Department of Electronic Engineering

PROJECT REPORT

BScIT-2001/02-(CS/HWC)-(CS/HWC-07-BSIT)

AI Scheduling
Using CSP for Sports League

Student Name: LAM Ngai Ming
Student ID: 50135941
Supervisor: Dr CHUN, Andy H W
Assessor: Mr CHENG, L L

Bachelor of Science (Honours) in Information Technology

C O N T E N T S

Table of Contents

I. Abstract

The nine universities in the Atlantic Coast Conference have a basketball competition in which each school plays home and away games against each other over a period. The creation of a suitable schedule of those round robin tournaments is a very difficult problem with a myriad of conflicting requirements and preferences. Different approaches have been proposed in the past to solve round robin sports tournament timetabling problems. One approach is to model this problem as a constraint-satisfaction problem (CSP). Because of the computational complexity of tournament timetabling, the problem is usually broken down into sub-problems and solved separately in steps. Although it is well known that a single-step CSP model does not scale well with increasing number of teams, this project investigates how far we can push the performance limit of a single-step model through the use of different search heuristics. Our approach yields reasonable schedules and better performance with suitable chainable heuristics. Experiments and pilot testing of the system confirm the effectiveness and efficiency of our approach.

Extended Abstract:

A paper entitled "Using Heuristics in Constraint-based Sports Tournament Timetabling ", which was written by Dr. Andy Chun of the Department of Computer Science in the City University of Hong Kong and I, was accepted for presentation in the Sixth World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002) in Orlando, USA. This paper has concluded some of the research and implementation which have been taken throughout the process of this project.

Problem Domain & Description

II. Domain

- Sports League Scheduling of a Major College Basketball Conference

III. Description

The Atlantic Coast Conference (ACC) is a group of 9 universities in the southeastern United States which compete against each other in a number of sports. One of the most significant sports for the ACC in terms of overall revenue is basketball. Television broadcast needs a steady stream of "high quality"

games. And the audience wants neither too much nor too few home games in a row. There are many other heterogeneous requirements and preferences from teams and other parties. The creation of a suitable schedule is thus a very pompous job.

Different approaches have been raised to solve this kind of round robin sports timetabling problems. Existing work on sport tournaments scheduling suggests using only finite-domain constraint programming. However, it still divided the problem into 3 phases of planning process.

Although it is well known that the problem will be highly computationally intensive for large values of teams in a single-step CSP model, this project so far investigates the performance and feasibility of a single-step model through the use of different types of heuristics.

IV. Project Objectives

- Solve sports league scheduling using Constraint Satisfaction Programming
- Implement the problem with a single-step model
- And compare the results with different types of heuristics

V. Why Artificial Intelligence Technology?

There are numerous benefits in using AI techniques for sports league scheduling. Some of the key application payoffs are outlined below:

Increased Revenue

The ACC and its universities earned in excess of $30 million in basketball revenue in 1995. Almost all of its revenue came from television and radio networks showing the matches, and from gate receipts.

These revenue streams are affected by the scheduling of the teams. They need a steady revenue streams which means a steady stream of “high quality”

games. By inputting suitable constraints and requirements from the TV broadcast, this task can be fulfilled and the revenue can be increased.

Preferences & Requirement

In addition to these revenue aspects, there are many different preferences from the teams, and audience. For example, spectators and teams don't want to have too few nor too many home or away games consecutively, some teams may prefer to end the season with a home match for celebration, some teams are traditionally strong teams, and no team wishes to play a series of such team consecutively. These are just a few of the examples of the schedule requirements and effects.

Furthermore, it is a very complex and time-consuming task to combine all of the preferences and requirements, the task of scheduling must be performed by several highly experienced staff without the use of computation effort.

Human Rostering Not "Fair"

Because the preferences, constraints and requirements used by our scheduler for each match are clearly specified and well publicized, bias against any teams is eliminated. We ensure all teams in the conference are treated fairly in terms of assignation of the home, away and bye matches. The schedule generated finally will fulfill all the constraints implemented into the scheduler. And there'll be more than one solution generated for each of the parameters input.

VI. Related Work

Nemhauser and Trick [1] has presented a host of criteria coming from teams, fans and media of the ACC problem. They show how the criteria are exploited in three phases of a solution process, implemented using integer programming and explicit enumeration, and report a "turn-around-time" of 24 hours, which means it takes one day of computing time from specifying/modifying the criteria using feedback from the ACC organizers to proposing new solutions. The model divides the search for a good schedule into three steps:

- A pattern is a string consisting of H (home), A (away), B (bye) of length equal to the number of slots in the schedule. In Step 1, they find a set of *patterns* with cardinality equal to the number of teams. This set is called a *pattern set*. The following is one such pattern for number of teams =9 and number of rounds = 2:

Date	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Pattern	+	-	+	+	-	+	-	-	+	b	+	-	-	+	-	+	b	-

  A set of *n patterns* that satisfies the tournament constraints is called a *pattern set*.

- In Step 2, they assign the teams to the pattern sets consistent with the HAB letters. The result from this stage is a *timetable*.

- In Step 3, they assign the teams to the patterns. Together with the timetable, this gives the *schedule*.

Henz [2, 3, 4, 5, 6] solves this problem by using an alternative approach that is finite domain constraint programming. This approach reduces the turnaround time to within one minute on a PC although it still decomposes the problem into three phases with separate constraint models.

Some other researchers used combinations of Genetic Algorithm and Tabu Search for solving the problem [12, 13].

VII. Theoretical Background

1 Temporally Dense Single Round Robin Tournaments

Our model focuses on solving a specific type of round robin tournaments – temporally dense single round robin tournaments (TDSRR). A temporally dense round robin tournament is one where r n (n-1)/2 matches can be completed within d dates, assuming each team plays at most one match per date.

In fact, round robin tournaments are popular in many sports disciplines. They consist of successions of dates in which each team plays each other team a fixed number of times r. The tournament is said to be single if r is 1, and double if

r is 2. A match can also be divided into home match or away match for a team. A team plays home match if the match is held at it's own stadium, otherwise it is called an away match. If the team doesn't have a match on that date, it is a bye for that team.

Most double round robin tournaments can be reduced to a single round robin problem using *mirroring*, where each date has a mirror date when the same match is played again but at the other opponent's site. Therefore, scheduling the first match automatically schedules the mirror match.

Despite the simplicity of the problem statement, the problem is highly computationally intensive for large values of *n*. Using a single-step CSP model, which we will describe later, and a common search heuristic of selecting the first unbound variable and assigning it the minimum value from its domain, the search stalls even for a very small value of *n=6* and *r=1*.

2 Constraint Satisfaction Problem

In this project, a constraint-based approach in the construction of this timetabling is used. This approach models the ACC problems as a constraint satisfaction problem CSP [10] and using constraint satisfaction tools (JSolver [7]) for system implementation.

A constraint satisfaction problem (CSP) is defined as a 3-tuple (V,D,C), where

- V is a finite set of variables $v_1,v_2,\ldots v_m$,
- D is a function mapping each variable $v \subseteq V$ to a finite set of possible values called the domain of v, and
- C is a finite set of constraints $c_1,c_2,\ldots c_k$.

And CSP is a problem that can be defined in terms of constrained variables and constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a consistent assignment of values to the variables so that all the constraints are satisfied simultaneously.

3 What is Constraint Programming (CP)?

With constraint programming, a problem is represented in terms of its unknowns, that is, its *variables*, and in terms of the *constraints* that must be satisfied by these variables. In other words, the unknowns of a problem are

represented as constrained variables. Thus, for any given problem, the *problem representation* consists of declaring the variables and posting the constraints on them.

Solving such a problem then consists of finding a value for each variable while simultaneously satisfying the constraints. We refer to this activity as the *search* for the solution because when we begin work on a problem, such as the ACC, we don't always know beforehand whether there is a solution that satisfies all the constraints of the problem.

Constraint programming thus entails two relatively distinct activities:

Problem Representation. A problem representation consists of the declaration of the unknowns and the constraints of the problem. This representation is specific to the problem domain under consideration and requires a very expressive programming language to capture that specificity. JSolver uses the object-orientation of Java to make this activity easier: classes of objects are provided for representing unknowns. These objects are called *constrained variables*. With each of these constrained variables, we associate a set of possible values called the domain of the variable. When the domain of a variable contains only one value, we say that the variable is *bound*.

Solution Search. Solving the problem consists of selecting a value in the domain of each constrained variable, so that all the constraints are satisfied. Moreover, JSolver can also be used to search for a solution that follows different chainable heuristics.

For both theoretical and practical reasons, the representation of the problem is separated from the search algorithm from the user's point of view. However, JSolver automatically uses the posted constraints during the search for the solution: it reduces the domains of the constrained variables by removing those values that are inconsistent with the constraints. Thus, we do not have to worry about the constraints; once we have articulated them in the problem representation, they will be handled automatically by JSolver during the problem search.

4 Why Constraint Programming (CP)?

In recent years, constraint programming has proved to be successful to implement algorithms to solve constraint-satisfaction problems (CSP). Many

different types of real-life scheduling, resource allocation and configuration problems can be modeled as CSP.

Resource allocation is a typical application of a CSP. It generally assigns the resource to fulfill the requirements or demands in different situations. On the other hand, timetabling is a type of resource allocation; it assigns the time slots, facilities (stadium) to the teams. By modeling the problem into CSP, we can combine the benefits from Object Technology with Constraint Technology. An object-oriented (OO) methodology can thus be used to support constraint-based system design and development.

5 JSolver-A Java Constraint Programming Class Library

JSolver extends the object-oriented programming paradigm of Java with constraint-based declarative programming. There are a wide variety of constraint programming languages and tools, such as Prolog III [8], CLP, CHIP, or extensions of C++, such as ILOG Solver [9]. These languages and tools, each differing in its constraint domain and serving in different areas of applications, offer an expressive and flexible language for problem specification and heuristics programming. CHIP and ILOG Solver have been applied successfully to solving such industrial applications as car sequencing, routing problem, logistic problem, graph coloring and disjunctive scheduling. However, with a growing number of Java server-side development tools, there is an increasing advantage in using Java to implement application servers for scheduling systems.

Most of the JSolver facility is accessible through the Solver class and the constrained variables are created through methods provided by the JSolver constrained variable classes – Var and VarVector.

```
        public class Example{
            public static void main(String argv[]) throws FailException{
                Solver solver = new Solver();
                Var a = solver.var(0,10, "a");
                Var b = solver.var(0,10, "b");
                solver.post(a.sum(b).eq(6));
                VarVector vars = solver.varVector(a,b);
                solver.solve(solver.generate(vars));
                System.out.println(vars);
            }
};
```

The above example outlines the typical declarative programming style using JSolver. We creates two constrained integer variables a and b, each with a domain from 0 to 10. After that, we post a constraint "*a+b=6*". The constrained variables a and b are then placed into a vector to be used as parameter to *solver.generate*(), which is a goal to instantiate each variable. Goals are the building blocks used to implement search algorithms in JSolver. Both predefined search algorithms and user-defined search algorithms can be expressed in JSolver through goals. The *solver.solve( )* method performs the non-deterministic constraint-based search.

When a constraint is post (*solver.post( )*), the constraint is used immediately to reduce the domains of the constrained variables that it involves, JSolver reduces a domain by removing those values that cannot satisfy the constraint and thus cannot participate in a solution.

Posting a constraint is *reversible* : the constraint is removed when JSolver backtracks to choice points set before that constraint was posted. In case constraint propagation causes a domain to be reduced to a single value, then the constrained variable will be *bound* to that remaining value.

6 Searching for a Solution

6.1 Constraint Propagation

Constraint Propagation is one of the essential techniques of the searching algorithm of the Jsolver (*Interested readers can refer to appendix for CSP Search algorithm*). It is a form of reasoning, using a network of related facts, in which a value, or range of possible values, determined for one variable constrains the possible values of the other variables to which it is related. When the range of possible values of a variable is narrowed, the constraints may allow the ranges of related variables to be narrowed (Domain Reduction), eventually resulting in consistent values or value sets for all variables in the network.

Each time the domain of a constrained variable is modified, the constraints are propagated to further reduce the domain of other constrained variables and cause domain reduction. In fact, it is used mainly to prune the search space until the solution is found. In case a variable has an empty domain, backtracking occurs to get out of dead end.

6.2 Backtracking

When propagating through the search tree, the search may sometimes end up in a dead end. Backtracking allows the search algorithm to get out of the dead end and propagate back to the previous search location for another alternatives (choice point) and continue the non-deterministic search from that point onwards.

