UNIVERSITY OF ALBERTA

A Constraint Satisfaction Approach to Timetabling

BY

Don Banks ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Masters of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1996

ISBN 0-612-18232-0

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Don Banks

TITLE OF THESIS: A Constraint Satisfaction Approach to Timetabling

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1996

(Signed) . . . .

Don Banks
248 Brookside Terrace
Edmonton, Alberta
Canada, T6H 4J6

Date: . . /. . . . .

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Constraint Satisfaction Approach to Timetabling** submitted by Don Banks in partial fulfillment of the requirements for the degree of Masters of Science.

[illegible]

Dr. P. van Beek (Supervisor)

[illegible]

Dr. A. Meisels (Co-Supervisor)

[illegible]

Dr. F. Lau (External)

[illegible]

Dr. R. Goebel (Examiner)

[illegible]

Dr. T.A. Marslund (Chair)

Date: Sept. 12, 1996

Abstract

The general timetabling problem is an assignment of activities to fixed time intervals, adhering to a predefined set of resource availabilities. Timetabling problems are difficult to solve and can be extremely time-consuming without some computer assistance. In this thesis the application of constraint-based reasoning to timetable generation is examined. Specifically, we consider how a timetabling problem can be represented as a Constraint Satisfaction Problem (CSP), and propose an algorithm for its solution which improves upon the basic idea of backtracking. Normally, when a backtracking routine fails to find a solution, there is nothing of value returned to the user; however, our algorithm extends this process by iteratively adding constraints to the CSP representation. This algorithm can be considered a general methodology for handling over-constrained networks. We solve three high school timetabling problems using this algorithm, yielding master timetables which are superior to human-generated timetables, since we are consistently able to schedule more than 98 percent of the students. Furthermore, a generalized random model of timetabling problems is proposed. This model creates a diverse range of problem instances, which are used to verify our search algorithm and identify the characteristics of difficult timetabling problems.

Acknowledgements

I would like to thank Mr. C.L. Lund, Mr. T. Rankel, Mrs. F. Schmidt, Mrs. K. Johnson and Mr. M. Demaine for their assistance in my understanding of the high school timetabling problem, and for providing me with the data that was used for this thesis. Their contributions have been invaluable.

Contents

Chapter 1

Introduction

Timetabling prol [illegible] arise in many real world situations. Although many computerized technique [illegible] or timetable construction, obtaining acceptable results is often difficult. In re[illegible] years, constraint-based reasoning has gained much attention in the Artificial Intelligence community. Using the CSP paradigm, our aim is to model and solve high school timetabling problems.

1.1 Overview

There is no standardized or commonly accepted terminology when describing the general timetabling problem. We begin with a loose description of this problem before discussing its variety. First, there are the *participants* which need to be scheduled, such as teachers, lessons, lecture halls, pieces of equipment and so forth. Generally, there will be more than one set of participants. Next, there is a set of *time slots* which dictate fixed points of activity commencement and termination. These time slots can also be called periods, hours, or even days; the important characteristic is that there is a specific number of them, each with pre-determined starting and stopping times. The third component of the timetabling problem is the set of *availabilities*, which dictate the subset of time slots in which each participant is able to be scheduled. In the simplest timetabling problems, the availabilities are simply lists of time slots for each participant. However, more complicated problems arise [illegible] there are constraints on the availabilities. For example, in the conference scheduling problem, if the participants are lecture theatres and the time slots are one hour intervals, there may be an availability which says that no lecture theatre may have more than one conference at the same time.

Timetabling and scheduling are two different problems, although in the literature the terms are sometimes used interchangeably. The main difference is that timetables make use of fixed, pre-arranged time slots. Schedules, on the other hand, can have their activities beginning and ending at arbitrary points in time.

The major areas of timetabling applications are manpower allocation, transportation problems and school timetabling. The commonality between all of these types of problems is the *time constraint*. This requirement says that no two participants may simultaneously occupy the same "block" of time. Manpower allocation, for example, generally describes a problem in which employees are scheduled into shifts, within other constraints such as a minimum number of hours per week for each employee. Other constraints may or may not be present in timetabling problems, such as an industrial shop requiring its machines to perform tasks in a certain order, or a railway requiring 3 engineers on duty at a time, and so forth. It is the nature of these constraints which distinguish different timetabling problems.

Furthermore, the scheduling goals of timetables may also differ. Some problems may require a single, "master timetable" of global events, or individual timetables, such as those issued to students, for a number of different resources. Some constraints in the timetabling problem are *rigid* and must not be violated, while others may be relaxed at an individual's preference; for instance, a bank may have the rigid requirement that its employees be scheduled between the hours of 9 am to 5 pm, while there may be a preference amongst the employees as to which days are taken off. Finally, some timetabling problems may involve the *satisfaction* of a number of constraints, while others require *optimization*. The satisfaction problem, which is the more simple of the two, may ask the question, "Can these students be scheduled in a school with 30 rooms?", while an optimization problem might be, "What is the fewest number of rooms required to schedule these students?".

The existing literature on timetabling is rich and diverse. It would seem that there are as many different timetabling problems as there are solution techniques. The main effort in this field recently has been seen in the Operations Research (OR) and Artificial Intelligence (AI) communities. The OR approach has been generally that of Integer Linear Programming. Using this strategy a timetabling problem is expressed as a system of linear equations. AI researchers have been using another general technique: constraint logic programming. Several CLP languages, such as CHIP [15] and Pecos [8] have been developed specifically for solving scheduling problems. The general technique is to express the problem using logic predicates, and proceed to find a solution by using *backtracking* and various *heuristics*. Within constraint logic programming lies a specific means of representing problems known as the Constraint Satisfaction Problem (CSP). Research on CSP's has blossomed in the last decade. However, most of the experimental work has been done on random binary constraint satisfaction problems. There have been relatively few reports of successful applications of CSP's to real world timetabling problems.

The contributions of this thesis are threefold. First, we have produced a CSP representation of a complex, real-world timetabling problem. Modelling a real world problem is difficult, since there are many nuances which need to be expressed. For example, our problem deals with constraints on the number of teachers in each subject, a finite capacity of rooms in the school, classrooms having a maximum of thirty students, and so forth. Much of the work in the literature on school timetabling focusses on more simple problems, such as room assignment or teacher scheduling, allowing for convenient reductions of the problem. However, our problem encompasses the broad problem of creating a master timetable of courses, along with scheduling students into their specified courses. The second contribution of this thesis is our solution technique. By prioritizing the various constraints, our algorithm will iteratively add the constraints to the problem, building upon the solution until no further improvements can be made. This technique is a general methodology for handling over-constrained networks, which is applicable to any constraint logic problem. Our CSP representation is justified by the algorithm's performance on three local high schools, on which we are able to schedule more than 98% of the students. The third contribution of this thesis is the random problem generator. We are able to demonstrate the *robustness* of our algorithm by solving a diverse range of timetabling problems. We show that, in most cases, our algorithm can satisfactorily schedule 95% of the students within a school of any particular number of rooms and courses offered. The amount of computational time required by the algorithm is limited to a few minutes. Furthermore, the generator helps us to identify difficult to solve timetabling problems. These problems can then be *recreated*, by giving the algorithm the same random seed, for testing on future timetabling algorithms.

1.1.1 Specification of the Problem

This thesis encompasses three broad areas of work: timetable problem representation, solution methods and random problem testbeds. We describe a timetabling problem found in high schools in Edmonton, Alberta. This particular problem involves creating a *master timetable* of courses into periods. The schools being studied have eight periods per week in each of two school semesters. Some courses are scheduled within one semester, while others are full year courses which must be taken in both semesters. Courses are divided into sections of students, and each section is scheduled into its own period. The input to the problem consists of a list of student course selections. The objective is to satisfy as many of the student course selections as possible by scheduling the sections of the courses into the eight weekly periods. Furthermore, the teachers of the school and the capacity of the school are considered. There are only a certain number of teachers for each subject, and so there is a limit on the number of courses of each subject that can be taken at a time. As well, there is a limit on the total number of rooms in the school, and there cannot be too many courses scheduled during any particular period.

A solution is one that satisfies all of the students' sets of course selections; however in practice this is not always possible. Therefore, we define a *satisfactory* solution as one that meets $N_{sat}\%$ of the students' course demands. The value of N_{sat} can be manipulated by the user, and is generally accepted at being between 95 and 100. In reality, the large high schools will settle for between 92 and 97 percent student satisfaction.

