{against:1 away:0 home:1}
        team{against:2 away:0 home:1}))
    round{date:4
      playsTeam:playsTeam(team{against:2 away:0 home:1}
        team{against:1 away:1 home:0}
        team{against:4 away:0 home:1}
        team{against:3 away:1 home:0}))
    round{date:5
      playsTeam:playsTeam(team{against:4 away:0 home:1}
        team{against:3 away:1 home:0}
        team{against:2 away:0 home:1}
        team{against:1 away:1 home:0}))
    round{date:6
      playsTeam:playsTeam(team{against:3 away:0 home:1}
        team{against:4 away:0 home:1}
        team{against:1 away:1 home:0}
        team{against:2 away:1 home:0})}}
  teamDist:teamDist(2011 2011 2127 2127)
  teamPattern:pattern(1 17 4 20)
  totalDistance:8276}]

```

Figure 3 The optimal timetable for NL4

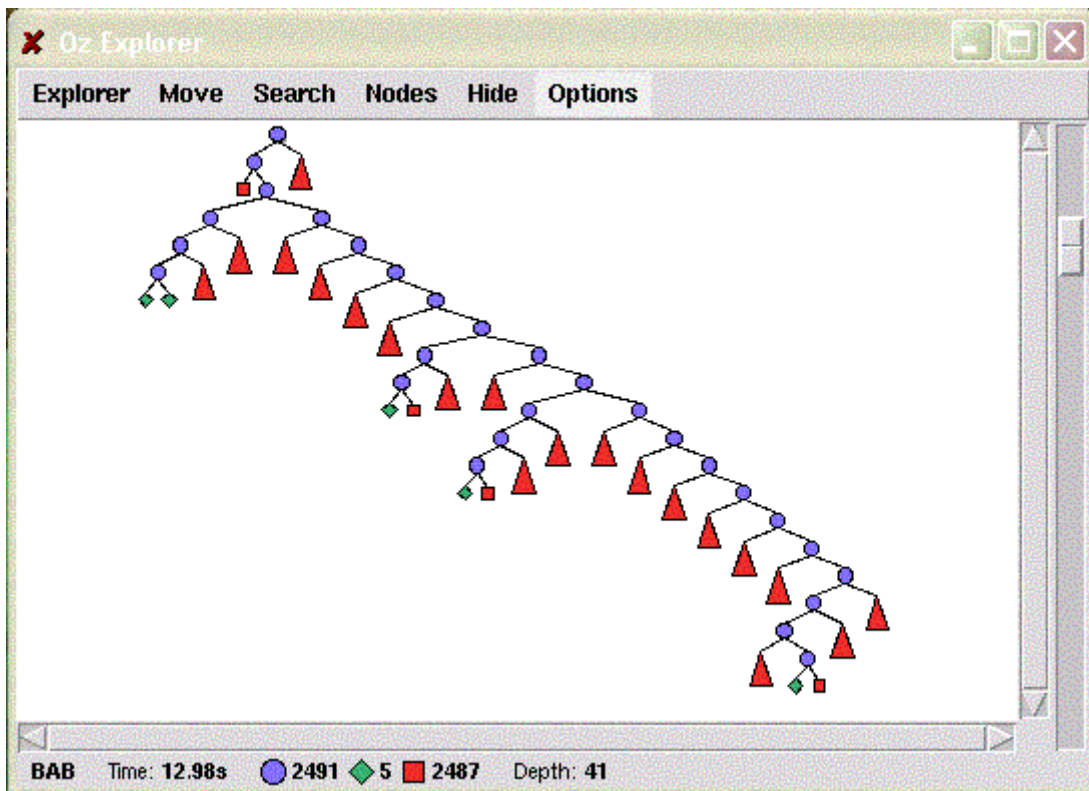


Figure 4 The complete search tree for NL4 with 1<sup>st</sup> propagator

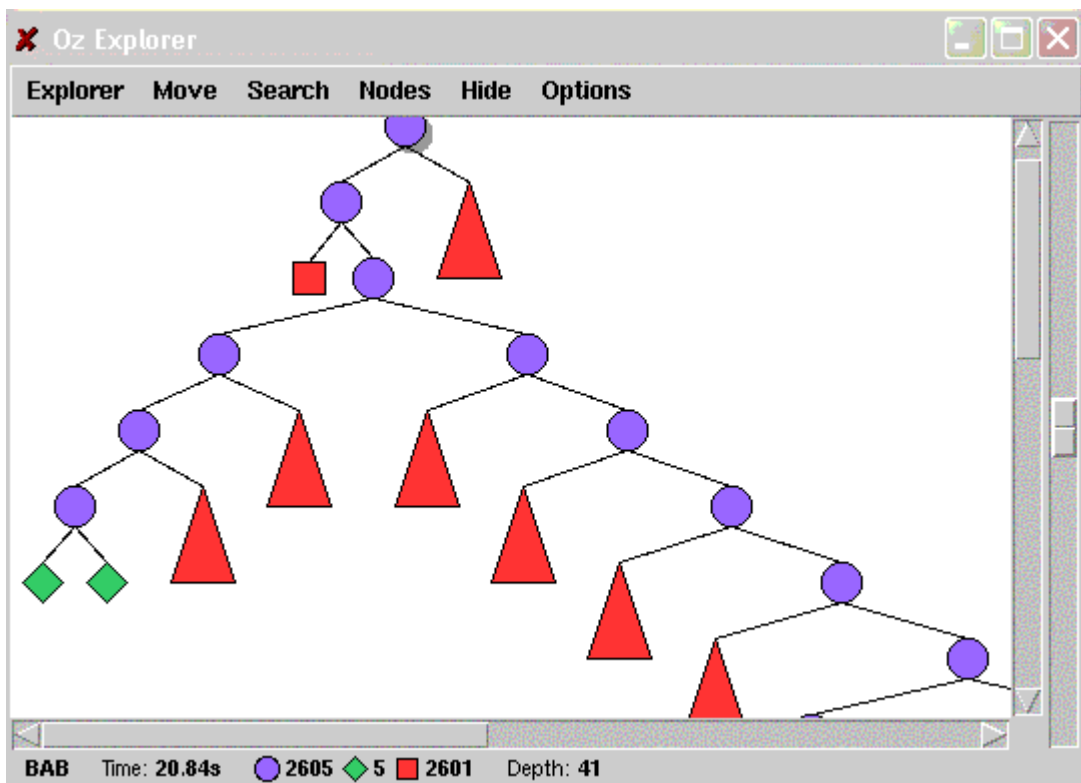


Figure 5 Part of the search tree for NL4 with 2<sup>nd</sup> propagator



# Chapter 4

## Discussion

### 4.1 Graph Model 1

The first model is space efficient since it requires only  $(N+2)$  nodes for all instances of  $N$  teams. However, we can get a high number of cycles within the graph which in turn complicates the process of extracting a tight lower bound. For example, a graph representing a state of propagation where the *Against* variables have not been reduced to singletons, might often contain multiple cycles, some between just two nodes. A number of problems are evident when we try to construct the minimum spanning tree (MST). Since the MST always chooses the smallest cost edges to span all nodes regardless of any implicit constraints that might exist, the resulting tree is often dissimilar to any real *tour* of the team. In these cases, we may end up with a loose travel distance lower bound.