One of the most simplest and important examples to show constraint propagation and backtracking is the ***N-Queens*** problem. The N-Queens problem is a well known problem that involves placing N queens on a chess board in such a way that none of them can capture any other using the conventional moves allowed to a queen. In other words, the problem is to select N squares on a chessboard so that any pair of selected squares is never aligned vertically, horizontally, nor diagonally. Stated in this way, the problem looks like a mathematical puzzle, but as a model, it can represent the search algorithm which have been used in our model as well as the constraint propagation algorithm of JSolver.

Of course, the problem can be generalized to a board of any size. In general terms, we have to select *N* squares on a board with *N* squares on each side, still respecting the constraints of non-alignment. We now here use a 4 queens problems to illustrate the constraint propagation with backtracking.

On the diagram below, each node represents successive move (constraint propagation) of the chessboard until we can find the solution that fulfills our constraints. You may notice some of the nodes are marked with a "cross" which means those are the "illegal" move of the model (dead end that violates the constraints). In case of dead end, backtracking allows the search algorithm to get out of this dead end and propagate back to the previous choice points and continue the search onwards. It allows the pruning of search space by domain reduction until the solution "tick" is found.

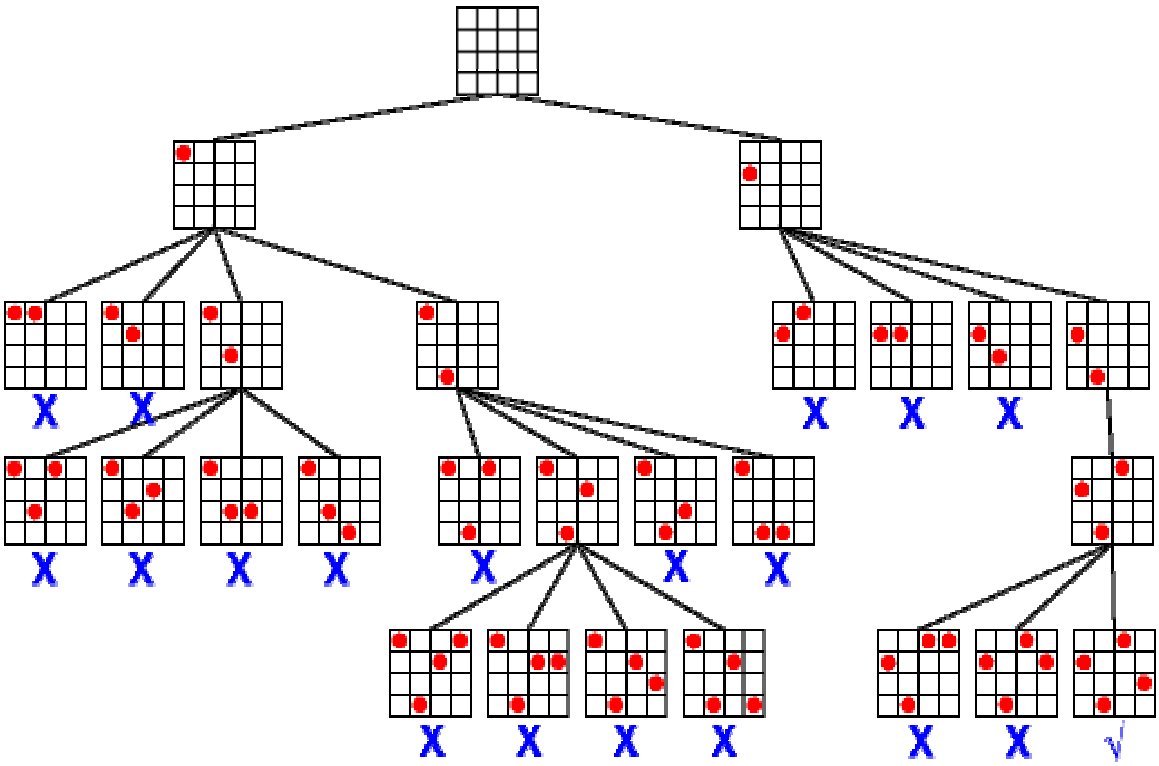


Figure 1 Search Tree of a 4-Queens Problem (Using Backtracking Algorithm)

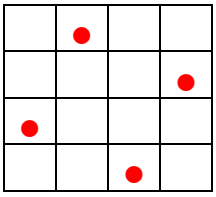


Figure 2 One of the Solutions for N = 4

7 Decrease the Number of Variables - Domain Reduction

The unknowns of a given problem are typically represented in its model by constrained variables. There are practical ways of cutting down on the number of variables and thus reducing the size of the model and its search space.

Problems best solved with constraint-based programming are generally subject to intrinsic combinatorial growth. Even if reducing the domains of

variables by propagating constraints makes it possible to reduce the search space, the initial size of this search space still remains a weighty factor in the execution time.

Consequently, good practice in designing a model should attempt to minimize the size of the search space in the first place. This size increases exponentially with the number of variables. Thus, limiting the number of such variables (even at the expense of enlarging their domains) can reduce the combinatorial complexity.

8 Search Heuristics

A heuristic is a fancy name for a "rule of thumb" - a rule or approach that is used to guide the CSP search. It doesn't always work or doesn't always produce completely optimal results, but which goes some way towards solving a particularly difficult problem for which no optimal or perfect solution is available.

In our Single-Step model, we would like to test how far our performance limit up to by using different chainable search heuristics. Although it is well known that this model doesn't scale well with increasing number of teams, most cases will just lead to simple exhaustive search. As we firmly believe that the "shape" of the CSP search tree is greatly determined by the search heuristics that guide the search, we would like to investigate the performance resulted of using different heuristics. And we trust that, by using a suitable heuristics, the performance will be greatly increased compared with other researchers' model.

Here we would like to outline the effect of using different heuristics to guide the search by several tree diagrams:

First let's imagine the Figure 3 here to be a search tree which represents a search space of a simple algorithm. The search will start with the ROOT node "S" and end with the GOAL node "G", which is the goal of our search. The search will just propagate from node to node until it finds the node "G". In fact, each search will follow some tactics of propagation, that is, heuristics, the rules of thumb of propagation. Let's also assume that we use a "*VISIT-LEFT-MOST-NODE-FIRST-HEURISTIC*" in the first part of our demonstration.

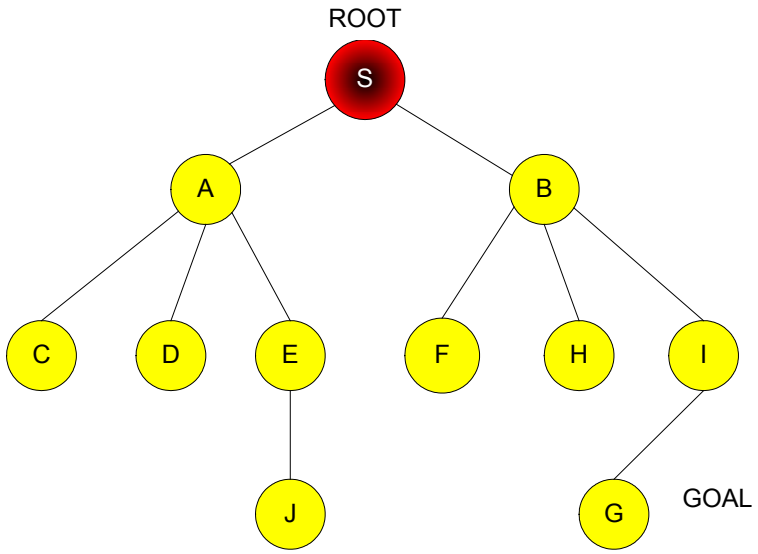


Figure 3 Search Tree - Outline

In Figure 4, the propagation of search is going toward node "A" which is the next target of visit according to our rules of thumb. Apparently, node "A" is not the goal of our search, the searching process will propagate to another node again.

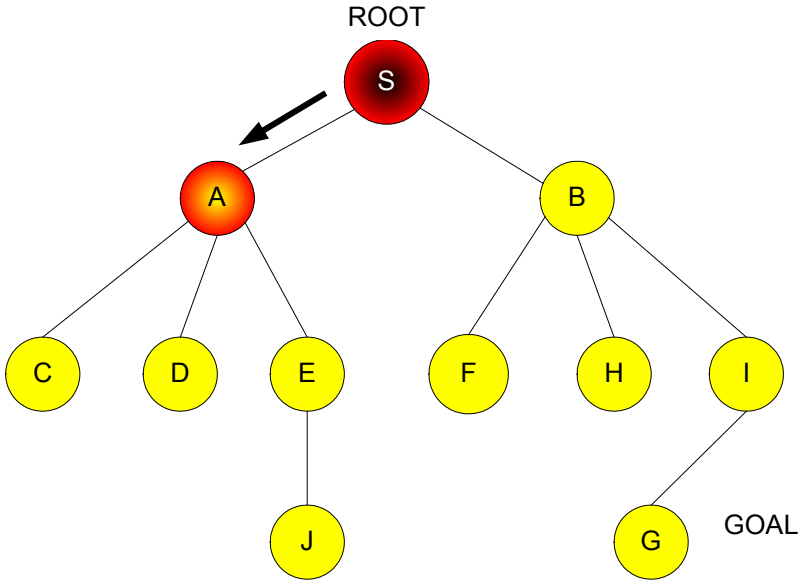


Figure 4 VISIT-LEFT-MOST-FIRST-HEURISTICS -"A"

In Figure 5 here, the propagation continues until it reaches the leaf of the search tree which is node "C" and finds that node "C" is not the goal of our solution. In constraint propagation of JSolver, the search will *backtrack* to the previous choice point to get out of dead end.

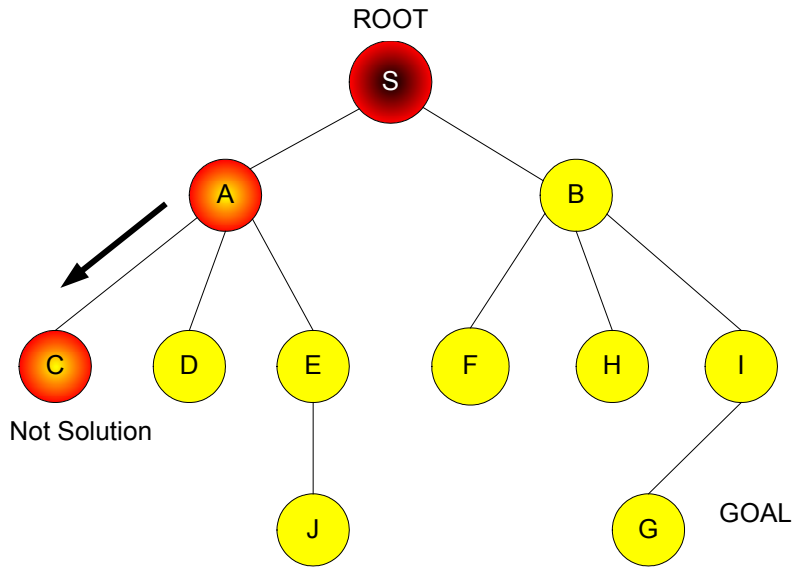


Figure 5 VISIT-LEFT-MOST-FIRST-HEURISTICS -"C"

Here you can see the backtracking algorithm is used here in Figure 6, the search propagates back from node "C" to node "A" and continues from here again.

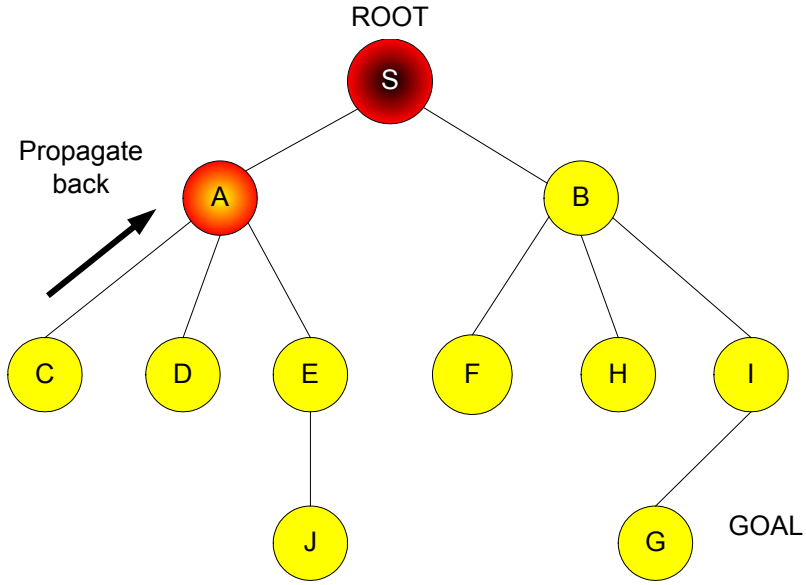


Figure 6 VISIT-LEFT-MOST-FIRST-HEURISCTIS - Propagate Back

In Figure 7, the propagation finds another leaf here but not the goal "G". It will again backtrack to the previous choice points and propagate to other choice points until it finds the goal "G" at last.