1.1.2 Motivation

There are several reasons for examining the application of CSP's to timetabling problems. First, it is valuable to understand the performance of search algorithms on structured, non-random constraint networks. As will be discussed in the next chapter, experimentation on CSP's has been mostly performed on contrived, "toy" problems, or purely random problems. These experiments are useful for obtaining a general understanding of search-intensive problems. Our research serves to complement previous results by examining more practical applications. The CSP representation of a real world problem is indeed practical, and we illustrate that existing CSP solution techniques can be applied. Of particular interest is our use of non-binary constraints as a means of structuring global attributes. These constraints clearly add to the difficulty of the problem. In the literature, non-binary constraints are often overlooked in CSP experimentation. Further motivation for our work is that timetabling is extremely common and is seen in everyday situations. As we have discussed, there are many different types of scheduling problems and many different solution techniques. We believe that understanding one particular problem enables a better understanding of other timetabling problems. Since we have gained experience in modelling constraints and handling over-constrained networks, these general techniques can be applied under different timetabling circumstances.

The described high school timet[illegible]ing problem is extremely challenging. In fact, at a local school of about 1650 stu[illegible] the administration requires several man weeks to arrive at a satisfactory mas[illegible]er timetable. There is no existing software available to perform the scheduling for them. Although they use a database to store student records, teachers' hours and validate course requests, the actual assignment of periods 1, . . . , 8 to course sections, and checking for course conflicts must be done by hand. Lastly, although school timetabling problems have been frequently examined [1] [5] [9], often the problems are simplified and do not include all of the actual constraints. We believe our representation of the problem is complete, and would not require any "fine-tuning" by a human scheduler.

1.1.3 Summary and scope of the thesis

In chapter 2 of this thesis we introduce the reader to the Constraint Satisfaction Problem. We provide a simple example of the CSP being applied to an imaginary problem, the Zebra Problem, and discuss some CSP solution techniques.

In chapter 3 of this thesis we explore the background behind the timetabling problem. There is indeed a wide variety of solution techniques: constraint logic programming, integer linear programming, simulated annealing, genetic algorithms and many others. We consider the place of high school timetabling within the context of all scheduling problems.

In the fourth chapter we present our CSP representation of the local high school timetabling problem. We have chosen a compact representation requiring a manageably low number of constraints. The *variables* of our CSP are *courses* within the school. Recall that a course is actually a set of *sections* of lessons which meet in the same period three times per week. Therefore, we express the *domains* as tuples of period values. For example, a course with three sections may have a legal domain value of $(1, 2, 4)$, indicating that the three sections will be in periods 1, 2 and 4 respectively. A heuristic we use to greatly reduce the domain size is to eliminate domains with the same period appearing more than once; that is, a tuple of values such as $\{1, 2, 2\}$ would not be considered. There is nothing wrong with having two sections of a course scheduled in the same period, but we choose to ignore this practice in the interest of diversifying the sections to increase the chances of student satisfaction, and decreasing the domain sizes and the overall search space.

In chapter 5 we propose an algorithm that iteratively adds constraints before proceeding with a backtracking search. We define the *binary constraints* to occur between pairs of courses chosen by the same students. We find that binary constraints between all pairs of chosen courses results in no solution existing to the problem. Thus a constraint *weighting* function is required to decide which constraints will satisfy the most students. Given two courses, one with x students enrolled, the other with y, and given that k students have registered in both courses, we define the weighting of the binary constraint between the two courses as

$$k/min(x, y)$$

A *threshold* value is used to determine the minimum weighting to include the binary constraints, and a forward checking routine is used to solve the specified CSP — which implies assigning periods to sections — and a master timetable is specified. Then, the procedure *iterates* and raises the threshold to include more binary constraints, which should increase the number of students successfully scheduled. The iterations continue until too many binary constraints are added and no solution can be found. Using three high schools' student data, we are able to successfully schedule more than 98% of the students.

In chapter 6 we explore the idea of generating random timetabling problems. Although we show in chapter 5 that our algorithm performs well on three local high schools' student data, we would like a wider array of test problems. In particular, we are interested in varying the *tightness* and quantity of the constraints. We introduce the two *non-binary* constraint classes of the timetabling CSP as the global rooms constraints and the teachers' constraint. The rooms constraint states that given a school with r rooms, no more than r courses can be scheduled in any period, since there would not be enough rooms. The teachers constraint says that given a *subject*, such as English, which has k instructors available, no more than k sections of that subject can be scheduled at a time, since there would be an insufficient number of teachers. By varying the number of students, courses and rooms in our random model, we are able to create a large testbed of problems akin to the random binary model used by other researchers. With the addition of non-binary constraints we have modified the forward checking algorithm. Several experiments reveal that when the number of rooms approaches some minimum, there is a sharp rise in the amount of effort needed to find a solution. We also identify difficult-to-solve problems when the number of courses becomes high and the number of students low. For the large part, though, our algorithm is almost always able to satisfactorily schedule at least 95% of the students on the randomly generated data.

We present our conclusions and discuss some possible future work in chapter 7.

Chapter 2

Background on Constraint Satisfaction Problems

Constraint-based reasoning is a simple means of representing a wide variety of problems. In this chapter we proceed with a succinct definition of the CSP, before discussing some of the solution techniques and drawbacks of the previous work on CSP experimentation.

2.1 The Constraint Satisfaction Problem

The CSP has three components: the *domains*, *variables* and *constraints*. Each CSP consists of a set of variables $\{x_1 \ldots x_n\}$, each with an associated domain of values $D_1 \ldots D_n$. A solution to a constraint satisfaction problem is an instantiation of each variable to one particular value from its domain, such that none of the constraints are violated. Constraints, therefore, are relations between variables which describe their legal values. These relations may be represented in an extensional or intensional manner. For example, suppose variable a has the domain $\{1,2,5\}$ and variable b has the domain $\{2,3,4\}$. A *binary* constraint — one that proposes the valid instantiations between *two* variables — may exist which says that $a < b$. An extensional representation of this relation is a subset of the Cartesian Product of the two domains giving the legal pairs of values:

$$\{(1,2),(1,3),(1,4),(2,3),(2,4)\}$$

While an intensional representation of this relation would be:

$$\{(a,b)|a \in \{1,2,5\}, b \in \{2,3,4\}, a < b\}$$

Another way of stating the extensional relation between two variables is to use a constraint matrix. This matrix of Boolean values declares the valid and non-permissable pairs as true and false values. For example,

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

illustrates the constraint $a < b$, where the columns are the different values of a, the rows are the values of b and a "1" (true) corresponds to a valid pair of values, while the "0" (false) represents an invalid pair. This matrix structure allows for some convenient CSP *pre-processing* algorithms to eliminate domain values. For instance, the column of zeroes in the matrix tells us that the value "5" for the variable a yields no legal instantiations of variable b, therefore, "5" can be *pruned* from the domain of a before any searching begins. However, this constraint matrix structure is really just an implementational detail, as it is an internal representation of data.

Not all constraints are binary. A *unary* constraint is one that applies to a single variable. Suppose, for example, there are five variables in a CSP, each representing one of the days of the week Monday through Friday for an employee. The domains of these variables might consist of hours of the day. A unary constraint may explicitly say that the employee does not work past 8 pm on any day. Thus, each variable would have the hours beyond 8 pm removed from its domains. Typically, unary constraints are pre-processed. More complicated constraints which can arise are *non-binary*. For example, suppose we have three variables, a, b and c, each with the domain $\{0\ldots9\}$. A non-binary (or ternary in this case) constraint may enforce that:

$$\{(a,b,c)|a+b+c=15\}$$

A non-binary constraint which includes all n variables of the CSP is known as a *global* constraint. Non-binary constraints are not usually explicitly stored as Boolean matrices, since this often requires an impractical amount of time to compute and store. These constraints are generally represented intensionally.

2.1.1 A sample problem

Determining the variables, domains and constraints of the CSP is the first step in finding a solution. This first step is not always easy, as some problems do not lend themselves to such a clear structure. As an example, let us define the *zebra problem*. This problem is one that has been often mentioned in CSP papers, serving to test new CSP algorithms [17]. The problem is simple enough to understand, but large enough to demonstrate the efficiency of an appropriate CSP representation.