The graph does not retain the match round aspect of the problem. As long as the team can play two particular opponents away in any two consecutive rounds, an edge will exist between the two opponent nodes in the graph. Therefore, any MST constructed from this graph will have relaxed constraints on the order of play. The MST's allowance for nodes to have multiple out-edges also violates the implicit constraint that a team's tour must be sequential. (see Figure 6)

The large number of cycles formed can affect the speed of the propagation since all cycles must be resolved by the MST algorithm. In fact, the NL6 instance had caused infinite looping in my implementation of the algorithm. In an attempt to remove a cycle from the tree, the algorithm incidentally creates a new cycle. During the removal of the second cycle, the first cycle is actually recreated. I attempted to get around the problem by permanently deleting the edge selected to break a cycle. Unfortunately, the removal of such edges actually caused some nodes to become unreachable. Even after studying the algorithm thoroughly, I was unable to discover the source of the problem. The propagator performs quite well in the NL4 instance giving a fair trade between increased delay in execution time and additional pruning of the search tree.

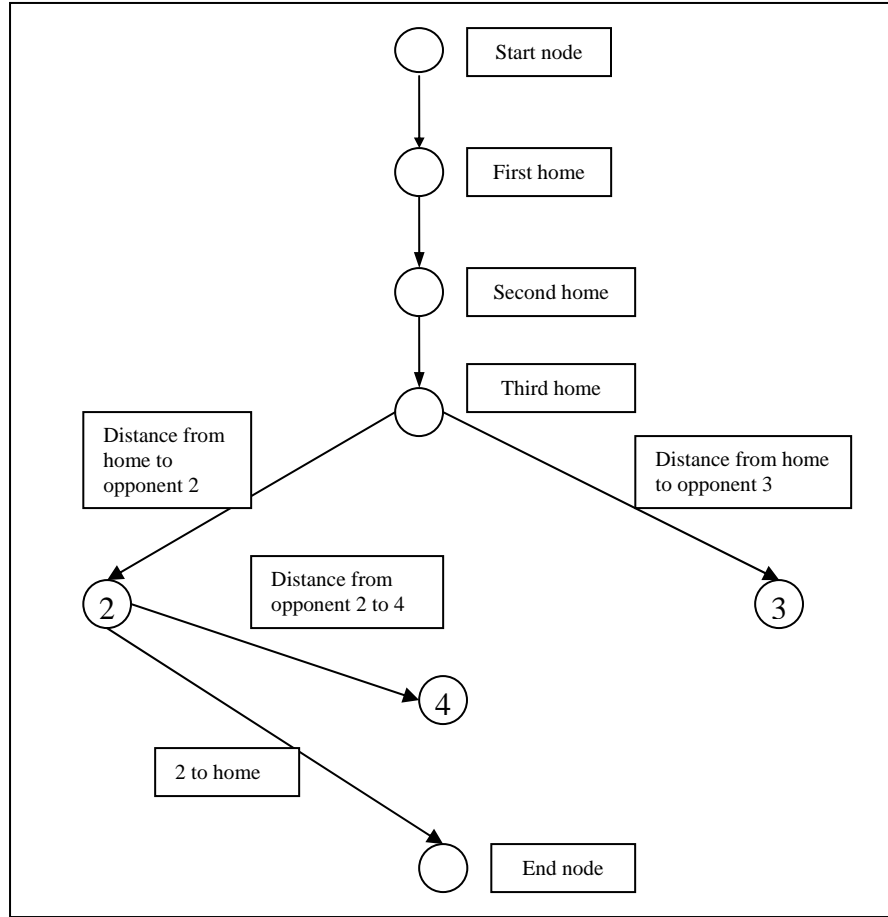


Figure 6 Minimum Spanning Tree

## 4.2 Graph Model 2

The second graph models our problem more closely with both the round and location information incorporated within the nodes. Since the nodes represent not only a game's location but also the round in which it occurs, the sequence in which a team can visit its opponents is well-constrained. As a further step to strengthen the model, we disallow any edges between all away nodes of any particular opponent. This step makes it impossible for a path to "visit" the same away opponent twice in a row. However, the graph is still a relaxation from the original problem where all paths are distinct and non-overlapping. The main effect of this relaxation becomes clear when we attempt to compute the shortest path from the start node to the end node as a lower bound. It is possible for us to get a path where the team visits the same opponent more than once in away games as long as the away rounds concerned are not consecutive (see Figure 8). In this case, the input values from the constraint store have team 2 as a possible opponent of team 1 in rounds 4

and 6. Both rounds happen to contain away games as well. In such a situation, both the paths in Figure 7 and 8 are possible. However, the path in Figure 8 is clear incorrect and most likely produces a very poor bound.