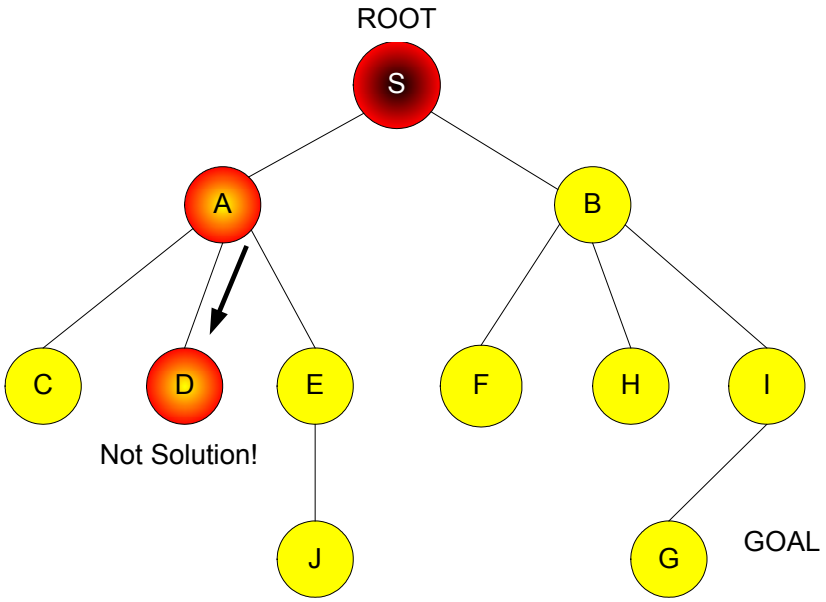


Figure 7 VISIT-LEFT-MOST-FIRST-HEURISTICS - No Solution

You may notice that by using this "VISIT-LEFT-MOST-NODE-FIRST-HEURISTIC", we may need more than ten steps until we can find the solution "G". We would like to use another heuristics to guide the search and see whether there is any difference between these two performances.

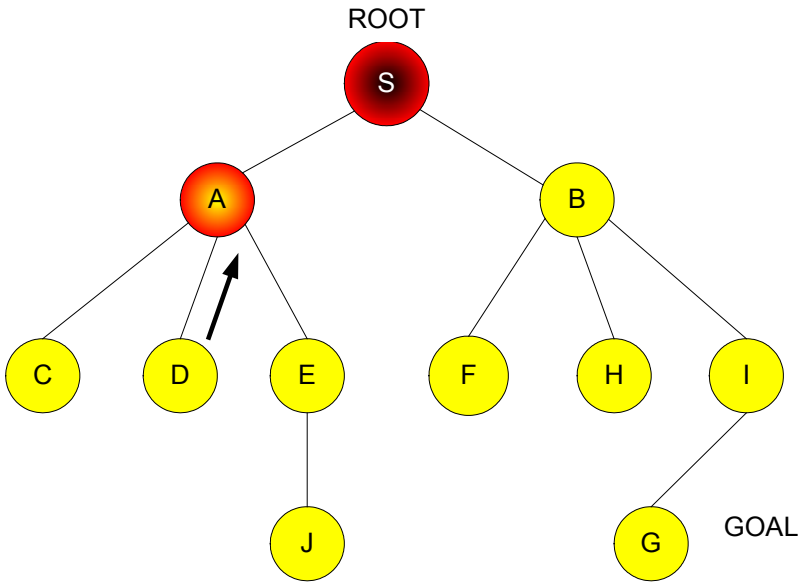


Figure 8 VISIT-LEFT-MOST-FIRST-HEURISTICS - Propagate Again

We try to use the "*VISIT-RIGHT-MOST-NODE-FIRST-HEURISTICS*" instead of the previous one and notice the performance increased. First, by the rules of thumb, it will propagate to node "B".

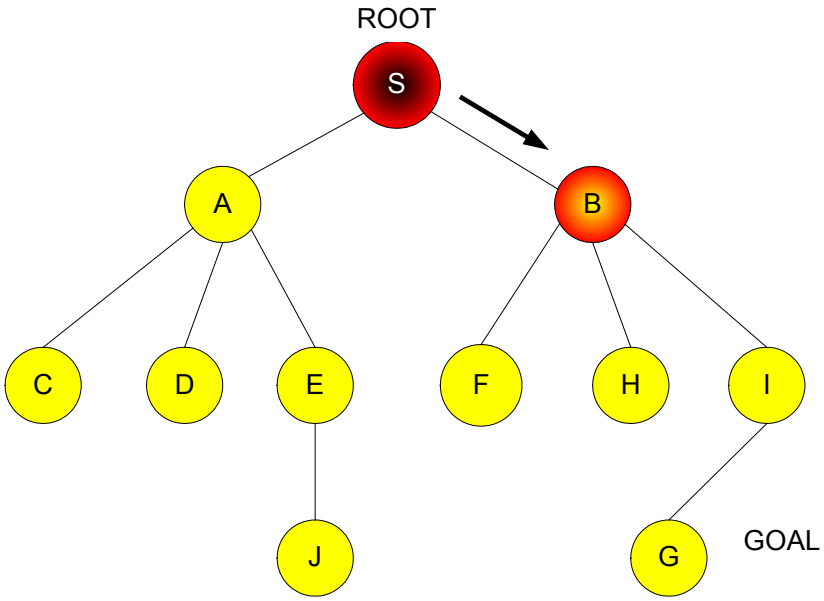


Figure 9 VISIT-RIGHT-MOST-FIRST-HEURISTICS - "B"

Followed by Node "I"

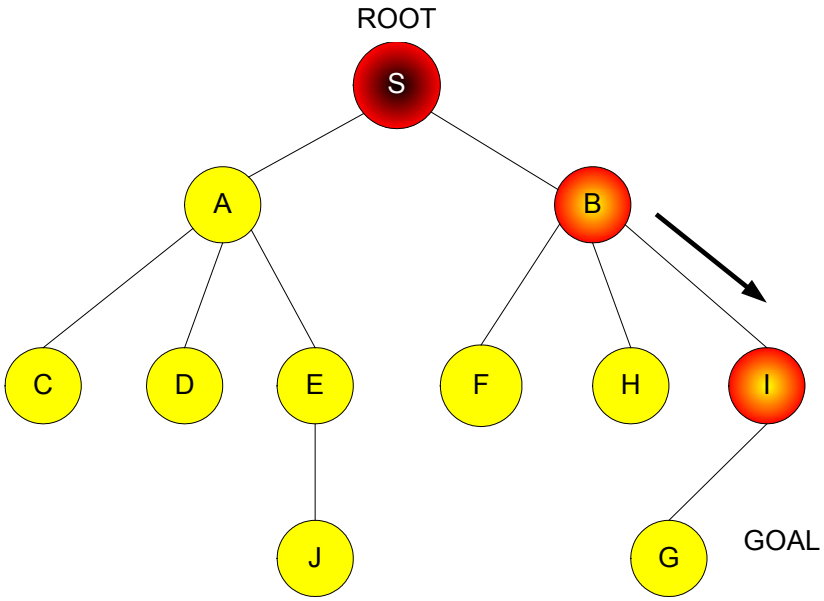


Figure 10 VISIT-RIGHT-MOST-FIRST-HEURISTICS -"I"

With just three single steps, the solution Node "G" can be found here. In a very simple and straightforward model like this, the improvement between two different heuristics may be just more than 10 steps. However, in our single-step constraint-based model, the number of choice points that we have to consider is more than thousands and the number of constraints is more than 30 thousands in some cases. It will be greatly different in terms of search time and constraint propagation. That drives me to utilize the heuristics technique to our model to improve the performance.

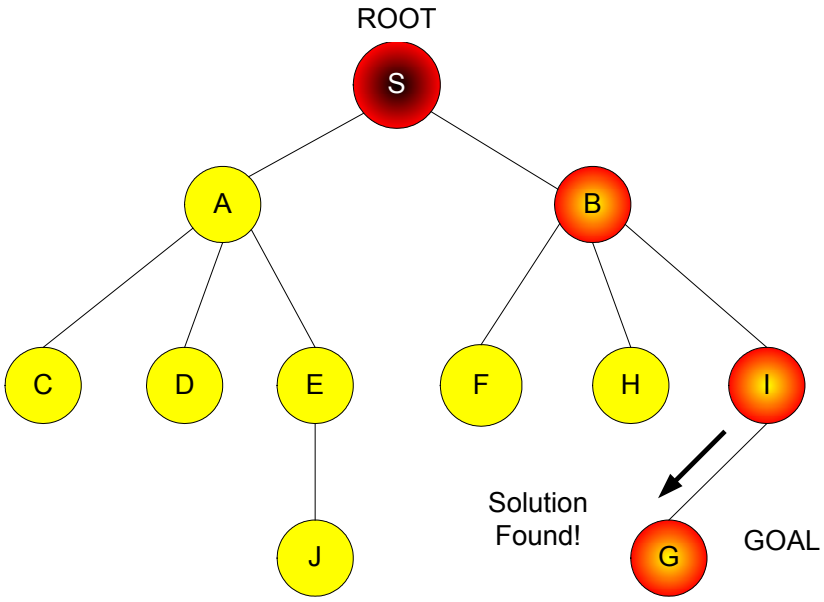


Figure 11 VISIT-RIGHT-MOST-FIRST-HEURISTICS -Solution Found

VIII. Methodology

1 OUR SINGLE-STEP MODEL

Previous approaches use 3 separate steps since a single-step model was too computationally intensive. Obviously, not all constraint-based problems can be sub-dividable into smaller problems. With the objective of finding general solutions to this type of timetabling problems, we instead use a single-step constraint-based model to test the limits of this approach. Our problem is a generalization of the problem solved in Nemhauser and Trick [9] and Henz [5], which was specific to only the 1997/1998 ACC problem with $n=9$.

Our single-step model on the other hand works for any value of n. The model itself is used mainly as a test bed to investigate the effects of search heuristics on performance in timetabling problems. McAloon [8] also used a similar single-step model. They remarked that their code's performance was very delicate and at best can find a solution to $n=14$ within 45 minutes on an UltraSparc. Our best heuristics was able to find $n=18$ solutions with an average of 2.3 seconds each on a PC.

The CSP search algorithm finds a solution by assigning a value to each variable. The value is taken from the variable's current domain, which gets reduced through constraint propagation as the search tree gets expanded. This d*omain reduction* effect helps "narrow" down the search (trim down the search tree) and hence reduces search time. Therefore, the "shape" of the search tree as it gets expanded has a major impact on the performance of any CSP algorithm. If the search tree is not expanded correctly, the CSP search might not be any better than simple exhaustive search.

2 Comparison Between Different Approaches

Major Comparison with Other Approaches

	Number of Steps	Maximum Number of Teams	Time Spent	Methodology
Nemhauser & Trick	3	Specific to 8	About 24 Hours	Integer Programming with Enumeration
Henz	3	Specific to 8	About 1 Minute	Constraint Programming
Our Approach	*1*	*18 Teams*	*Less Than 20 Seconds*	*Constraint Programming*

Table 1 Major Comparison with Different Approaches

Other researchers generally use Genetic algorithm and Tabu Search to solve this problems.

3 Object Design

A Match class represents the match of tournament
An Application class coordinates the scheduling and provides the UI for user to choose different chainable heuristics.