The zebra problem goes as follows: There are five houses; in each house there is a woman of different nationality, each woman has a different pet, each woman smokes a different brand of cigarettes, each woman drinks a different beverage, and each house is a different color. We know the following facts about the situation at hand:

- The Englishwoman lives in the red house
- The Spaniard owns the dog
- Coffee is drank in the green house
- The Ukrainian drinks tea
- The ivory house is to the right of the green house
- Oldgolds are smoked by the woman who owns the snail
- Kools are smoked in the yellow house
- Milk is drank in the middle house
- The Norwegian lives in the house on the far left

- The woman who smokes Chesterfields lives next to the woman who owns the fox
- The woman who owns the horse lives next to the yellow house
- Lucky Strikes are smoked by the woman who lives in the orange house
- The Japanese smokes Parliaments
- The Norwegian lives next to the blue house
- One woman drinks water
- One woman owns a zebra

The problem asks where do each of the women live, what color are their houses, what are their pets, which cigarettes do they smoke and which beverages do they drink? There is only one solution to this problem, despite the 25^5 possible different combinations. How can this problem be cast as a CSP? In other words, what are the variables, domains and constraints which allow us to conveniently solve the problem?

An initial, naive approach may be the following: Suppose there are 20 variables: five for owner-pet pairs, Englishwoman-Pet, Spaniard-Pet, and so on, each with the domain $\{dog, horse, fox, snail, zebra\}$, five variables for the owner-color pairs: Englishwoman-House, Spaniard-House ..., each with the domain $\{red, green, blue, ivory, yellow\}$, as well, similar definitions for the sets of variables owner-smokes and owner-drinks. What are the constraints under this scheme? We know, for example, that the Englishwoman lives in the red house, so the variable for Englishwoman-House could be pre-defined to be "red". However, we have no information with respect to other owners and their pets. Furthermore, information such as "the ivory house is to the right of the green house" are difficult to account for. A good CSP representation is one with constraints that are as specific and as descriptive as possible. Under this naive approach, there are many variables which cannot be constrained at all.

A better representation of this problem is as follows. There are 25 ***variables***, one for each of the five nationalities, and one for each of the five colors, beverages, cigarettes and pets. If we number the houses from left to right as 1, 2, 3, 4, 5, we now have the *domains* for each of the variables, $\{1, 2 \ldots, 5\}$. The *constraints* are now easily described. For example, "The Englishwoman lives in the red house" is simply saying that the variable for "Englishwoman" must have the same value (house number) as the variable for "Red". Similarly, the variable "Spaniard" must be equal to the variable "Dog". These kinds of equality constraints can be specifically modelled by the relation:

$$Englishwoman = Red$$

or by using the intensional notation of the legal pairs of values:

$$\{(1,1),(2,2),(3,3),(4,4),(5,5)\}$$

Since we are given the statement, "the Norwegian lives in the house on the far left", the domain of the variable "Norwegian" could be initialized to 1.

The "next to" constraints, such as green is next to ivory, are easily modelled since we have numbered the houses 1 through 5. We know that 1 is next to 2, 2 is next to 1 and 3, and so forth, so the general relation of "next to" is modelled by using absolute values:

$$\{|Green - Ivory| = 1\}$$

or

$$\{(1,2),(2,1),(2,3),(3,2),(3,4),(4,3),(4,5),(5,4)\}$$

This formulation of the problem is superior to the naive approach because it *exploits* the natural structure of the problem. By enumerating the houses 1 through 5, the other participants in the problem are more easily identified.

2.2 Solution techniques

2.2.1 Backtracking

A solution to a CSP consists of an assignment to each variable of one value from its domain, such that all of the constraints are satisfied. A typical method of finding a solution to a CSP is a tree search algorithm. Also known as *backtracking*, the most naive form of this algorithm was discovered more than a century ago, and has been rediscovered many times since. In simplest terms, the backtracking algorithm will at each step instantiate a variable with a value from its domain. Then, all of the constraints that the variable participates in are checked, and if one is violated, the procedure must backtrack to a previously instantiated variable in the tree. This process is systematic, and will check all possible values from each domain until a variable is satisfied. If there are n variables, each with uniform domain size d, the search space may be thought of as a tree, with the "root" being the first variable to be instantiated, and each node in the tree having branches to d nodes for the next variable. Therefore, in the worst case the algorithm explores d^n nodes, but just n in the best case if a solution is found without any backtracking.

In general, this naive backtracking algorithm is not used in practice. When an inconsistent variable is found, the backstep is only one node in the search tree. There are other more complicated tree search algorithms which have received attention. The Back*jumping* routine of Gaschnig[13], for example, attempts to minimize the number of nodes visited by backtracking directly to the variable which is in conflict. A different approach is the explicit forward checking algorithm of Haralick and Elliot[13]. When the search routine instantiates a variable it looks ahead to the future, uninstantiated variables, and removes from their domain any values which are currently incompatible. If the domain of a variable is completely annihilated, the procedure backtracks chronologically.

2.2.2 Branch and Bound

Branch and bound, like naive backtracking, is a *retrospective* technique, meaning that a new value is selected to try to extend an incomplete solution by "looking back" to test consistency. However, branch and bound serves to find a *maximal* solution, i.e. one that violates the *fewest* number of constraints. Backtracking, on the other hand, will halt if it has found a perfect solution — one that violates *no* constraints — or has exhausted all of the possibilities.

Branch and bound will initially build up a partial solution, keeping track of the number of constraint violations. The complete solution may result in, say v violations, which then becomes the *bound* of the search. The algorithm then systematically attempts all other possible "branches" of the search tree, but now will abort the current branch if the partial solution has exceeded v violations. If a partial solution is completed, and has fewer than v constraint violations, then v is now updated and this solution is saved as the "best" so far. The Branch and Bound algorithm is considered an "anytime" algorithm, because at any point in the search process, there is a best solution available to the user. The quality of the solution is proportional to the algorithm's running time.

2.2.3 CSP Experiments

In the literature are many discussions on experiments on CSP's. These experiments are usually performed on benchmark problems, such as the "Zebra" problem or the n-queens problem, or experiments may be on completely random problems. In the n-queens problem, a $n \times n$ chess board is used to position n queens such that none are attacking each other. Random CSP's are produced by an algorithm which generates binary constraints by creating random Boolean matrices. The experimenter can vary the number of constraints, the "tightness" of the constraints (referring to the ratio of 0's to 1's in the matrices), as well as the number of variables and the size of the domains. These random CSP's have the virtue of offering a diverse range of problems which are easy to generate. There have been few reported experiments on problems with non-binary constraints, however.

Experiments are usually performed to test new backtracking algorithms or *heuristics*. The two main types of heuristics are *variable* and *value ordering*. The former will determine the order in which the variables are instantiated by the search algorithm, while the latter orders the values within the domains to be instantiated. The success or failure of a heuristic or algorithm is usually judged by the total number of consistency checks (which is better than simply measuring the run-time, since that may vary from machine to machine). The forward checking routine, for example, can require several orders of magnitude fewer consistency checks than the simple backtracking routine on the same problem.[13]

Our research follows from these ideas of CSP experimentation. There are, however, some shortcomings of the previous work in this area which we have overcome. The general random binary CSP's do not resemble anything in reality, since real world problems are far more *structured* in nature. In fact, these random problems are only useful for a *theoretical* examination of CSP properties. While this work is useful, our work is more centred on a practical application. Since we have developed an algorithm specifically for timetabling problems, it would be pointless to test it on purely random binary CSP problems. Furthermore, our problem has non-binary constraints, which are not included in the general binary model. Thus, we see the motivation in developing a random timetabling problem model. We wish to create an adequate supply of problems with differing binary and non-binary constraints. This model has three parameters, the number of courses (variables in the CSP), the number of students (which influences the distribution of the binary constraints and the domain sizes) and the capacity of the school (for the non-binary constraint). In doing so, we are able to generate a diverse range of timetabling problems suitable for testing our algorithm. Furthermore, we identify difficult to solve timetabling instances which occur when the global capacity of the number of rooms approaches a minimum value, depending on the number of students.

We propose a backtracking algorithm which iteratively adds constraints to the network until no solution remains. Backtracking algorithms will either find a solution, and then halt, or else find no solution and halt. In the latter case, the user is left with nothing of value. However, our algorithm assigns *weights* to the binary constraints and then iteratively adds them to the CSP. The motivation for these steps is that if all of the constraints are added, then there is no solution at all. Several iterations may be necessary until a satisfactory solution is found. However, there will always be a timetable generated, and the user may halt the search routine at any time to retrieve the best result found up to that point.

Having described the constraint satisfaction problem, we will proceed in the next chapter with a formalization of timetabling, and describe some of the previous work in timetable generation.