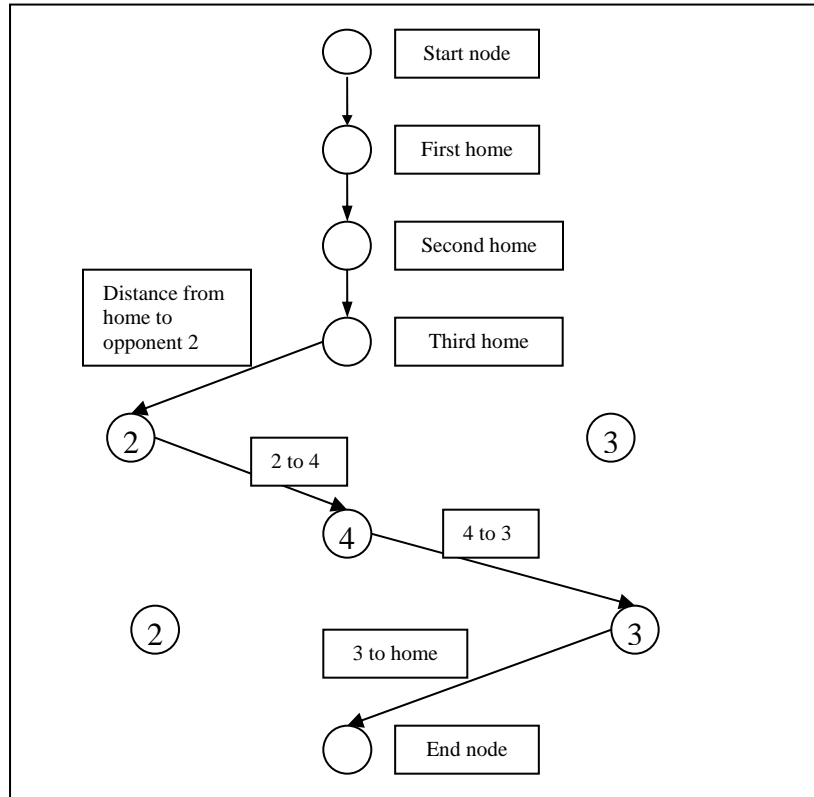


Figure 7 A shortest path from start to end

This model is simple to visualise and understand. The propagator implementing the model is more reliable than the first propagator. Unfortunately, this model seems to perform poorly when compared with the first model. From the NL4 instance, the second propagator takes almost twice the time and does less pruning than the first propagator. The poor pruning is most likely the result of the relaxed constrained discussed earlier. The delay in execution time could in part be due to the shortest path algorithm. We should consider increasing the speed of the shortest path algorithm or choose another approach to compute a lower bound for the graph. Based on the results from the NL4 instance, it appeared that this model is inefficient and not really useful. However, the tests with lower bounds returned some interesting results that suggest more research on this model is necessary.