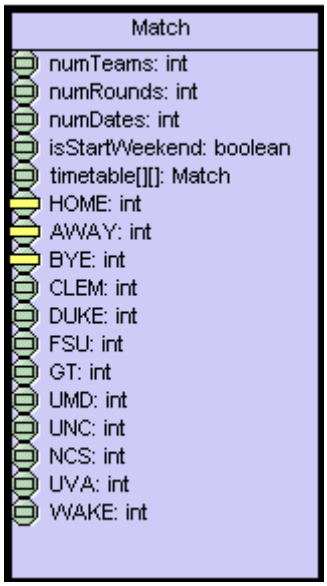


Figure 12 Class Diagram of the class Match

The Match class represents the match of the tournament. We store each match in the array *timetabl[][][]* which acts as the tournament timetable. This matrix contains all the matches in a season e.g.: *timetable*[*thisTeam*][*date*]. For example, *timetable*[2][3] represents the match of FSU (2) on date 3 and timetable[4][2] represents the match of UMD (4) on date 2. All the match instances are represented using this matrix and created by the *Application* class. In addition, we have simply enumerated the teams in the conference with numbers from 0 to 8

```
// parameters of the ACC tournament ----------------------------------
static int numTeams; // The total number of teams in the tournament
static int numRounds; //The total number of rounds (either 1 or 2)
static int numDates; // The computed total number of dates in the tournament

/**
 * True if the first date is a weekend. This program assumes that there are
 * only 2 games per week - one on a weekend and one on a weekday.
 */
static boolean isStartWeekend;
static Match timetable [][]; // The tournament timetable

/**
 * Global variables
 */
public final static int HOME = 0;
public final static int AWAY = 1;
public final static int BYE = 2;
public static int BYETEAMID;        // the value of otherTeam if this is a bye

private final static int CLEM = 0;
private final static int DUKE = 1;
private final static int FSU = 2;
private final static int GT = 3;
private final static int UMD = 4;
private final static int UNC = 5;
private final static int NCS = 6;
private final static int UVA = 7;
private final static int WAKE = 8;
```

The *Application* class is used to coordinate the scheduling task and provide an UI for user to choose the chainable heuristics. It generally make the time table by passing the parameters to the Match class and constructing the matrix of tournament timetable *timetable*[][]. By calling the *postAllConstraints*() method of the super class *Match*, the constraints here in the tournament model will be posted to our scheduler.

```
public class Application extends Match{
//….
  final static void runTest(int numTeams) throws FailException {
    boolean isStartWeekend = false; // second day is weekend
    System.gc();

//Create a multidimensional matrix of Match objects that represents the timetable
    makeTimetable(numTeams, numRounds, isStartWeekend);

//Post all the constraints in this problem
    postAllConstraints();

    // the two vectors to be solved
    VarVector otherTeamVars = getAllOtherTeamVars();
    VarVector homeAwayVars = getAllHomeAwayVars();

    // put all variables into one vector
    VarVector allVars = solver.varVector();
    allVars.append(homeAwayVars);
    allVars.append(otherTeamVars);

    // find time for first 10 solutions
    solver.activate(solver.generate(allVars, choose, select));
```


4 Extending the OO Design with CP

Up to now, the OO design and Java implementation do not contain any constraint programming features. The next step is to augment the design with constrained variables.

Instead of representing all the unknowns into constrained variables, such as the date of the match, the home team of the match, the away team of the match, we implement the model with just two constrained variables for each Match instance, the location of each match and the *otherTeam* of each match. We simply increased the performance of the model by decreasing the number of constrained variables. And it is a typical kind of domain reduction which we will explain later.

The location will be represented as the constrained variable *homeAway* and the other team that this team is playing against will be represented by the constrained variable *otherTeam*.

```
public class Match {
//..
  protected Match(final int team, final int date) {
    this.thisTeam = team;
    this.date = date;
    homeAway.setObject(this);
    otherTeam.setObject(this);

  /**
   * This constrained variable represents whether the match is played at
   * "home" (0), "away" (1), or "bye" (2).
   */
  final Var homeAway = solver.var(0, 2, "home/away/bye");

  /**
   * This constrained variable represents the other team that this team is
   * playing against. Team id start from 0, but will be incremented by 1
   * on printout. If otherTeam=numTeams, then this is a "bye".
   */
  final Var otherTeam = solver.var(0, numTeams, "other team");
//..
}
```

Using just two variables *homeAway* and *otherTeam* along with the *timetable*[][] matrix instances, the date of the match is no longer a constrained variable of the model. The posting of constraint cannot become easier. For example, we try to post a constraint that team CLEM 0 is not playing team DUKE 1 on date 4, we can use the following with the *Var.neq*():

```
solver.post(timetable[0][4].otherTeam.neq (1) );
```

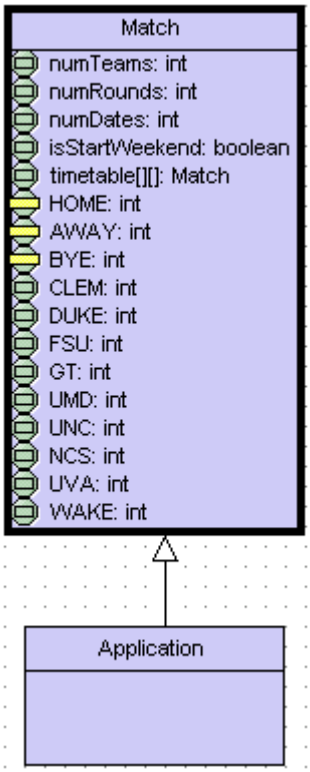


Figure 13 Class Diagram of the Model

5 Problem Representation: Designing the Model

5.1 Constrained Variables in the Model

Variables and Domain

What are the unknowns – the variables?

The unknowns in the scheduling model are:

- The location/stadium of where a team will play on a specific date (*homeAway*)
- The opponent team (*otherTeam*) of a team on a specific date

What are the potential values of these unknowns – the domains?

- The domain of location is either a home, away, or a bye (*homeAway*) represented using integers [0,1,2] respectively

- The domain of *otherTeam* is just all the teams in the conference implemented as [0, … 8]

We use a single-step constrained-based model with 2 constrained variables – $loc_{i,j}$ and $team_{i,j}$ of each match $M_{i,j}$ of row i and column j:

- $loc_{i,j}$ is a constrained variable representing the location of the match for team i on date j with a domain of 3 possible values - [home, away, bye]

- $team_{i,j}$ is a constrained variable representing the opponent team that team i will play against on date j with domain consisting of all the possible teams except i. If $loc_{i,j}$ = bye, $team_{i,j}$ will be assigned a dummy value as there will be no opponent.

5.2 Constraints in the Model

The basic constraints in our model are (borrowed from Nembauser [1]):

- At least one bye per round robin if odd number of teams
- Number of home/away per team is equal to total number of dates/2
- Number of home/away per day is equal to total number of teams/2
- Ensure round robin - each team plays against each other team
- No team can play away on both last dates
- No team may have more than two away matches in a row
- No team may have more than two home matches in a row
- No team may have more than three away matches or byes in a row
- No team may have more than four home matches or byes in a row

5.3 Three Types of Constraints

Within an object-oriented model, we can classify the constraints into 3 types: Inherent, intrinsic, or relational. Inherent constraints are usually posted in the constructor of each class. They apply to all objects within the class. Intrinsic constraints are posted just after object is created. They only apply to a specific object. Relational constraints are posted after all objects have been created and before solving or during the search using value-action demons. Value-action demons are programs that get "*trigger*" whenever a constraint variable is assigned a value. Actions will be undone if the constraints are not satisfied and backtracking occurs. They apply among two or more objects.

Here we classify some of our constraints in the model into these three types:

Intrinsic Constraints

Constraints that are intrinsic to an object
[C1] Each team plays against each other team

Relational Constraints

Constraints that are relational among objects
[C2] At least one bye per round robin if odd number of teams
[C3] Ensure if team A has a bye, its opponent team also has a bye in that match
[C4] No team can play away on both last dates

Constraint [C1] can be implemented using JSolver as shown below. A *FailException* will be thrown in case the constraint posting causes some variables to have a null domain. The object *timetable[i][j]* represents the match of team *i* on date *j*, the *otherTeam* variable is the opponent team this team will play against. The *vars.cardEq*(*k, numRounds*) method creates a cardinality constraint that restricts the total number of matches this team will play against another team *i*. The total number must be equal to "*numRounds*".

```
private static void postRoundRobin() throws FailException {
  // (a) cannot play against itself
  for (int i = 0 ; i<numTeams; i++) {
    for (int j = 0 ; j<numDates; j++) {
      solver.post(timetable[i][j].otherTeam.neq(i));
    }
  }
  // (b) play against another team once (single) or twice (double) only
  // Horizontal constraint
  for (int i = 0 ; i<numTeams; i++) {
    VarVector vars = getOtherTeamVars(i);
    for (int k = 0 ; k<numTeams; k++) {
      if (i!=k) {
        solver.post(vars.cardEq(k, numRounds));
      }
    }
        //…
  }
}
```

Constraint [C2] is posted after all objects have been created and before the search. It can be implemented as shown below. Again, we use to *cardEq*(*int, int*) method to define cardinality constraints. It is posted so that if team *i* has a bye on date *j*, its opponent team *otherTeam* will also has a *bye* on date *j*.

```
private static void postByes() throws FailException {
  final int numByes = (numTeams%2==0) ? 0 : numRounds;

  for (int i = 0 ; i<numTeams; i++) {
    solver.post(teamHomeAwayVars[i].cardEq(BYE, numByes));
    solver.post(teamOtherTeamVars[i].cardEq(BYETEAMID, numByes));

    // other team should not be assigned if bye
    /* move to demon
    for (int j = 0 ; j<numDates; j++) {
      solver.post(timetable[i][j].homeAway.eq(BYE).
                  ifThen(timetable[i][j].otherTeam.eq(BYETEAMID)));
      solver.post(timetable[i][j].otherTeam.eq(BYETEAMID).
                  ifThen(timetable[i][j].homeAway.eq(BYE)));
    }
          //……
  }
}
```

Constraint [C3] is implemented as a value-action demon. Value-action demons are usually created in the constructor. We subclassed the JSolver provided Demon class and overrode the *Demon.executeDemon*() abstract method to represent constraint [C3].It will throws an *FailException* if the constraint has been violated using the *solver.fail*() method. This will trigger the JSolver backtracking mechanism to backtrack to a previous choice-point and continue the non-deterministic search from that point onwards.

```
  homeAway.addValueAction( new Demon(solver) {
    public void executeDemon() throws FailException {
      //System.out.println("assign homeAway: "+name+"="+homeAway.getObject());
      if (homeAway.getValue()==BYE) ensureBye();
      else ensureNotBye();
    }
  });
private final void ensureBye() throws FailException {
  if (homeAway.isBound()) {
        //if the team has a bye and the value is bound
    if (homeAway.getValue()==BYE) {
      if (!otherTeam.isBound()) {
```

```
          solver.post(otherTeam.eq(BYETEAMID));
        }
            //but its opponent does not have a bye
        else if (otherTeam.getValue()!=BYETEAMID) {
          if (debugging) {
            System.out.println("Bye but otherTeam is not BYETEAMID");
          }
            //force backtracking
          solver.fail();
        }
//….
```

In constraint [C4], we make use of the *cardLt*(*int, int*) to prevent both the last dates matches are away matches. This method creates a cardinality constraint that restricts the total number of variables within this vector that are equal to the given "value". All we have to do is to create a *VarVector* which contains the variables of both of the last dates *homeAway* variables. Here is the code:


```
  /**
   * Constraint [C4]: No Two Final Aways.
   * No team can play away on both last dates.
   */
  private static void postNoTwoFinalAways() throws FailException {
    for (int i = 0 ; i<numTeams; i++) {
//…
      final Var HAVar1 = timetable[i][numDates-2].homeAway;
      final Var HAVar2 = timetable[i][numDates-1].homeAway;
      final VarVector vars = solver.varVector(HAVar1, HAVar2);

      // cannot be away on "both" the last 2 dates
      solver.post(vars.cardLt(AWAY, 2));
    }
  }
```

6 CSP Design

Each instance of the Match object class represents a match in the sports scheduling system. Each Match object contains two constrained variables: *homeAway* and *otherTeam* which refers to the home/away/bye and opponent team variables respectively. And Each Match object also contains static information which includes:

- Team ID: Clemson (abbreviation Clem; team0), Duke(Duke; 1), Florida State (FSU; 2), Georgia Tech (GT; 3), Maryland (UMD; 4), North Carolina (UNC; 5), North Carolina State (NCSt; 6), Virginia (UVA; 7), and Wake Forest (Wake; 8).

- Date, domain between 0 and 8; And Team Name (a string)

```
//Class Match
protected Match(final int team, final int date) {
    this.thisTeam = team;
    this.date = date;
    homeAway.setObject(this);
    otherTeam.setObject(this);
    //…

    static Match timetable[][];   //the tournament timetable

    final int thisTeam;           //instance variables
    final int date;
    final String name;

    //Two constrained variables in each Match object
    final Var homeAway = solver.var (0,2, "home/away/bye" );
    final Var otherTeam   = solver.var(0, numTeams, "other Team" );
    //…
}
```

7 Heuristics Models

CSP search heuristics generally consists of two types: *Choose-Variable Heuristics* and *Select-Value Heuristics*. The former one determines the order of selection of constrained variables for instantiation and the later one determines the sequence of selection of values from domain to instantiate the variable value.

Choose-Variable Heuristics can greatly affect the branching factor and hence also the size of the resulting search space. Select-Value Heuristics affect the ordering of branches in the search tree. This is important when users are interested in obtaining the first solution of a CSP fast using some form of depth-first search strategy. Therefore, various choose-variable and select-value heuristics, such as the *ChooseMinSize* principle, have been devised to speed up solving of particular CSP instances. These heuristics are, however, usually problem and domain specific.