Chapter 3

Background on Timetabling

The problem of timetabling is the focus of ongoing research in both the Artificial Intelligence (AI) and Operations Research (OR) communities. In general terms, timetabling is a matter of *resource allocation*. Activities (or "participants") consume resources, which are to be scheduled over time. The problem may be to create a *master schedule* for all resources, or to create individual schedules for each activity. Timetabling refers to the particular set of scheduling problems where the starting and completion times available for the activities are known in advance. The scheduling process requires satisfying or sometimes optimizing a number of constraints. These constraints may assert things such as activity durations, resource capacities, resource availability, and so forth. Further, these constraints may need to be rigidly enforced, or may be relaxed with a particular level of preference. The rigid constraints will define the subspace of allowable solutions, while the relaxable constraints characterize the quality of the scheduling solutions. Timetabling is a problem which is known to be NP-Complete[7] . In this chapter we will offer precise definitions of the three main categories of scheduling problems — manpower planning, transportation, and academic scheduling. We will further describe the difference between the AI and OR approaches to solving the scheduling problem before looking specifically at some past applications of CSP's to timetabling problems.

3.1 Timetabling constraints

Let us discuss the basic timetabling problem. First, there is the set of *participants* being scheduled. Within this set may exist the *resources* and *activities*. Under this general model, it is the activities which consume resources, and the activities are to be scheduled. However, not all timetabling problems require both sets: there may simply be one uniform group of participants being scheduled. Furthermore, the terms "resources" and "activities", can be interchangeable; for example, under the common *class-teacher* timetabling problem, groups of students (classes) are assigned teachers, or the problem may be viewed as teachers being assigned classes. The next component of the timetabling problem is the set of *time slots*. These are simply pre-determined, fixed points in time at which the activities begin and end. The third component of the general timetabling problem are the availabilities. Each participant may have specific times at which they can or cannot be scheduled. Therefore, the availabilities may be thought of as sets of legal time slots for each participant. A solution to a timetabling problem is an assignment of the activities to the time slots, such that all of the availabilities are met.

There may also be availabilities which are rules, specifying legal combinations of activities to resources. For example, "each class needs one and exactly one teacher" is an availability. Or, "employees must not work a night shift followed by a morning shift the next day" is another. The nature of the availabilities are what distinguish different timetabling problems. Thus, in the next section we describe the main types of availabilities, also known as constraints, before proceeding with some particular examples from the literature.

3.1.1 Not-equals (mutual exclusion) constraint

At the heart of most scheduling problems is the fact that events cannot occur at the same time. Indeed, were this constraint not present, the problem would either be extremely easy, or not a timetabling problem at all. To demonstrate these mutual exclusion constraints, consider a problem of Examination timetabling. This problem is commonly faced by Universities when, at the end of the term, final examinations are to be taken by students. As a simple example, suppose there are five exams, $E_1 \dots E_5$, which need to be scheduled, and there are three days in the examination period, Monday, Tuesday and Wednesday. Each student can only take one exam in each day.

- The **participants** of the problem are the examinations (five)
- The **time slots** of the problem are days (three)
- The **availabilities** are that no student can take more than one examination in a day. The difficulty is that there are some students who are taking more than one exam. As long as there is at least one student taking E_i and E_j, then E_i and E_j may not be scheduled on the same day.

Suppose that a check of the class lists reveals that there are some students who must take E_1 and E_3, E_2 and E_4, and one student who takes E_1, E_4 and E_5. Therefore, there are not-equals constraints between the pairs of courses, E_1-E_3 E_1-E_4, E_1-E_5, E_2-E_4 and E_4-E_5.

A solution to this problem is an assignment of days to the five examinations, such that all of the availabilities are met. One such solution is:

- E_1 is scheduled on Monday
- E_2 is scheduled on Monday
- E_3 is scheduled on Tuesday
- E_4 is scheduled on Tuesday
- E_5 is scheduled on Wednesday

The not-equals constraints in this example are simplistic, since there is only *one* group of participants being scheduled. The more typical case is when there are two groups of participants, resources and activities. This same examination timetabling problem could have the added scheduling need of the exams being assigned to *rooms*. Now, there are two groups of participants, examinations and rooms, both of which are being assigned to time slots. The *intrinsic* not-equals constraints which exists in this problem are that no two exams can take place in the same room, nor can two rooms be assigned to the same exam. This constraint between two participants describes the *bipartite* scheduling problem[3].

3.1.2 Capacity constraints

The capacity constraint can exist in many different ways. One type of capacity constraint is an upper or lower limit on the participants being scheduled. Returning to the examination scheduling problem, a school may be faced with a problem of a shortage of rooms. If there are r rooms, then a capacity constraint enforces that no more than r exams may be scheduled during one particular time slot. Other capacity constraints can exist when there is more than one group of participants. In hospital shift scheduling, for instance, each shift may require exactly three doctors and five nurses.

3.1.3 Temporal constraints

The third type of constraint describes temporal relationships between the activities being scheduled. One kind of temporal constraint is logically expressed by a formula $(I1 + d \leq I2)$, where I1 and I2 are time points representing beginning and ending times of activities, and d is a minimal delay that elapses between the two time points [8]. In exam scheduling, there may be a required one day ($d = 1$) break between any students examinations. Or, in course scheduling, a physics lecture may be followed immediately by a physics laboratory ($d = 0$). Further, complex sequences of operations may be represented by Boolean functions such as $during(J_1, J_2)$ (job 1 must occur while job 2 is occuring), $precedes(J_1, J_2)$, $overlap(J_1, J_2)$. In short, these constraints enforce some sequential relationship between the times assigned to the resources.

3.1.4 Constraint preferences and problem difficulty

In real timetabling problems, not all of the constraints need to be satisfied for a solution to be acceptable. As well, it may not even be *possible* for all of the constraints to be satisfied. In either of these cases, some degree of preference must be specified as to which constraints are more important than others. Constraints which specify suitable time slots for activities, such as, a particular instructor would like to teach only afternoon classes, are often assigned *weightings*. The constraints involving instructors with a higher seniority, or more unavoidable time restrictions, are assigned a higher weighting. The objective, then, is to find the solution which maximizes the total weighting of satisfied constraints. Another option is to simply eliminate unimportant constraints until an acceptable solution can be found. Generally, the most important constraints are the not-equals constraints, since they are the fundamental basis for the timetable.

The nature of the constraints describe three levels of theoretical difficulty in timetabling problems. The simplest goal is *feasibility*, the decision problem. Here, we are asked, is there a feasible solution to the constraints? In the examination timetabling problem, we may ask, given 25 exams and a particular set of availabilities, are six time slots enough to allow a suitable timetable to be constructed? In reality, the feasibility scheduling question is relatively uninteresting. The second type of scheduling goal is *Satisfiability*, which asks the question, what is a feasible solution to the problem? Finally, *optimization*, the most difficult of the objectives, says, assuming we are given a cost function that estimates the quality of each solution, which is the best? This problem is analogous to finding *all* satisfiable solutions.

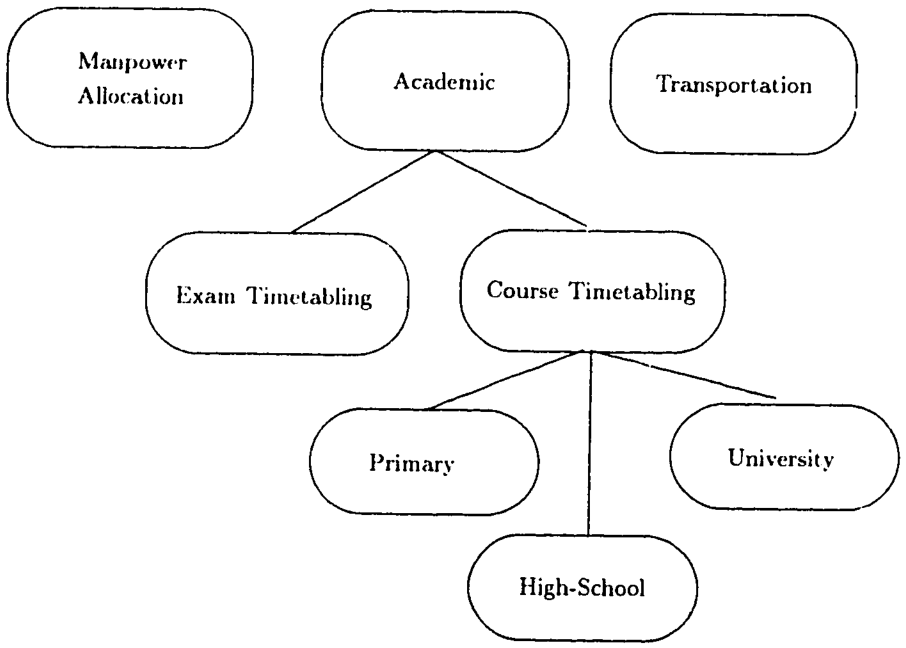

Figure 3.1: The applications of timetabling generation discussed in the literature.