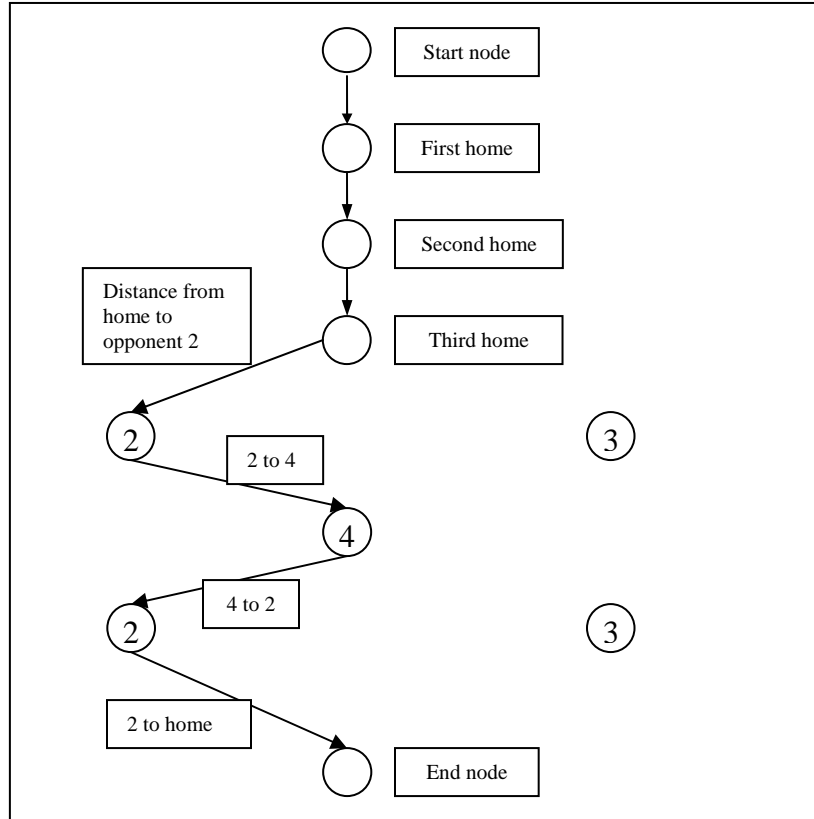


Figure 8 A problematic shortest path

### 4.3 Artificial bounds

After examining the results of the searches with the new lower bound propagator, we were quite disappointed. Neither of the two models produced impressive pruning or reduced running time. The second model which we had hoped would outperform the first was a total disappointment. We began having doubts on the effectiveness of our approach. Therefore, we began experimenting with explicitly declared upper and lower bounds for team travelling distances.

The results we obtained from these experiments were interesting not because of the pruning achieved, that part was to be expected. The effect that all the bounds seem to have on the performance of the second propagator was on the other hand quite unexpected. In all four variations of artificial bounds testing, the second propagator outperformed the first in terms of pruning. After careful consideration, I have a hypothesis on the reason for this change in performance. The second propagator might not be a total failure but has a critical weakness. It may be the relaxed constraint on playing opponents more than once away which is at fault. I believe that the second

propagator may actually be producing a tighter lower bound than the first in certain cases while having very poor bounds in others. The existence of artificial bounds overwrites the worst case lower bounds for both models. Therefore, only “reasonable” lower bounds from the propagators play a factor. Since we know that the second propagator actually models the problem more precisely, it is conceivable that the second propagator should produce tighter bounds. The worst cases come about when we have the scenario depicted in Figure 8. The effect is especially bad if the distances to the alternative opponents are very unbalanced. i.e the distance for the “repeated” visit is a lot lower than the correct alternative. Therefore, the key to improving the second model is to eliminate or reduce the occurrence of the “repeated” visits.

From the experiments, we confirm that strong lower bounds are effective in reducing unnecessary computation of other possible solutions once the optimum is reached. Interestingly, a good upper bound on the team distance also has a strong effect on reducing search beyond the optimum. My explanation for this effect is that these upper bounds actually produce tight lower bounds because of the sum constraint between the total travelling distance and the individual team distances. The upper bound on the team distance means that the other teams must have a minimum distance for them to achieve the total distance’s upper bound. However, I may be wrong and the upper bound of the total distance may instead be reduced by the constraint.

#### **4.4 Discussion of results**

All variants of our solver were able to reach the optimal solution for NL4 fairly quickly. The normal constraint solver using branch and bound without additional bounds managed to outperform our “improved” models in terms of execution time. This was quite disappointing for me. The two propagators did manage to perform significant pruning of the search tree giving us about 21% - 25 % reduction in the number of explored nodes.