We aims to compare the effects of using different types of heuristics on the performance of the TDSRR scheduling problem. Here we further decompose the choose-variable heuristics into four different branches:

- **Variable-based Heuristics** – these kinds of heuristics choose constrained variables based purely on the location, domain size or value of the variables.

Feature	Heuristic	Description
order	ChooseFirstUnboundHeuristic ChooseLastUnboundHeuristic	First unbound var first Last unbound var first
size	ChooseMinSize ChooseMaxSize	var with min domain size first var with max domain size first
value	ChooseMinValue ChooseMaxValue	var with min value in domain first var with max value in domain first

- **Constraint-based Heuristics** – these kinds of heuristics choose constrained variables based on the association of constraints and variables.

Feature	Heuristic	Description
cnst	ChooseMinConstraint ChooseMaxConstraint	var with min number of constraints first var with max number of constraints first
pcnst	ChooseMinPropConstraint ChooseMaxPropConstraint	var with min number of propagatable constraints first var with max number of propagatable constraints first
avar	ChooseMinAssocVars ChooseMaxAssocVars	var with min number of associated constrained variables first var with max number of associated constrained variables first
uavar	ChooseMinUnboundAssocVars ChooseMaxUnboundAssocVars	var with min number of unbound associated constrained variables first var with max number of unbound associated constrained variables first
bavar	ChooseMinBoundAssocVars ChooseMaxBoundAssocVars	var with min number of bound associated constrained variables first var with min number of bound associated constrained variables first

- **Structured-based Heuristics** – these kinds of heuristics choose constrained variables based on structure of the timetable.

Feature	Heuristic	Description
row	ChooseMinRow ChooseMaxRow	var with min row index first var with max row index first
col	ChooseMinCol ChooseMaxCol	var with min column index first var with max column index first
xrow	ChooseRowExtreme ChooseRowCenter	var in both extremes first var in middle rows first
xcol	ChooseColExtreme ChooseColCenter	var in both column extremes first var in middle columns first
uni	ChooseUniform	vars uniformly spread out

- **Model-based Heuristics** – these kinds of heuristics choose constrained variables based on aspects of the model.

Feature	Heuristic	Description
match	ChooseMatch	assign one match at a time
team	ChooseOtherTeam	assign team variables first
loc	ChooseHomeAway	assign loc (home/away/bye) variables first

Similarly, we divide the select-value heuristics into two branches:

- **Local Heuristics** – these kinds of heuristics choose values based on value in the domain.

Feature	Heuristic	Description
value	SelectMinHeuristic SelectMaxHeuristic	min value in domain first max value in domain first

- **Global Heuristics** – these kinds of heuristics choose values based on the frequency of use.

Feature	Heuristic	Description
freq	SelectLeastUsed	least frequent value used in all bound vars first

IX. Application Programme

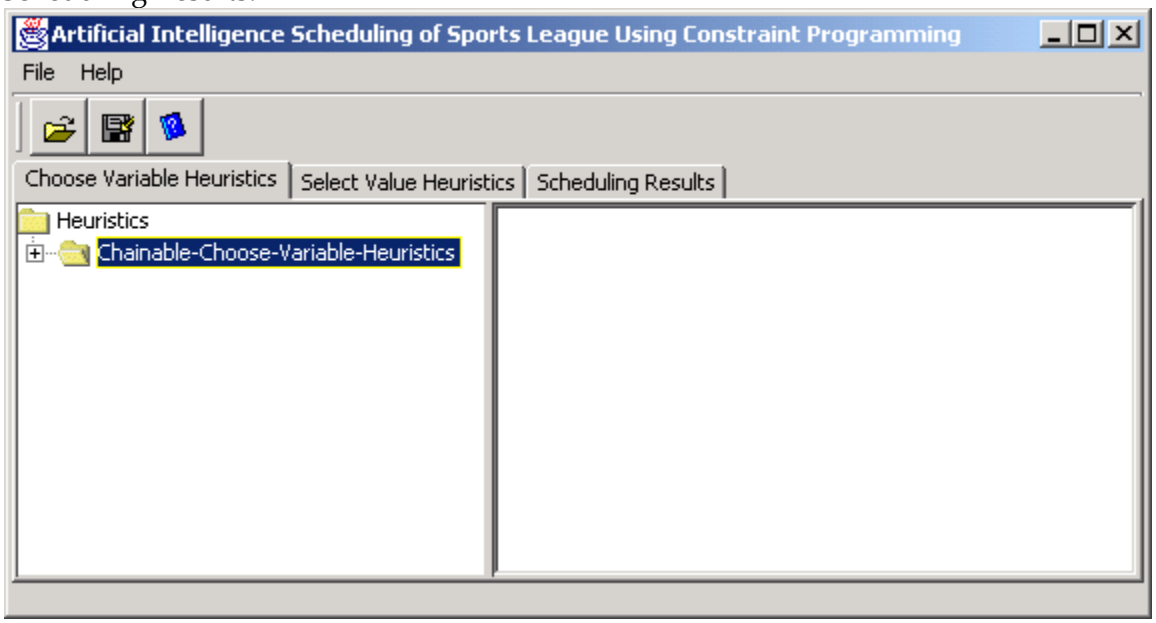
A Java-based application programme has been developed to act as a test bed for illustrating the performance of different chainable heuristics. And we have implemented a Java Swing UI instead of a console application to outline the results generated. Users are freely to choose different combination of heuristics to guide the CSP search although some of them fail to generate the solution within 3 minutes (Timeout).

As we have stated before, the heuristics can generally divided into two main types: the Choose-Variable Heuristics and the Select-Value Heuristics. The Java Swing user interface is designed so that the user can choose different chainable Choose-Variable heuristics and the Select-Value heuristics. We prefer to use chainable heuristics instead of non-chainable one because it provides a larger combination of heuristics and hence a more accurate test bed for we to investigate both the schedule and performance.

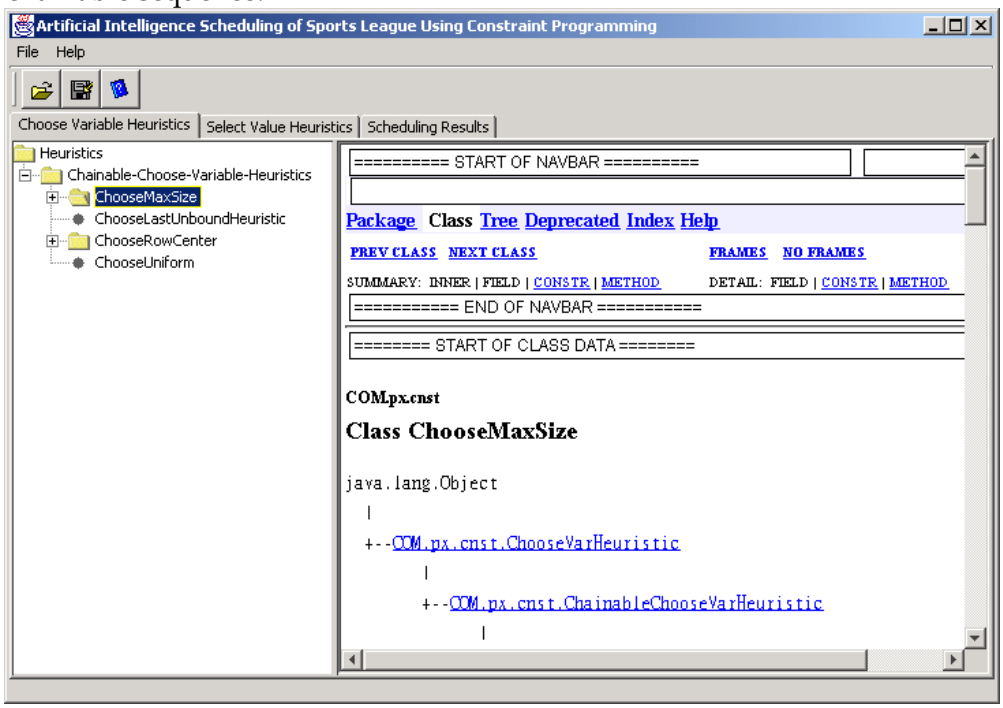
In fact, heuristics functions that chained together behaved like "and". For example, chaining a maximum constraint heuristic and a minimum domain heuristic will result in choosing the variable the maximum number of constraints and minimum domain size. The minimum domain heuristic will be used as a secondary variable selection heuristic.

Here let 's show the procedure of using our *Scheduler:*

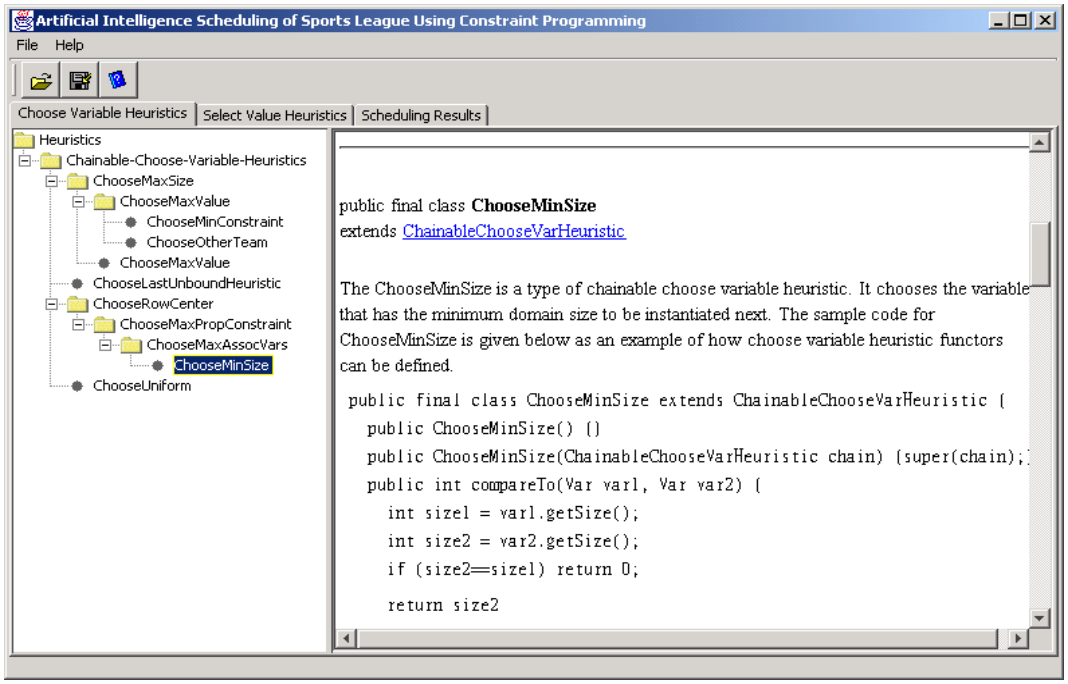
Here you can have the first glance of our swing UI of the application which mainly consists of three tags: Choose Variable Heuristics, Select Value Heuristics and the Scheduling Results.



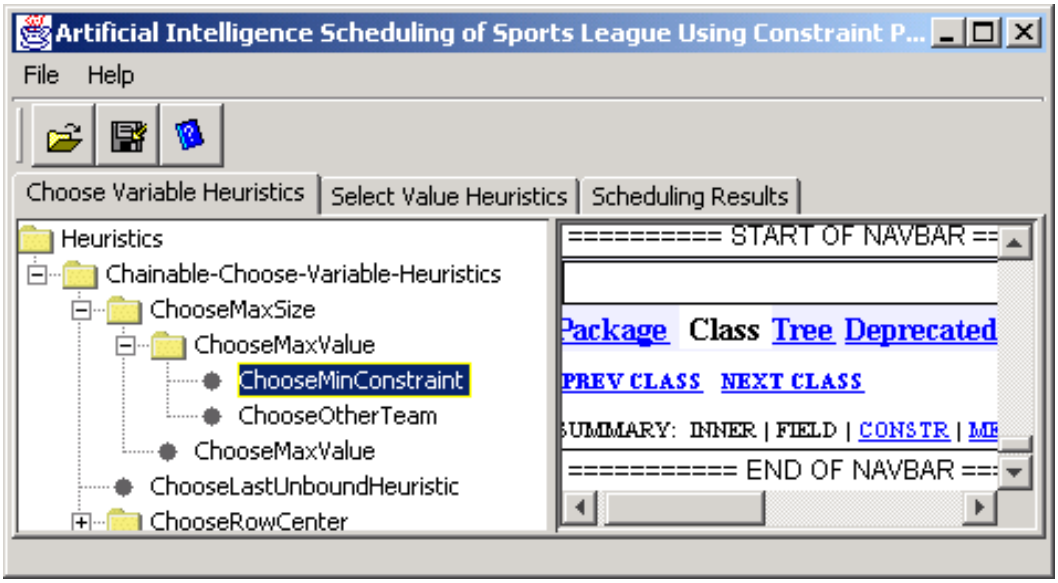
On the left part of the frame, user can choose the chainable form of heuristics. The heuristics are formatted in a tree structure so we can choose them one by one according to the chainable sequence.