3.2 Examples of timetabling problems

In this section we describe the three main areas in which timetabling problems can be found. with particular emphasis on academic scheduling. Figure 3.1 supplies the reader with an illustration of these three main areas. In the following sections we stress the differences between each category of problems.

3.2.1 Manpower allocation

Also known as the general *manpower planning problem*, here employees must be assigned to shifts on a daily or weekly basis. Any institution, such as a hospital, factory or bank may have several shifts which require differing numbers of personnel. The formal definition of the problem is:

- The **participants** are sets of employees. Each set represents a group with different scheduling needs, such as doctors, nurses, janitors, and so forth. The other participants are the sets of *tasks*
- The **time slots** are generally *shifts*, several hours in duration
- The **availabilities** are: (*Mutual Exclusion*) Each employee has only one task at a time, and each task is accomplished during its available hours, (*Capacity*) Each task has the required number of employees assigned to it. Each employee has a maximum number of working hours that can be allocated.

For example, a ship yard may require 1 chief, 1 foreman and 7 operators for each of its three 8 hour shifts in a day. This rigid constraint is coupled with the fact that each individual employee is constrained by the number of shifts they may work over the week or month. There may be other preferences, for example in [11] P. Jacques found in his study of the Banque Bruxelles that management wished to have employees not work two consecutive weeks of night shifts, or to work six consecutive days. Constraints of this type are typical to some of the various manpower planning problems. As well, the set of *tasks* may be omitted in some problems, if each employee has one particular job to perform at all times.

3.2.2 Transportation problems

The transportation problem deals with a network of locations, a corresponding function determining the cost of travel between each location, and a set of participants which will be travelling across the network. There will exist some kind of time constraint, which asserts the particular times on which the agents may or may not be in particular locations. For example, Puget [14] describes locomotive scheduling as a transportation problem. In this problem, given a set of trains and their departure and arrival time and stations, the goal is to minimize the number of locomotives required to meet the entire railway demand. Thus, each locomotive requires its own schedule of stations and departure times. More generally we define the transportation problem as:

- The **participants** are a set of resources and a set of locations
- The **time slots** are generally days or hours
- The **availabilities** are: (*Mutual Exclusion*) Each resource can only be in one location at a time, and each location is visited by a resource during its available times. (*Capacity*) Each location is visited by a resource a particular number of times. (*Temporal*) Each sequence of locations visited by the each resource has an associated cost

First, there are similarities between the transportation and manpower allocation problems; *resources* are analogous to *employees*, as both can only be assigned to one *location* or *task* at the same time. However, while tasks require a particular number of employees at a given time, locations require a total number of resource visits over the entire schedule. Furthermore, the transportation problem has the added temporal constraint of measuring the cost of travel. The problem may be posed as a question of satisfaction — can the network be travelled within a total cost - or a question of optimization — what is the minimal cost for travelling the entire network. Another transportation problem may be seen in the scheduling of games for a professional sports league. Teams are scheduled into cities, and there must be exactly two teams scheduled to be in the same city at the same time. The goal is usually to reduce the amount of travel for each team. Additionally, constraints seen in the Manpower planning problem may also arise; teams may not be able to play on three consecutive days, for example.

3.2.3 Academic scheduling

The third and final type of timetabling problem is academic scheduling. There can be no doubt as to the prevalence of such problems; virtually every school requires a schedule of some kind. In the literature, there are two main types of academic scheduling tasks: exam scheduling and class-teacher assignment. We have already shown an instance of the general exam scheduling problem. This task, as mentioned, is seen mainly at University level institutions. Because universities offer such flexible programs for their students, with many elective courses, the students will have diverse sets of course selections. Because of the limited time period - usually two weeks - in which to schedule exams, the problem becomes increasingly difficult if there are a large number of students which need to be satisfied. Further, there may exist room capacity constraints, which state that the number of students writing an exam may not exceed the number of seats in the room in which it is being written. Under this formulation, the exam scheduling problem is not unlike the graph coloring problem. In fact, graph coloring techniques have often been applied to solve this problem[3].

The second type of academic scheduling problem lies at the centre of this research — generating a school's master timetable. There are many different formulations of this problem, as schools throughout the world have different scheduling needs. These "needs", or scheduling goals, will include some or all of the following:

i. assigning teachers to classes

ii. assigning teachers to time slots (creating teacher's schedules)

iii. assigning students to classes

iv. assigning students to time slots (creating student's schedules)

v. assigning classes to time slots (creating a master timetable)

vi. assigning classes to rooms

where a "class" is defined as a group of students who meet at particular times of the week for their lessons in a particular subject. Now, there are few real world school scheduling problems which would require the satisfaction of *all* of the goals (*i*) through (*vi*). A particular school may only require satisfying goals (*i*) and (*vi*), for example, while the other four goals may be trivial or irrelevant. We may identify three general categories of timetabling problems, primary schools, universities and high schools. Primary schools usually conform to a "home-room" model; that is, the students remain together as a class throughout the day. So, the only scheduling task that exists may be (*i*), assigning teachers to classes, (*ii*), assigning teachers to time slots, and (*iii*), assigning students to classes. Additionally, in crowded schools goal(*vi*) may also be of concern. Universities can often be more simple. The master schedule is determined before the students select their courses, and is based on projected demand. Then, students are left to themselves to register in courses at particular times in a conflict free manner. Professors usually only teach between one and three courses per semester, so creating their schedules is trivial. Therefore, the only problems that remain are (*v*) and (*vi*). High School problems are the most varied. Some high schools also use the "home-room" model, while others let their students freely register in the courses they wish. In the latter, more difficult version of the problem, the students supply the administration with a list of courses they wish to take, and, provided they are eligible to register in these courses, the students' course demands must be satisfied. Teachers are usually specifically trained in one area of expertise, such as Chemistry and Music, so they must also be assigned to classes. A high school timetabling problem (HSTTP) would usually be a combination of goals (*i*), (*iv*), and (*vi*).

We can describe the general HSTTP problem as:

- The **participants** are generally teachers and classes (or classrooms)
- The **time slots** are lessons, approximately 1 hour in length.

The availabilities will always include:

- no teacher may teach two lessons at the same time, or be required to be in two different rooms at the same time. There is one and only one teacher required for every course.
- no student may receive two lessons at the same time, or be required to be in two different rooms at the same time.

Next, there are other constraints which may or may not be relevant:

a. teachers have a minimum and maximum number of hours of instruction per week

b. teachers can only teach certain classes

c. teachers have preferred working hours

d. certain classes need to be assigned certain types of class rooms

e. rooms have student capacities

f. classes have student capacities

g. classes need to be scheduled at particular times

h. classes may need to be scheduled in a certain sequence

i. the school has a finite number of classrooms

j. students prefer to take lessons at a particular time or semester

k. students may receive a minimum and maximum number of hours per week of instruction

l. students may have preferences on which teacher they receive instruction from

Constraint (*d*), for example, may refer to a fact like physical education must be scheduled in the gymnasium. As well, note the difference between constraints (*e*) and (*f*). While constraint (*e*) refers to a physical limit on the number of students who can be in one room at the same time, (*f*) is some number set by the administration. For example, a special work experience class may require that at most five students be in any one lesson at a time. For constraint (*g*), classes that are scheduled at particular times may refer to, say, being only in the morning, being only on a specific day, or being scheduled only in a particular semester. Constraint (*h*) may occur if, for example, a course such as Biology is required to have its Laboratory scheduled immediately afterwards. Constraint (*l*), as remarkable as it sounds, actually is taken into account at one local high school. The students there, as a reward for good attendance, are given the opportunity to select their individual teachers for the next year's courses. Again, we would certainly not expect to find a real world problem which exhibited *all* of the constraints *a* through *l*, only a particular subset may apply. As will be seen in the review of previous work, using different combinations of the goals and constraints renders a wide array of problem formulations and necessary solution methods.