Without the lower bounds, our solver managed to reach 12 solutions before failing for NL6. The shortest travelling distance obtained was 27039. Unfortunately this is still a sizable deviation of 13.1% from the optimal solution 23916. With the second propagator, we managed to get comparable results, however the solver was exploring the nodes of the search tree at less than half the original speed. As previously mentioned, we were unable

to complete the execution of any variants of the solver for NL6. However, the execution of the variant with the second propagator did not terminate due to internal errors. Therefore, it is still likely that this variant might be able to obtain better results or even the optimal solution for the NL6. However, given the existing results, we must accept that the NL6 instance is not solvable by our system. The new lower bounds formulated while having a noticeable effect, are still not tight enough to perform the pruning necessary for our system to solve to optimality, instances greater than NL4.

# Chapter 5

## Conclusions

### 5.1 Conclusion on the lower bounds

We have managed to obtain a tighter lower bound by solving the minimum spanning tree and shortest path problems for the two dynamically generated team constrained graph models. As mentioned in preceding sections, while the lower bounds have a significant effect in the reduction of search space by allowing us to prune off branches of our search before completing team assignment, the effect becomes negligible as soon as the number of teams playing increases. The overall increase in complexity in larger problem instances far exceeds the extra pruning we achieve. Even for the NL4 instance where we were able to obtain the optimal solution, the slowdown in execution speed surpassed any gains from pruning the search space.

From our observations, the main weakness of our solver is the slow rate of convergence of the lower and upper bounds. Given the complexity of our problem, I am doubtful of the tightness we can possibly achieve through modelling. Even if we created an exact model which allows only feasible paths and found the shortest path in it (getting the exact minimum for any feasible *tour* by the team), there could still be a huge difference between our *perfect* lower bound and the actual team distance in the optimal solution. The optimal solution for NL4 is a good example of this situation. The team distances making up the optimal total distance are clearly non-minimal. Therefore, I strongly believe that there is a limit to the degree of tightness we can achieve for lower bounds.

The reason we did not start with an exact graph model in the beginning was our concern over the time and space complexity involved. For the exact model, we need extend the second model by creating separate sub-trees for each away node encountered. For example, if team 1 can play away against teams 2, 3 and 4 in round 1, we create three opponent nodes. In round 2, we need to create either a home node for home games or more opponent nodes for away games just as in the second model. However, we now create a set of the node(s) for round two for each of the opponent nodes in round 1. For each set, we exclude an opponent from all future away games if the opponent has appeared in an earlier away game. Hence, branches of opponent nodes never overlap and

the constraint on playing away against each opponent exactly once is enforced. With the exponential growth in the number of nodes, the construction of the graph and the solving of the shortest path problem will certainly take a longer time and more space.

I concluded that we cannot expect great results purely with a computed lower bound. For the lower bound approach to perform well, we need to find very tight bounds on the team distance in very short time. From the discussion above, we have showed that such a bound will be difficult if not impossible to achieve. Even if we do develop a good lower bound algorithm, its performance will always degrade with larger instances since the large number of possibilities will guarantee a wide range of possible total distances.

At the other end of the scale, we also observed the effect of the upper bound of search tree pruning. Using the best solution found as an upper bound, we ensure a steady approach towards optimality. However, in large instances, this stepwise improvement will probably take a very long time. This is very clearly seen in the NL6 instance where the rate of reduction in the total distances of the non optimal solutions is in the range of tens or hundreds while the difference between the optimal and the current solution are in the range of thousands. In the event we achieve a near optimal solution early in the search process, we get a very good upper bound. Such a bound coupled with our lower bound propagator will allow us to perform extensive pruning of our search tree. However, it is more likely that our initial solution is quite poor and we end up going through a large number of poor solutions with very slow convergence towards the optimal solution.