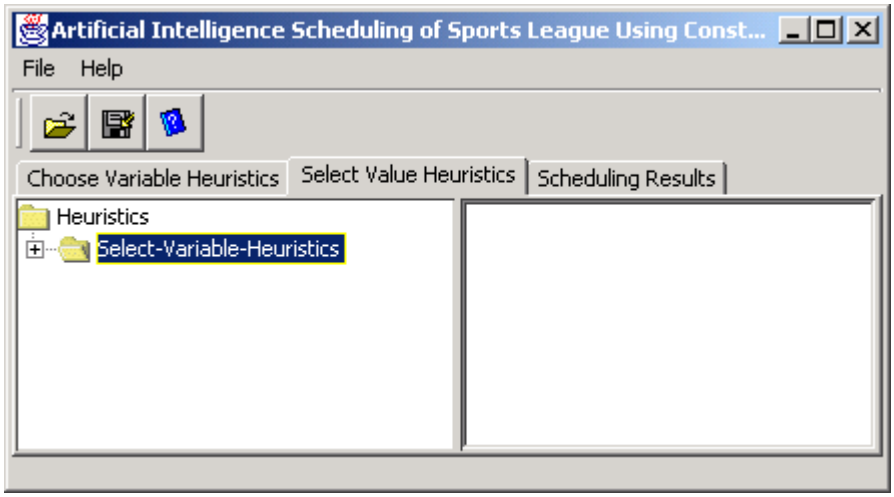
On the other hand, the right part of the frame is for documentation of the heuristics. After clicking a suitable heuristics, the documentation will be shown on the right side of the frame for user to reference.



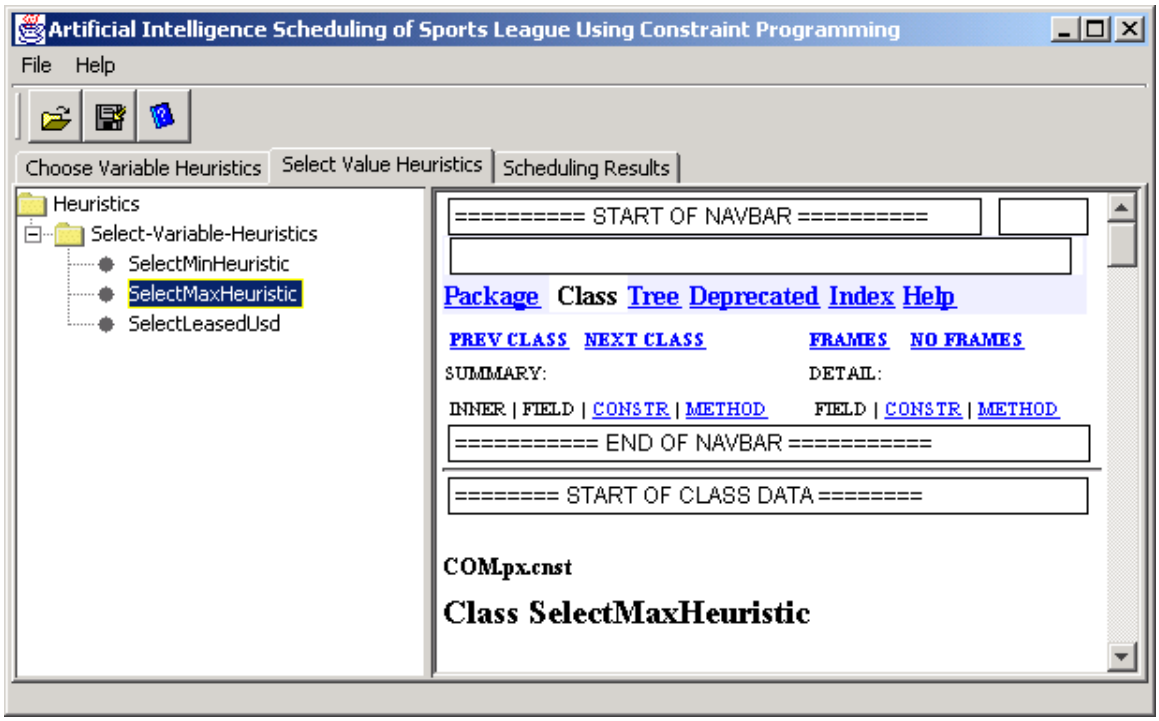
For example, we want to choose the ChooseMaxSize-ChooseMaxValue-ChooseMinConstraint of the Choose-Variable Heuristics. We choose them by clicking each heuristic by that sequence until we reach the last heuristic – ChooseMinConstraint.



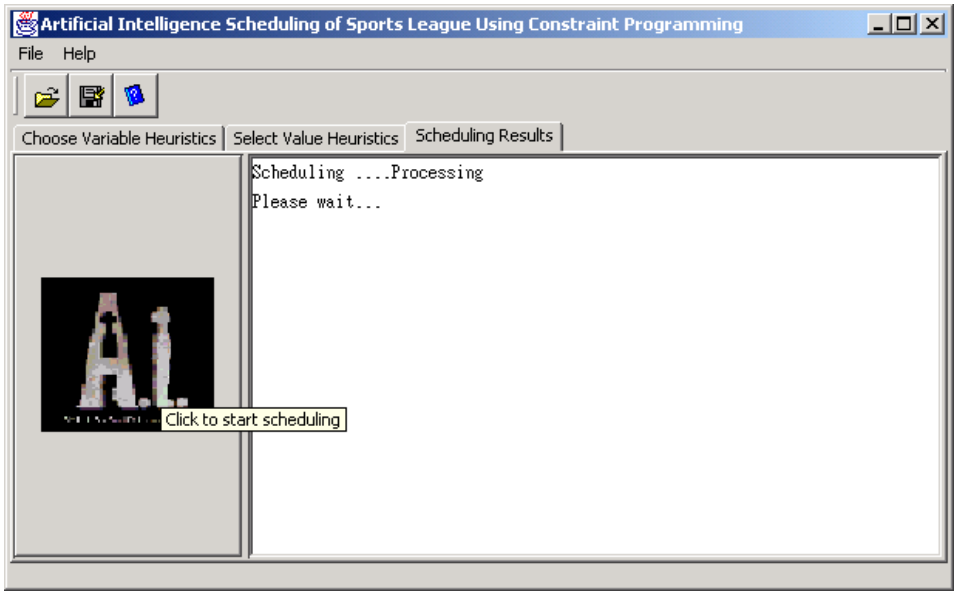
Until now, the Choose-Variable Heuristics part have been chosen. We have to select the Select-Value Heuristics by first clicking the tag on top of the frame "Select Value Heuristics".



This is similar to the way of the Choose-Variable Heuristics tag. We select the "SelectMaxHeuristic" in this case.



The next step is to use the combination of the user input to generate the schedule guided by these heuristics.

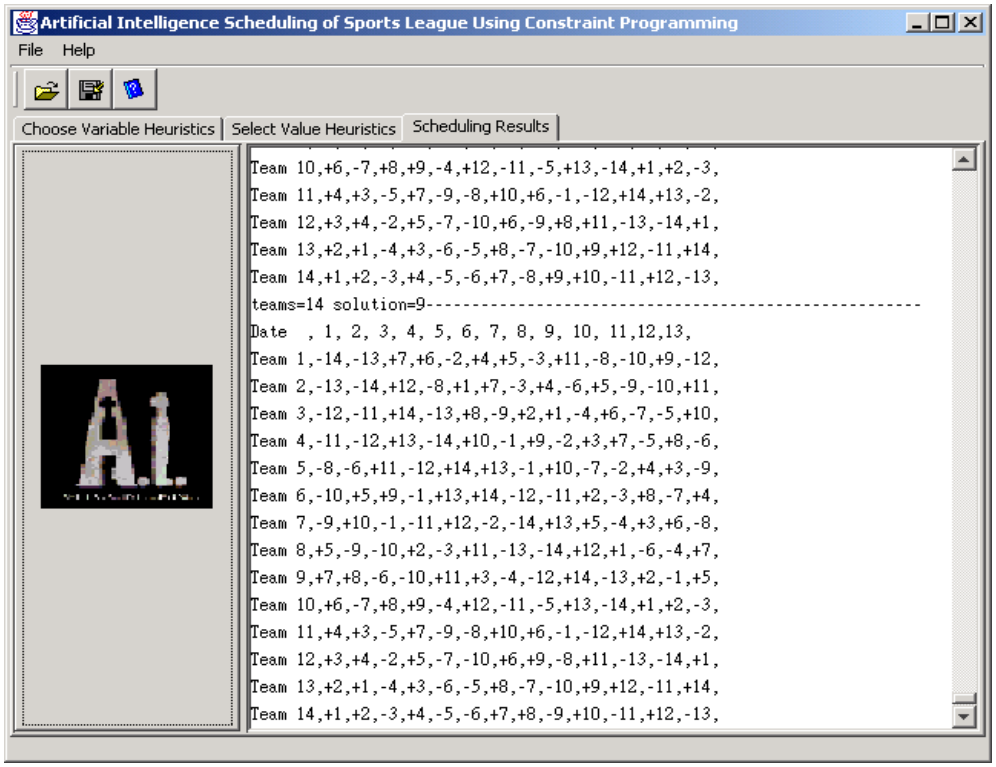


The real schedule of a temporally dense single round robin tournament is generated. The following is a sample tournament schedule for n=9 teams, r=1 (single round robin), and d= (r n) = 9 dates.

The notation used in here is similar to that of Henz [3] and Schreuder [22]. Each matrix row contains all the matches that a particular team will play at different dates. The number in the matrix is the opponent team's ID. "*b*" represents a *bye*, "+" is a *home* match, and "-" is an *away* match. Each column, on the other hand, contains all the matches that are to be held in one particular date. In order words, row *i* and column *j* shows the match that team *i* will play in date *j*. Numbers in each column are unique as each team only plays one match per date. Numbers in each row are also unique as each team plays against team only once in a single round robin.

Date	1	2	3	4	5	6	7	8	9
Team1	+6	+4	-2	-7	+5	-3	b	+9	-8
Team2	-4	b	+1	-3	+7	+9	-8	+5	-6
Team3	b	-6	-5	+2	-9	+1	-7	+8	+4
Team4	+2	-1	-8	+5	b	+7	-9	+6	-3
Team5	-7	+9	+3	-4	-1	+8	+6	-2	b
Team6	-1	+3	+7	-9	+8	b	-5	-4	+2
Team7	+5	+8	-6	+1	-2	-4	+3	b	-9
Team8	+9	-7	+4	b	-6	-5	+2	-3	+1
Team9	-8	-5	b	+6	+3	-2	+4	-1	+7

Table 2 One Solution to TDSRR with 9 teams



Here is a snapshot of the statistics of JSolver execution which is output by the application itself. It includes the number of choice points, the number of variables, the number of constraints and the elapsed time of the application. The statistics will be used as a reference for the performance.

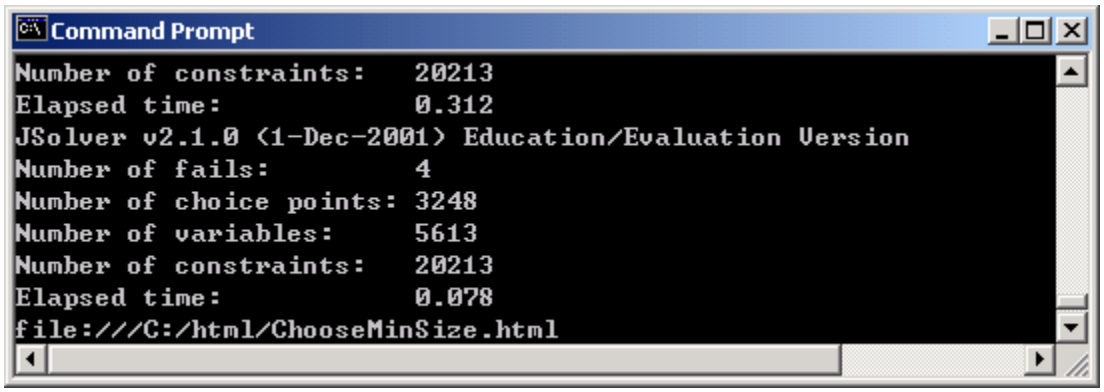


Figure 14 Statistics of One Execution

X. Benchmarking Results

We implemented our single-step CSP model using a Java constraint-programming class library and ran the experiments on a Pentium III 750 MHz PC. The constrained variables of the TDSRR problem were created and stored in a list. The initial order of the list consists of all the *loc* variables followed by all the *team* variables. The variables are stored in row-major order. For a problem with n teams and d dates, there are $2(nd)$ variables total in the list. We then search for a solution to the TSDRR problem by assigning a value to each variable. Variables and values were selected in the order determined dynamically by our heuristics chain.