3.3 Major approaches to timetabling

3.3.1 Integer Linear Programming

The Operations Research community was the first to investigate solving complex scheduling problems, and other such combinatorial problems. Their approach is a quantitative one. The idea is to represent the constraints into a system of linear equations of several variables, a technique known as integer linear programming. The idea is that a Boolean proposition, such as ($\neg x_1$ or $x_2 \rightarrow x_3$), could be expressed equivalently as the equation $x_1 + (1 - x_2) + x_3 \geq 1$, where the Boolean values true and false are denoted by 1 and 0 respectively. Therefore, solving a complete scheduling problem involves solving a system of possibly several hundreds of linear equations such as this. Additionally, capacity constraints can also be simply represented by the same idea. If there is a shift where 6 of our 10 craftsmen must be working, for example, the linear equation is $c_1 + c_2 \ldots + c_{10} = 6$, where any c_i is a Boolean values (1 for true, 0 for false) stating if craftsman c_i is working on that shift. Integer linear programming has the advantage of allowing these optimization problems to be expressed exactly. This approach also allows for the incorporation of *weighted* constraints quite easily. Consider the general mathematical model of the class-teacher timetabling problem presented by de Werra:

We are given a collection of q courses $K_1 \ldots K_q$; course K_i consists of k_i lectures of one period each. The total number of periods is p. The students are divided into r groups $S_1, \ldots S_r$ such that in each S_L all students take exactly the same courses; L_k is the maximum number of lectures which can be scheduled at period k - i.e. the number of available classrooms. We define $y_{ik} = 1$ if a lecture of course K_i is scheduled at period k and $y_{ik} = 0$ otherwise. We let C_{ik} be an objective function of assigning course K_i to period k, where a high value indicates a high preference for $y_{ik} = 1$, and a low or 0 value for C_{ik}value corresponds to a preference for $y_{ik} = 0$. The problem that remains, then, is to maximize the overall quality of the solution:

$$\sum_{i=1}^{q} \sum_{k=1}^{p} C_{ik} y_{ik}$$

such that:

$$\sum_{k=1}^{p} y_{ik} = k_i (i = 1, \ldots, q),$$

$$\sum_{i=1}^{q} y_{ik} \leq L_k (k = 1, \ldots, p),$$

$$\sum_{i \in S_L} y_{ik} \leq 1 (l = 1, \ldots, r; k = 1, \ldots, p).$$

This example shows how ILP reduces a concrete real world problem to an abstract. mathematical problem of solving equations. For any reasonably sized school, this problem would require extensive computational time, however the solution that is returned is guaranteed to be optimal. The main shortcoming of this technique, and of Integer linear programming in general, is that in the event the constraints are too strong, there may not be any solution at all. In this case, nothing of value is returned to the user. This problem can be avoided by transforming some constraints into preference constraints, and incorporating them into the cost function; but in this case finding a solution may take far too much time. [1]

3.3.2 Reduction to graph-coloring

Since there already exists several methods for solving the graph-coloring problem, many theorists have found it appealing to apply these ideas to timetabling. A graph-coloring problem is a classical NP-complete problem wherein a graph G is an associated set of vertices V, and a set edges E between various vertices. The problem is to *color* the vertices such that no two vertices can be the same color if there is an edge between them. An early attempt at applying graph coloring to timetabling was by Neufeld and Tartar [10]. They proposed a model in which each vertex of the graph represented lecture, and an edge occurred between lectures which could not be scheduled at the same time. By assigning a period to each color, their timetabling problem was completely specified. However, in practice their model is not applicable to actual timetabling problems. The graph coloring paradigm is too simplistic to convey all of the nuances of a real timetabling problem.

3.3.3 Constraint Logic Programming

The more recent approach to formulating scheduling problems is seen in the constraint-based methodologies developed in the AI regime. This area includes CLP (constraint logic programming) and CSP (constraint satisfaction problems). Like the OR approach of timetabling, one is allowed precise, mathematical definitions of the constraints which define the problem being solved. These constraints may be symbolic in nature, asserted as *predicates*, such as *before*, *during*, and so on. The main advantage to these techniques is the distinction made between the formulation of the problem, and the algorithms and heuristics which solve the problem. Therefore, given one particular formulation of constraints, we are still free to choose any type of search algorithm to manipulate these constraints. This flexibility has lead to the design of extensible scheduling systems, which attempt to efficiently model a wide range of real-world problems. The PROLOG language, based on the principles of *unification* and *resolution*, was the first such system. Constraint logic programming can often find a solution faster than ILP. Smith et al. [16] have identified three main reasons why this may be the case. First, there is the idea of constraint *propagation*. Once one variable has been assigned a value from its domain, the effect of this assignment can be propagated throughout the network (forward-checking) to determine its effects on other domains. Further to propagation, constraint programming exceeds ILP in that it can better capitalize on heuristics during the search process. Because the solution to a CSP is built on an iterative process, solving each variable at a time and backtracking when stuck, a heuristic can often point the search in the best direction. Finally, Smith notes that CSP representations are often more compact than their ILP counterparts. CSP models are *declarative*, allowing for an expressive and flexible formulation. The symbolic encoding of the constraints means that a particular problem representation often needs fewer variables and constraints than an Integer Linear Programming formulation of the same problem. Furthermore, CLP languages are not restricted to using backtracking as a search method. Most such languages also have branch and bound available for partial constraint satisfaction problems.

3.3.4 Hierarchical Constraint Logic Programming

HCLP is an extension of general constraint logic programming formalized by Wilson and Borning [2] which allows for the expression of constraint *preferences* as well as strict requirements. The authors propose a generalized software implementation which allows the user to determine the relative importance of constraints, known as a *hierarchy*. In addition to scheduling problems, their work also allows for the simulation of such things as electrical circuits, mechanical linkages, and graphical calculators. The user may specify ranges of acceptable values for the variables in the problem, and the constraint preference level as being required, strong, medium or weak. There also exist comparators, which are functions that evaluate the quality of solutions, using a least-squares or weighted sum method. There can be one, global comparator or local comparators applied to various levels of the hierarchy. Therefore, an optimal solution depends on the structure of the hierarchy of constraints and functions used for their evaluation. The internal search process used by HCLP is backtracking guided by the heuristics of satisfying required constraints first, then strong constraints, and so forth. The main contribution of HCLP is the numerical, weighted handling of preference constraints in an efficient manner. However, expressing a high school timetabling problem such as our own would be extremely time consuming, since each of the thousands of constraints would need to be explicitly hard-coded.

3.4 Previous Work

The thrust of our research lies in representing school timetabling problems as a Constraint Satisfaction problem. An important step is to consider the previous work done in this area. However, formulating comparisons is difficult, because of the wide range of problems being solved. Even though there has been considerable work on school timetabling, schools' differing needs have led to many different representations and solution methods.

3.4.1 An initial approach

Meisels, Ell-sana' Gudes [9] looked at a heuristic approach to a particular high school system. The model they worked with was akin to the more simplistic "home-room" timetabling problem, however. In their problem, each class of students remain together for the entire term. Although different classes may take different subjects, it is known in advance which students would assigned to which particular class. It is also known in advance the teachers that will be teaching which particular classes. So the problem that remains is creating a master schedule which has teachers assigned to classes (goal i, as described in the previously section), such that no teacher has to teach more than one class during a time slot, and no class has to be taught more than one lesson in a time slot. From this information, a constraint satisfaction problem view can be quite easily seen. The variables of the CSP are defined by the authors to be teacher-class pairs. The domains of the variables will be the time-slots of the week. Each of the five days of the school week are divided into m hour long periods, and so the domain sizes will be $m \times 5$. The constraints will be binary between the teacher-class variables. Specifically, the constraints are of the mutual exclusion form, occurring between any two variables which share a class or a teacher. The mutual exclusion constraint simply states that the two variables may not take on the same value. Further, there exist unary constraints on the variables which represent teachers' restricted hours. Some part time teachers may only teach in the afternoon, for example, and any variable with a part-time teacher would have the morning time-slots removed from its domain.

Once the problem has been expressed in this manner, a divide and assign decomposition technique can be applied. The authors show that by reducing the problem to smaller sub-instances, a solution can be found with minimal effort, since consistent solutions form a large fraction of the search space. However, this divide and assign techniques is made possible by the fact that there are no non-binary constraints in the problem.

In all, this CSP representation is successful. The problem naturally lends itself to having binary mutual exclusion constraints, which are simple for the search algorithm to work with. However, clearly this constraint alone does not sufficiently represent a real world high school timetabling. The given problem makes use of much predetermined, implicit knowledge. For example, knowing which courses the students will take, and that each class remains together as a group, completely eliminates the task of scheduling individual students. Further, the problem does not make any mention of global, non-binary constraints on the variables, which are indeed present in our own local problem. For example, it may be that there are a finite number of rooms of a certain type, such as one gymnasium for physical education classes. As such, there would have to be an n-ary constraint, where n is the number of variables, ensuring that no two class-period variables have physical education at the same time. The existence of non-binary constraints makes the problem more difficult to express as a CSP. Thus, this particular CSP representation is adequate for a simplified high school timetabling problem, and gives us an excellent first look at how the CSP can be used, but unfortunately does not consider enough "real-world" constraints.