In conclusion, we managed to formulate and test two different graph algorithms for the TTP and analysed their effectiveness. We have learned that the tightening of the bounds of individual team distance has a greater effect in reducing search complexity for the TTP than bounds on the total travelling distance of the tournament. Therefore, we need to find an upper bound for team distances to complement their lower bounds and the upper bound for total distance. We have shown that the graph based approach to TTP optimisation has the potential of producing good solutions. However, given the magnitude of the TTP, a more aggressive approach to reducing the upper and lower bound gap quickly has to be the main focus for future approaches to make optimality computation a reality for large instances.



# Chapter 6

## Future Work

### 6.1 Possible approach 1

After analysing our approaches and results carefully, I have developed two new approaches that may lead to good improvements on the time and space efficiency of our system. The first one uses the binary search algorithm to try and speed up the closing of the gap between the upper and lower bounds. The idea is very simple. Once we get an initial solution, we have an upper bound of a domain in which the optimum must lie. In most cases, this initial domain is large and we face the possibility of slow search for the hidden optimum. We apply the binary search algorithm at this stage to try and force a rapid convergence of the upper and lower bounds. Since we are trying to find the minimal travel distance of feasible timetables, we split the domain in half and search for a new solution; starting from the lower domain. Each time a new solution is found in a sub-domain, our domain is constrained in between the solution and the lower bound of the sub-domain.

#### Binary Search for gap reduction algorithm

1. Find initial solution using constraint programming. Set lower bound to 0 or predetermined value. Set upper bound to solution found.
2. Constrain the finite domain of total distance with our bound values.
3. Split the finite domain into two equal sub-domains.
4. Find new solution within the lower sub-domain.
5. If not found, find new solution within the upper sub-domain.
6. If found, set total distance domain to sub-domain with solution and use the solution as a new upper bound. Goto step 3.
7. If no solution found in either sub-domain, the optimal solution is found.

### 6.2 Possible approach 2

The second approach is based on the idea of breaking a large NLx instance into a series of smaller NLx instances. The optimal solution to a sub-problem (smaller instance) is

lesser than the optimal solution to the problem. We can compute the optimal solution for next smaller NLx instance to use as a lower bound for the current NLx instance.

### **6.3 Possible approach 3**

My third idea is simply a combination of a number of previous approaches. Given the good improvements demonstrated for each of the past approaches, it is logical to try and combine their approach to maximise their effect. For example, our lower bound propagator could be combined with sorting methods designed to produce good initial solutions. Once we get a good initial solution, we automatically get a strong upper bound to prune the search. We could also use the approximation algorithms to derive a near optimal solution and proceed with branch and bound search from that point.

## References

- Easton K., Nemhauser G. and Trick M. (2001). The Traveling Tournament Problem Description and Benchmarks. In Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, (CP 2001, Paphos, Cyprus), pp. 580-584.
- Henz M. (1999). Constraint-based Round Robin Tournament Planning. Proceedings of the 1999 International Conference on Logic Programming, (Las Cruces, NM).
- Henz M. (2000). Scheduling a Major College Basketball Conference: Revisited. Operations Research, 49, pp. 163-168.
- Henz M, Müller T, Thiel S. (2002) Global Constraints for Round Robin Tournament Scheduling. European Journal of Operational Research (EJORS).
- Henz M, Müller T. (2000). An Overview of Finite Domain Constraint Programming. APORS 2000.
- Nemhauser G. and Trick M. (1998). Scheduling a Major College Basketball Conference. Operations Research, 46, pp.1-8.
- Ng K B, Choi C W, Henz M. (2002). A Software Engineering Approach to Constraint Programming Systems. APSEC 2002.
- Trick M. (2003). Challenge Traveling Tournament Problems. <http://mat.gsia.cmu.edu/TTP/> or <http://mat.gsia.cmu.edu/TOURN/>.
- Yang S J. (2003). The Directed Minimum Spanning Tree Problem. <http://www.ce.rit.edu/~sjyeec/dmst.html>