We tested these search heuristics by running experiments with $n=4$ and upwards. The first 10 solutions for each value of n were retrieved. Each run has a time limit of 3 minutes, after which it will time out and continue with the next experiment. The following Chart 1 shows some of our results.

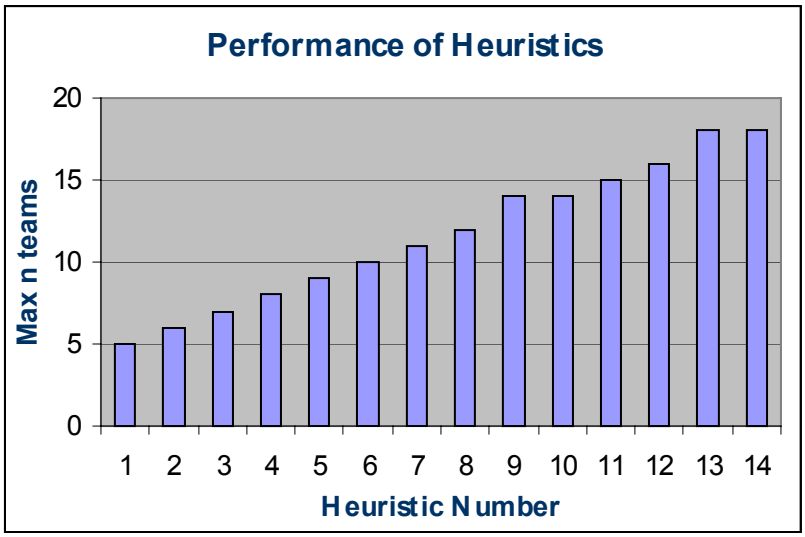


Chart 1. The maximum number of teams solved by different heuristics

The respective heuristics were:

Heuristics	Variable	Value	Max n teams
1	ChooseMinSize-ChooseMaxValue	SelectMaxHeuristic	5
2	ChooseUniformTeamFirst	SelectMinHeuristic	6
3	ChooseMinConstraint-ChooseMinSize	SelectMinHeuristic	7
4	ChooseExtremeRow	SelectMinHeuristic	8
5	ChooseMinRow-ChooseMaxSize-ChooseMaxValue	SelectMaxHeuristic	9
6	ChooseUniform	SelectLeasedUsed	10
7	ChooseMinRow-ChooseMaxSize-ChooseMaxValue-ChooseMinCnst	SelectMaxHeuristic	11
8	ChooseMaxSize-ChooseMaxValue	SelectMinHeuristic	12
9	ChooseMaxSize-ChooseMaxValue-ChooseMinCnst	SelectMaxHeuristic	14
10	ChooseMaxSize-ChooseMinValue	SelectMaxHeuristic	14
11	ChooseLastUnbound	SelectMinHeuristic	15
12	ChooseHomeAway-ChooseMaxSize-ChooseMaxValue	SelectMaxHeuristic	16
13	ChooseMaxSize-ChooseMaxValue-ChooseOtherTeam	SelectMaxHeuristic	18
14	ChooseMaxSize-ChooseMaxValue	SelectMaxHeuristic	18

It can be shown that the heuristics 13 and 14 have higher performances around these search heuristics chain. It is originally expected that the more

complex constraint-based heuristics would perform better, such as the heuristics chain 7. Instead, a more generic *ChooseMaxSize-ChooseMaxValue* combined with *SelectMaxHeuristic* introduced a better performance, solving TDSRR tournaments with n =18 teams with the first solution in 18.7 seconds and then an average of 0.5 seconds for subsequent solutions. This chainable heuristics choose constrained variables which are less constrained. This leaves highly constrained variables to be pushed to the bottom of the search tree, probably allowing more efficient backtracking and hence better propagation for this ACC problem.

The following (Table 3) is a sample tournament schedule for n = 9 teams, r = 1 (single round robin), and d = (r n) = 9 dates. The number in the matrix is the opponent team's ID. "b" represents a *bye*, "+" is a *home* match, and "-" is an *away* match. Each column represents the matches played on that date.

Date	1	2	3	4	5	6	7	8	9
Team1	+6	+4	-2	-7	+5	-3	b	+9	-8
Team2	-4	b	+1	-3	+7	+9	-8	+5	-6
Team3	b	-6	-5	+2	-9	+1	-7	+8	+4
Team4	+2	-1	-8	+5	b	+7	-9	+6	-3
Team5	-7	+9	+3	-4	-1	+8	+6	-2	b
Team6	-1	+3	+7	-9	+8	b	-5	-4	+2
Team7	+5	+8	-6	+1	-2	-4	+3	b	-9
Team8	+9	-7	+4	b	-6	-5	+2	-3	+1
Team9	-8	-5	b	+6	+3	-2	+4	-1	+7

Table 3 One solution to TDSRR with 9 teams and r=1 (single round robin)

XI. Discussion

1 Problems Encountered

As we design an object model for this problem, we consider the classes, the attributes, and the relations to declare as well as the constraints to define on them. At the *early stage of the design*, we came up with three variables to implement the model in CSP. The problem is also originally modeled as a single-step model; the timetable is modeled with three constrained variables. The constrained variables are:

- $date_j$, with domain [0, numberDates]
- $homeTeam_i$, with domain [0, numberTeams]
- $awayTeam_i$, with domain [0, numberTeams]

In which the $date_j$, $homeTeam_i$, and $awayTeam_i$ represent the date, the home team and the away team of the match respectively.

We encounter difficulties in the *early stage* of the development process. First, the search space is very large which affects the efficiency of the scheduling system. The model fails to return the solution within a reasonable time. In this approach, the model only works for a number of combinations of constraints, the search space increases combinatorially with the number of constraints. Solution generation efficiency is degraded and exhaustive search is resulted.

2 Solutions to the Problem

We cope with these difficulties by firstly reducing the number of constrained variables in the model. It is based on an observation that the $date_j$ variable is already pre-set in this model. In an object model, $date_j$ should be implemented as an attribute instead. Reducing the number of constrained variables in a model means a reduction of search space in the domain and hence faster propagation. We instead use a single-step constrained-based model with 2 constrained variables – $loc_{i,j}$ and $team_{i,j}$ of each match M*i,j* of row *i* and column *j*:

- $loc_{i,j}$ is a constrained variable representing the location of the match for team *i* on date *j* with a domain of 3 possible values - [home, away, bye]

- $team_{i,j}$ is a constrained variable representing the opponent team that team i will play against on date j with domain consisting of all the possible teams except i. If $loc_{i,j}$ = bye, $team_{i,j}$ will be assigned a dummy value as there will be no opponent.

There are some other approaches to increase the efficiency of constraint solving. The first approach amounts to incorporating variable- and value-ordering heuristics in the labeling procedure of a propagation-based constraint solver. The second approach is to introduce redundant constraints whenever possible. This technique is never formally documented or studied but one can find it used in many places in the literature while surprisingly results could sometimes exhibit. And we have used both of these two approaches to implement and enhance our model in order to have faster propagation and make comparison between different heuristics.

2.1 Redundant Constraints

In our model of ACC problem, redundant constraints are used not to change the solution that JSolver ultimately finds, but to improve performance by improving propagation in certain cases. Here is the part of the program code in our model introducing redundant constraint to reduce symmetry: (Reader may also refer to the appendix for details of redundant constraints)

```
//ensure this team is a bye
private final void ensureBye() throws FailException {
  if (homeAway.isBound()) {
    if (homeAway.getValue()==BYE) {
      if (!otherTeam.isBound()) {
          //ensure this team is a bye
        solver.post(otherTeam.eq(BYETEAMID));
      }
      //…
    }
  }
// redundant constraint (faster propagation)
for (int i = 0 ; i<numTeams; i++) {
  if (i!=thisTeam) {
    //ensure other team will not play this byeteam on that date
    solver.post(timetable[i][date].otherTeam.neq(thisTeam));
    //ensure only one byeteam on that date
    solver.post(timetable[i][date].otherTeam.neq(BYETEAMID));
     //…
```

XII Summary

The contribution of the project is three-fold. We have successfully completed the construction of a Temporally Dense Single Round Robin Tournaments Schedule which is implemented by a single-step constraint based model. Instead of consuming time in generating the schedule, it demonstrates the potential of constraint technology in the computerization of scheduling and resource allocation.

Our system compares well against the other researchers'. Our scheduler is able to generate a schedule maximally with 18 teams compared with only specific to just 9 teams of the Nemhauser and Trick's model [1] and the Henz's model [1]. And the time spent to generate the maximum number of teams is faster than the Nemhauser and Trick's model by about 4 thousands time, and the Henz's model by about twice. Our model is able to generate the schedule with 18 teams within 20 seconds for suitable chainable heuristics. Moreover, we just use a single-step constraint-based model which is highly computational intensive compared with other several sub-steps models.

By using suitable ***Chainable Choose-Variable Heuristics*** and the ***Select-Value Heuristics***, the searching time of solution has been decreased due to more efficient backtracking and effective constraint propagation, and hence a better performance for the TDSRR timetabling problem.

Last but not least, a paper entitled "*Using Heuristics in Constraint-based Sports Tournament Timetabling* ", which was written by Dr. Andy Chun of the Department of Computer Science in the City University of Hong Kong and I, was accepted for presentation in the Sixth World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002) in Orlando, USA. This paper has concluded some of the research and implementation which have been taken throughout the process of this project. We focused on explaining the technique and comparison of performance of using different types of heuristics.

XIII Further Works or Suggestions

1 Project Enhancement

A future direction of development is to come up with different approaches to increase the efficiency of constraint solving. We have already tried two approaches so far, incorporating heuristics chain and introducing redundant constraints. Both of them exhibit surprisingly good results and propagation in the TDSRR model. Our enhancement will then be focused on solving the highly computationally intensive Double Round Robin Tournament with a single-step CSP model.

There are at least three other approaches to increase the performance of CSP. The first approach is to replace propagation-based techniques, which are algorithmic in nature, entirely by the promising stochastic techniques such as the iterative repair method.

The second approach employs cooperation among different constraint solvers. The third approach uses redundant modeling [14] through cooperation among different models for the same problem. For example, a generic job-shop scheduling problem can be formulated as assigning machines to workers or assigning workers to machines. Similarly, a tournament timetabling problem can be seen as allocation of match to timeslots or allocation of timeslots to match.

This approach is actually inspired by the effectiveness of redundant constraints, which are redundant semantically but augment the propagation and pruning of the original constraints of a CSP, in speeding up constraint-propagation. In terms of redundant modeling, the method amounts to implementing more than one model of a given problem and somehow connect the model implementations. Theoretically, any one model implementation suffices to solve the CSP at hand completely. Therefore, the models are redundant with respect to one another, hence an effective way to increase constraint propagation. Readers interested in details of the redundant modeling approach are referred to [14].

Apart from these, there are still numerous approaches which we have not exploit yet. And implementation with other Object-Oriented languages may be also another alternative, such as C#, since the explosion of the .net technologies.

2 Other Areas of Application

It can be shown that many different types of real-life scheduling problems, resource allocation problems can be modeled as a Constraint Satisfaction Problem. The resource allocation problems are to assign resource to satisfy the demand which is a typical of CSP.

Due to the need of logistics & human resources management in Hong Kong, we expect constraint programming is definitely one of those successful paradigms to solve this growing need. It brings the computational efficiency and reliability by implementing artificial intelligence techniques to encode knowledge of business rules, operational constraints, marketing knowledge, product or service knowledge into those areas. In most of the cases, it can replace knowledge normally possess by a human expert, customer assistant or salesperson.

Here are some of the areas which have already been proved to be successful by using CP:

CSP Application Area	Successful Applications
Resource Allocation	Airport Stand Allocation System [15]
3D Graphic	Waltz Filtering & Line Labelling Problem [16]
Crew Scheduling	Crew Scheduling System [17], Nurse Rostering in Hospital [18]
Business & Service Management	Customer Order Scheduling System [19], IT Service Management [20], Job-Shop Scheduling [21]

Table 4 Different Areas of Applications of CSP

XIV Acknowledgements

The author would like to thank Dr. Andy Chun of the Computer Science Department, my supervisor, for his guidance, valuable advice and efforts in helping me understand the real issues in Constraint Programming and in pointing out the direction of the early research study. He also generously provides a Java-Class Library – JSolver as a Constraint Programming tools for me throughout the development process. I would also like to extend my appreciation to my accessor, Mr. Cheng Lee Lung, for his support and helpful comments.