3.4.2 A CSP Representation with Flexible Constraints

Feldman and Golumbic [6] consider the use priority constraints in their CSP representation. The researchers are strictly looking at generating schedules for University students, based on their course preferences. Input to the program consists of a list of courses that the student is willing to take, and a lower and upper bounds on the number of courses willing to be taken. This would imply that only some, or possibly all, of the student's list of requested courses be included in the schedule. The student also supplies, if desired, a list of time constraints. These constraints may be times during the day classes may or may not be scheduled or entire days in which classes must not be scheduled. As well, the student may specify a range of hours for each day in which classes may be scheduled. The output from the program should be a schedule which best meets all of the student's wishes. The master timetable of course offerings and professor's timetables have already been created by hand and are known by the program. Where this research is significant is in the observation that in a practical scheduling system, it is generally the case that some constraints must be satisfied while others can be relaxed. This fact implies that if at first no solution can be found, then particular time constraints will be omitted until an optimal solution can be found. (Referring back to the list of scheduling goals in the previous section, this problem is a combination of goals *iii* an *iv*, with constraints of type *j* and *k*).

The representation of the problem as a CSP is as follows. Variables are represented by each course included on the student's list of requests. The domains of the variables are all of the offerings of the particular course, as given by the master timetable. Since it may turn out that not all of the courses requested will be scheduled, this CSP has the unique property of having variables which may not be instantiated. As such, a null "dummy" value is added to each of the domains, representing a non-scheduled course. There are many constraints, the first of which is the binary mutual exclusion constraint between courses. As seen in the previous representation, this constraint models the fact that no two courses may be scheduled at the same time. Second, there are unary constraints on the variables, corresponding to the student's preferred hours of instruction choices. Any hour of the day which is deemed undesirable is removed from all of the domains. The same effect occurs for the range of hours specified; those hours outside of the range are also omitted from the domains.

The upper and lower bounds on the number of courses are not as easy to facilitate. Since they apply to all of the variables, the effect is a non-binary constraint. From here, the authors proceeded to experiment on actual student data an test different search strategies. Regular backtracking, various forward checking and hill-climbing strategies were measured against their running times and how often an optimal solution - one meeting all of the student's requests - was generated. Not surprisingly, a tradeoff between speed and optimality existed, where word-wise forward checking was the fastest on average, and a heuristic ordering of the courses proved to be the most optimal.

In comparing this research with that of Meisels et al, it is clear that both successfully model a timetabling problem. Both strategies do, of course, begin with a great deal of knowledge handed to them. The decomposition technique of [9] had the luxury of knowing that the students did not need to be considered as individuals, only as a single group known as the class. The student scheduling problem described in this section makes use of the given master schedule, and the teacher's schedule isn't relevant. A further similarity is that both employ unary constraints to reduce domains, and use binary mutual exclusion constraints to ensure classes aren't scheduled at the same time. The methods differ in that the student scheduling method makes use of constraint relaxation, as well as non-binary time constraints. A further difference is in the scale of the problem — the decomposition method of [9] schedules a large number of teachers and classes while the CSP described in this section will only generate a timetable for one student. Yet, neither of these two methods will adequately solve high school timetabling common to North America — one that requires scheduling of both students and teachers.

3.4.3 A Constraint Relaxation Approach

Yoshikawa et. al. [5] have described a general purpose Constraint Relaxation Problem solver, known as COASTOOL. Their research focussed on solving extremely complicated Japanese High School scheduling problems, which include 30 classes (a defined group of students),60 teachers and 34 time-slots during a week. However, instead of using the CSP formulation discussed in the previously cited works, this research uses a CRP, or Constraint Relaxation Problem method. A CRP is the same as a CSP, in that both have variables, domains and constraints; the only difference is that some constraints have penalty values associated with them, and the goal is to minimize the total penalty due to inconsistencies.

The CRP formulation of the problem is as follows. Each variable represents a lesson and its domain is the set of all time-slots during a week. There are a number of real world constraints modelled under this representation. First, there are mutual exclusion constraints, meant to enforce that no teacher and no class is scheduled to be at different lessons at the same time. Further, there exist unary constraints which model teachers' schedules. Many of the teachers are preferred to work on certain days, or at certain times of the day, and as well the schools prefer that many lessons are to be taught at particular times of the day. So the unary constraints will rule out particular time-slot values from the domains of the constrained variables. Furthermore, there is also a binary constraint that two lessons for a two credit subject should be taken two days or more apart by a class. All of these constraints are assigned penalty values, which may be manipulated until the most desirable master timetable is produced. The researchers claim that timetabling by hand requires 100 man hours, whereas using their software requires only about a tenth of that.

It is the problem solving method of their CRP formulation which is the most intriguing. Theirs is a two stage process, using what they call a Really-Full-Lookahead Greedy (RFLG) algorithm. First, an initialization of all of the variables is made, using an arc-consistency algorithm. The arc-consistency removes values that cannot satisfy a constraint from the domains of the variables. In this first stage of the algorithm, an inconsistent assignment of variables is never made. If arc-consistency eliminates all of the values of its domain, then the variable will remain uninstantiated. The authors note that without this guarantee, a poor initialization could result in highly undesirable exponential computational time. The second stage of the algorithm will utilize typical Hill-Climbing techniques to assign values to the remaining variables, such that the total penalty of violated constraints is minimized. This second stage is actually Minton's Minimum-Conflicts Hill Climbing algorithm (1992).

In all, the COASTOOL software proved to be invaluable to the high schools, as the number of man hours expended on timetabling was greatly reduced. However, the authors admitted that their work does not adequately model global (non-binary) constraints, and that some of the master timetables created needed to be modified before actual use. As again, it is apparent that this formulatization of the problem suffers from the drawbacks seen in the previous two methods. In North American high schools, students have a wide variety of subject selections, and it is not practical to simply group together students into common classes. A truly beneficial timetabling system would be one that considered not just teachers' individual needs, but every students' demands as well.

Chapter 4

Representation of a High School Timetabling Problem as a CSP

This research focusses on high-school timetabling. Of all of the school scheduling problems, high-school timetabling is the most difficult. Unlike primary schools, high students do not remain together in "home rooms", and instead each student attends a different set of courses during the week. University timetabling is made easier by separating the problem into assigning lectures to rooms, and then letting the students register individually for their courses. In general, the High School Timetabling Problem (HSTTP) consists of scheduling teachers and students to classes, such that all constraints hold (for example, no two teachers are scheduled to the same class and no student has two classes at the same time). In this chapter, we are concerned with the problem of representing this HSTTP as a Constraint Satisfaction Problem (CSP).

4.1 The data motivating the research

The high school timetabling problems that are the centre of the present investigation arise from schools in Edmonton, Alberta. It is a large problem ranging in size from 250 to 2200 students and having between 45 and 320 courses to timetable. A *course* is defined as a group of students who meet three times per week for instruction. There is one and only one teacher assigned to instruct each of these three lessons. A course identifier consists of the *subject* of the course and a number indicating the relative content level of the course. For example, Science 10 refers to the first science course students may take, while Science 20 would be for those who have completed Science 10. A less academic version of Science 10 would be Science 13, which in turn would be followed by Science 23. The students have much freedom in the courses they will take. Each student provides the school with a list of desired courses four months in advance of the school year. This list will include selections of the *core* subjects, such as English, Math, or Science, as well some *options*, like Creative Writing, Band or Drama. It is an absolute requirement that the students receive instruction in any *core* course they choose, providing they are eligible to register in the course. The students will need these core courses in order to gain admittance to most universities. Students have individualized schedules based on the courses that they select. There are 8 periods in each week, with each period divided into 3 non-overlapping time slots. Each day has 5 time slots. A sample blank timetable is shown in Figure 4.1.