XV References

[1] Nemhauser, G. and M. Trick: 1998, "Scheduling a major college basketball conference," Operations Research 46(1), 1-8.

[2] Henz, Martin, and Jörg Würtz, "Constraint-based Time Tabling - A Case Study," *Applied Artificial Intelligence*, 10:439-453, 1996.

[3] Henz, Martin, "Constraint-based Round Robin Tournament Planning," In the *Proceedings of the 1999 International Conference on Logic Programming,* pp.545-557, 1999.

[4] Henz, Martin, "Scheduling a major college basketball conference – revisited," Operations Research, 49(1), Jan/Feb, 2001.

[5] Henz, Martin, Tobias Müller, Sven Thiel and Marleen van Brandenburg, "Global Constraints for Round Robin Tournament Scheduling," *EJORS Special Issue on Timetabling*, 30 September 2001.

[6] Henz, Martin, Tobias Müller, Sven Thiel and Marleen van Brandenburg, "Benchmark Results for Constraint-based Round Robin Tournament Scheduling," to be published.

[7] Chun, H.W., "Constraint Programming in Java with JSolver," *In Proceedings of the First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*, London, April, 1999

[8] A. Colmerauer. An introduction to Prolog III. Communications of the ACM, pages 69-90, July 1990.

[9] ILOG Solver User's Manual 5.0, August 2000.

[10] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99-118, 1977.

[11] Ken McAloon, Carol Tretkoff, Gerhard Wetzel, "Sports League Scheduling" In *Proceedings of the Third ILOG Optimization Suite International Users' Conference.* Paris, France (1997)

[12] Costa, Daniel, "An evolutionary tabu search algorithm and the NHL scheduling problem," ORWP 92/11, Swiss Federal Institute of Technology, Department of Mathematics, 1992.

[13] J.P. Hamiez and J.K. Hao, " Solving the sports league scheduling problem with Tabu search," Lecture Notes in Artificial Intelligence 2148: 24-36, Springer-Verlag. (Preliminary version presented at ECAI00 Workshop on Local Search for Planning and Scheduling, Berlin, August, 2000).

[14] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. "A constraint-based nurse rostering system using a redundant modeling approach." Technical report (published),

Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1996.

[15] Andy Hon Wai Chun, Steve Ho Chuen Chan, Francis Ming Fai Tsang and Dennis Wai Ming Yeung "HKIA SAS: A Constraint-Based Airport Stand Allocation System Developed with Software Components" 1999 *Innovative Applications of Artificial Intelligence Award Winner*

[16] Andy Hon Wai Chun "Waltz Filtering in Java with JSolver" Published in the *Proceedings of PA Java99 The Practical Applications of Java*, London, April 1999

[17] Benny Shum Kai Shun, Accede Computer Solutions Pte Ltd "Crew Scheduling System" ILOG Optimization Suite International Users Meeting, July 9 1997

[18] Andy Hon Wai Chun, Steve Ho Chuen Chan, Garbbie Pui Shan Lam, Francis Ming Fai Tsang, Jean Wong and Dennis Wai Ming Yeung "Nurse Rostering at the Hospital Authority of Hong Kong" 2000 *Innovative Applications of Artificial Intelligence Award Winner*

[19] Kimberly Fekel, Martha Mulvaney, Fred Garrett, "Customer Order Scheduling System for SABRE Travel Information Network (STIN)"

[20] G. Dreo Rodosek, T. Kaiser, R. Rodosek, "A CSP Approach to IT Service Management"

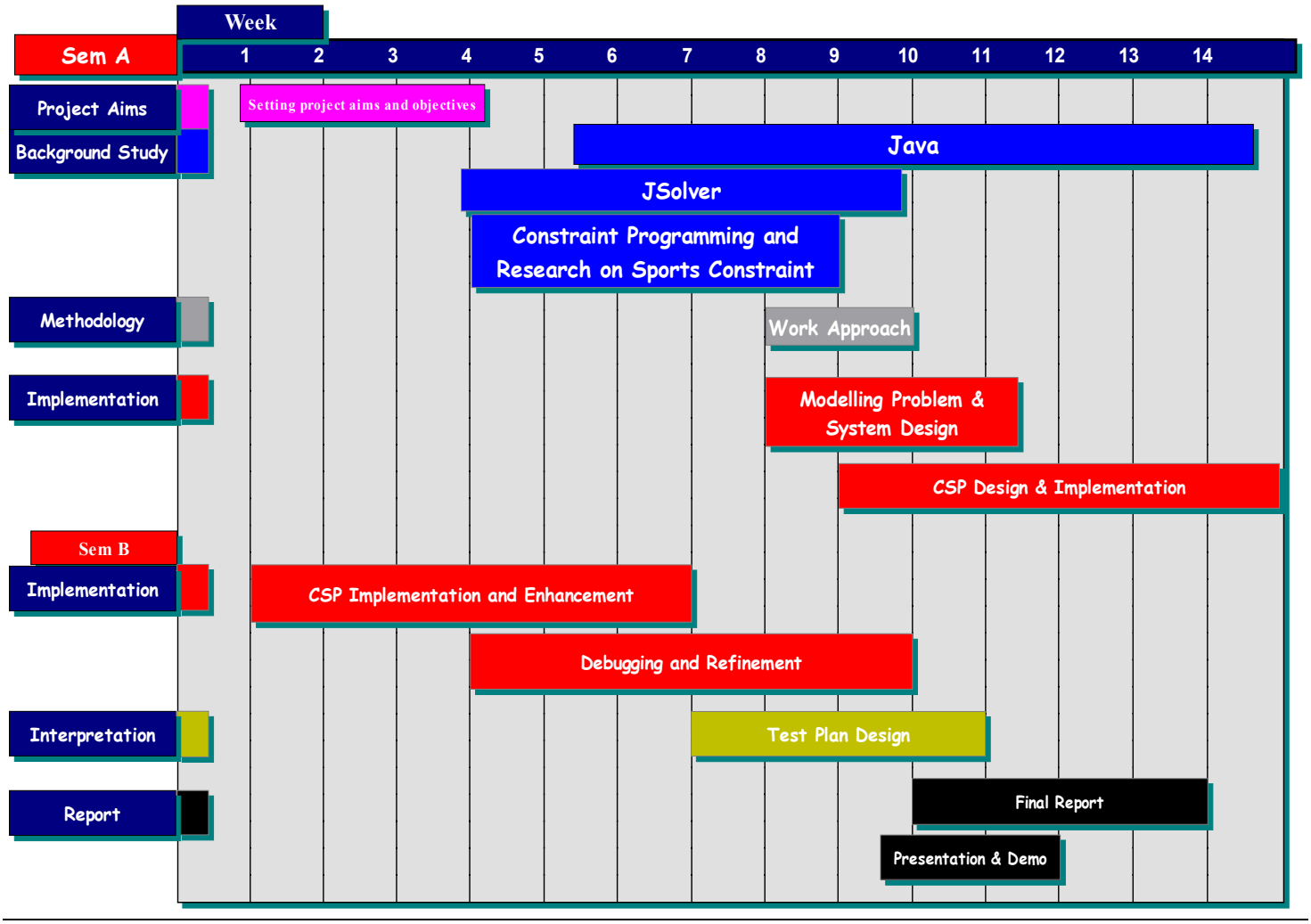
[21] Shengxiang Yang & Dingwei Wang, "Constraint Satisfaction Adaptive Neural Network and Heuristics Combined Approaches for Generalized Job-Shop Scheduling" *IEEE Transactions on Neural Networks*, Vol. 11, No.2, March 2000

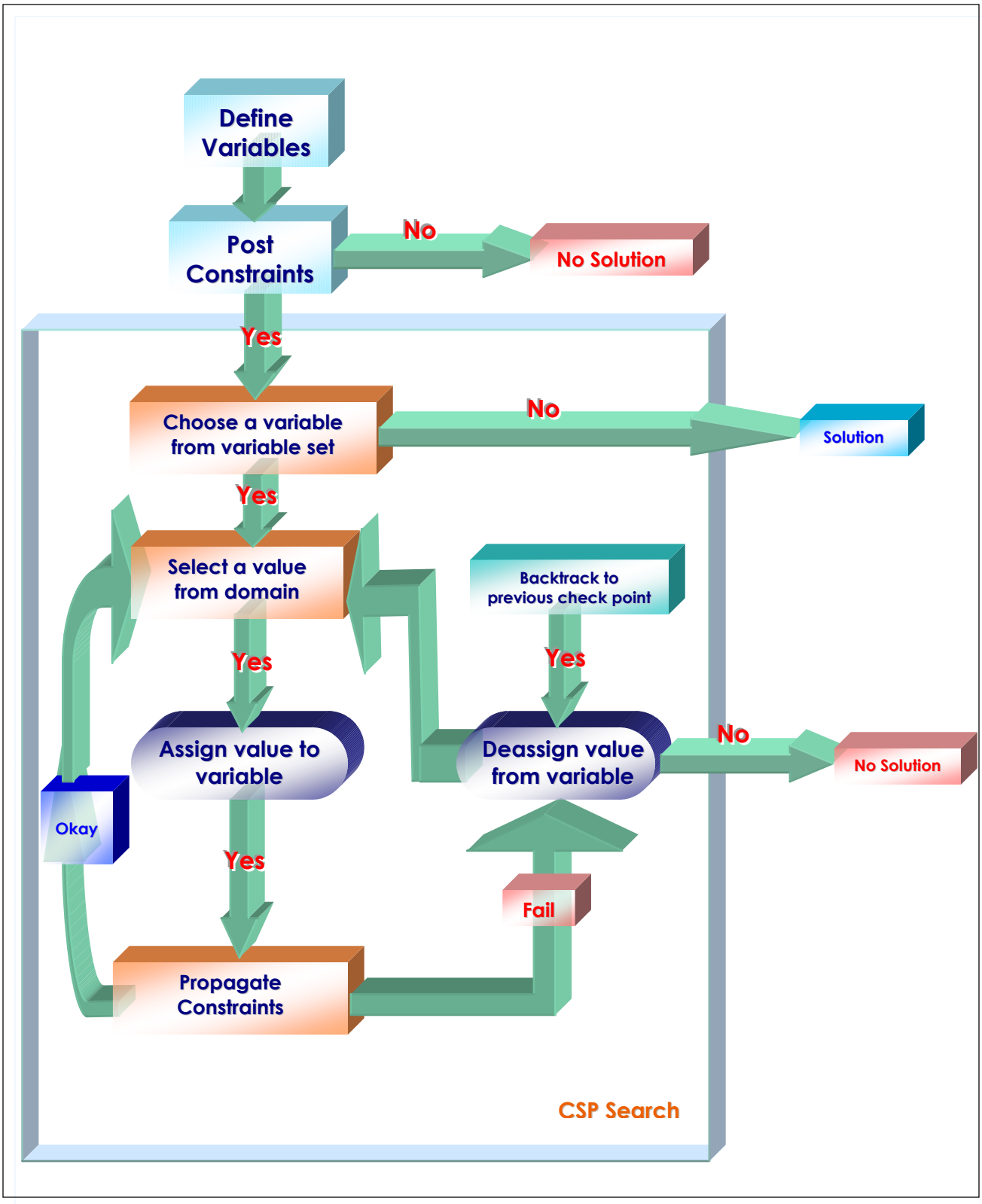
[22] Schreduer, Jan A. M, "Combinatorial aspects of construction of competition dutch professional football leagues, " *Discrete Applied Mathematics*, 35:301-312, 1992.

[23] Andy Hon Wai Chun, Ngai Ming Lam "Using Heuristics in Constraint-based Sports Tournament Timetabling" Accepted for presentation in the Sixth World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002)

XVI Appendix

Original Project Planning Time Chart




Define Variables
Post Constraints
No
No Solution
Yes
Choose a variable from variable set
No
Solution
Yes
Select a value from domain
Backtrack to previous check point
Yes
Yes
Assign value to variable
Deassign value from variable
No
No Solution
Okay
Yes
Fail
Propagate Constraints
CSP Search

Example of Redundant Constraint

The problem of the magic sequence is described in Example: *Magic Sequence* [9], beginning on page 62. Let's describe it here briefly as a demonstration. Let's assume that we have $n+1$ unknowns, namely, $x_0, x_1, \ldots, x_n$. These x_i must respect the following constraints:

- 0 appears x_0 times in the solution.
- 1 appears x_1 times.
- In general, i appears x_i times.
- n appears x_n times.

However, the search for a solution can be greatly accelerated by introducing the following redundant constraint that expresses the fact that n+1 numbers are counted.

$$1\,x_1 + 2x_2 + \ldots + n\,x_n = n + 1.$$