	Mon.	Tue.	Wed.	Thr.	Fri.
9:00am	1	2	1	2	1
10:00am	3	4	3	3	2
11:00am	5	5	4	5	4
1:00pm	6	7	6	6	7
2:00pm	8	8	7	8	-

Figure 4.1: Sample blank timetable

To clarify; if a student is scheduled to have Chemistry 30 in period 1, he would attend this class every Monday, Wednesday and Friday at 9:00 am. This structure implies that a student may register in at most 8 different courses in a term, but may certainly register in fewer. Some courses may be in high demand. For example, if Math 20 has numerous students register for it, the course is divided into many sections, where each section refers to a specific group of students who will always meet at the same period for the lesson. The maximum number of students allowed in one section is thirty, so if 110 students register in Math 20, then 4 sections will be allocated for this course, with each ideally having roughly 27 students. Most courses have between 1 and 4 sections, but some, such as English 10, have as many as 13. Each section will then be assigned a period number of 1 through 8. Therefore, part of the goal of this timetabling problem is, given all of the student's requests, assign a period number from 1 to 8 to each section of each course, and assign the students to a section for each course chosen, such that no student is assigned to more than one section in the same period. The other piece of the puzzle is scheduling the teachers.

Each teacher is required, by contract, to teach 7 of the 8 periods per week. It is known beforehand which teachers are able to teach which subjects. So, an English teacher may teach any of English 10 through 33. Some teachers may teach many different subjects. At any rate, the result is that each full-time teacher will be scheduled to teach 7 periods per week, and each section must have a teacher. However, the actual process of determining the teachers' schedules is not a part of the problem at hand, since the administration of the high school prefers to do this themselves. Nonetheless, one of our goals is to have a final solution which *guarantees* that valid teachers' schedules can be generated from the resulting master timetable. A method for scheduling the teachers is shown at the conclusion of this chapter.

Additionally, the school has two separate semesters of instruction. The first semester lasts from September to January, and the second is from February to June. The local high schools offer two types of courses – half-credit and full-credit. Most of the "core" courses, such as Math and English are full-credit, while other "options", such as Accounting and Law are half-credit. A half-credit course is simply scheduled into a period in either the first or second semester. A full-credit course can be scheduled throughout both semesters, and would be taken in a certain period for the whole year. However, there is also the possibility that a full-credit course may be scheduled as a "double-block" in one semester or the other. For example, a student scheduled for the full-credit Science 10 in the first semester may attend this subject in periods 1 and 2. In general, the majority of the full-credit courses will be taken throughout the year, in both semesters; however, for educational reasons it may be better for students to completely learn one subject in the first semester before continuing in a different subject in the second semester. Furthermore, these "double-block" instances will always be scheduled in consecutive periods. In fact, the school's administration has decided that these courses must occupy one of 4 pairs of periods: Periods 1 and 2, 3 and 4, 5 and 6 or 7 and 8. So, there are two terms which need to be scheduled, but the terms are not independent since some courses are held in the same time slot throughout both. Students register for both terms in advance, and since a student may take up to 8 courses in a term, a student may select as many as 16 courses beforehand.

4.1.1 Formalized definition of the HSTTP

Returning our attention to the previous chapter, where we identified the six main goals of academic scheduling problems, the current problem is a combination of goals (iii),(iv) and (v). There is also the additional task of assuring that goal (i) is able to be met. Goal (vi), assigning classes to rooms, is also not relevant to our work, and is handled by the administration.

From our set of possible constraints seen in chapter 3, constraints (b), (f) and (i) are relevant in the scheduling process. Constraint (b) raises the issue that teachers are only available to teach certain subjects. For example, suppose there are exactly 3 Physics teachers. Constraint (b) would therefore assert that no more than 3 Physics courses can be scheduled in the same period, otherwise there would not be enough teachers for all of them. Constraint (f) imposes that no more than 30 students may be scheduled into one particular class. Constraint (i) tells us that we may not have more courses scheduled in any one period then there are rooms in the school.

Now, for this problem we are given:

- The school offers c courses, eg. Accounting 10, Biology 10...Zoology 39
- The school has n students
- The school has r total rooms
- Every course C_i, $1 \leq i \leq c$, has an associated subject S_i. For instance, Math 10 through Math 33, all belong to the subject, "Math". But, English 10-35, Creative Writing 30, and Comparative Literature 20 - 30 all belong to the subject "English".
- There are z different subjects offered by the school.
- Every course C_i, $1 \leq i \leq c$ is subdivided into j_i *sections* so that the sections may be referenced as C_{ij}. The course Accounting 10 may have five sections, for example. The period of the third section could be referenced by C_{13}, assuming Accounting 10 was the first course in the database.
- Each subject S_i, $1 \leq i \leq z$, has a number of available teachers T_i
- Each student submits a list of desired courses L_i for both terms, where each $L_i \subset \{C_1, \ldots, C_c\}$
- The school has 8 weekly periods

The formalization to the problem is:

- The **participants** in the problem are the sections of the c courses
- The **time slots** are numbered $1 \ldots 8$
- The **availabilities** are that: Each of the lists of course selections L_i, are able to be scheduled for each student, i.e. the student is not required to attend the same lecture at the same time

- The total number of sections of each subject S_i does not exceed T_i during any period
- The total number of sections does not exceed r during any period.

Additionally, we desire the n individual student timetables to be generated from the resulting assignment of periods to sections, such that each list of course selections L_i is satisfied. This goal is also subject to the constraint that no more than 30 students can be scheduled into one section. By no means does this problem fit any classical description of the types of timetabling problems discussed in Chapter 3. The main constraint, that the students must have the courses they have chosen available to them, is of the not-equals variety. However, there are also two distinct capacity constraint in this problem.

4.2 HSTTP as a CSP

In order to formulate the HSTTP problem as a Constraint Satisfaction Problem, we must define what are to be the variables, domains and constraints.

4.2.1 Variables

Each variable represents a course C_i, such as Math 10. Further, each course has a number of attributes associated with it:

- the number of sections j_i in that course
- the number of students enrolled in that course
- the capacity of each section in that course (typically 30 for all courses)
- the subject S_i of that course
- the semester type of the course which can be one of the five values shown in Figure 4.2

FIRST TERM, HALF CREDIT COURSE (H1)
FIRST TERM, FULL CREDIT COURSE (D1)
YEARLY FULL CREDIT COURSE (F)
SECOND TERM, HALF CREDIT COURSE (H2)
SECOND TERM, FULL CREDIT COURSE (D2)

Figure 4.2: Types of semestered courses.

4.2.2 Domains

The domains of the variables each consist of an m-tuple of periods, where m is the number of sections of each course. Each tuple consists of m periods in the range $\{ 1 \ldots 8 \}$. Therefore, in the general case a 3-sections course would have the domain values $< 1,1,1 >, < 1,1,2 >, < 1,1,3 >, \ldots$ and so on. The result of this choice is that domain sizes becomes an unmanageable 8^n, where m is the number of sections of the course. However, we may employ a simple technique to prune the domains. If we enforce that no period's value appears more than once in each permutation, then the domain values for the 3-section course become $< 1,2,3 >, < 1,2,4 >, < 1,2,5 >, \ldots < 6,7,8 >$. The domain size in this case would be 56, as opposed to the size of 512 which would exist under the complete representation. In effect, we are enforcing unary constraints that are overly strong. We acknowledge that this over-constraint *may have the consequence that no solution exists* when there may have been one otherwise. However, the tradeoff is that finding a solution should now be much easier. The natural heuristic of disallowing multiple sections in the same period reduces the size of the domains from 8^n to at most 70, and therefore the total search space is reduced exponentially. The maximum domain size of 70 occurs when the number of sections is four and the tuples are $< 1,2,3,4 >, < 1,2,3,5 >, \ldots < 5,6,7,8 >$. Additionally, this heuristic directly corresponds to what the actual high school schedulers do; only rarely will a course be "doubled up" in the same period. We believe that disallowing these occurances will diversify the sections, which should more quickly lead to acceptable solutions and avoid clashes of sections being scheduled at the same time.

Courses with large numbers of sections are also worth illustrating. A variable representing a course with 8 sections would have a domain size of 1, simply containing $< 1,2,3,4,5,6,7,8 >$. In the event that the course has more than 8 sections, some overlap is impossible to avoid. So, in this case we assume that the first 8 sections of the course have the implicit $< 1..8 >$ distribution, while the remaining sections obey the non-overlap rule. For example, a 10 sections course would have the domain values $< 1,2 >, < 1,3 >, \ldots < 7,8 >$, which actually correspond to $< 1,1,2,2,3,4 \ldots 8 >$ $, < 1,1,2,3,3,4 \ldots 8 >, \ldots < 1,2 \ldots 6,7,7,8,8 >$. In this case, the domain size is 24, the same as a 2 section course.

However, in the event the course is full credit but only in one term, (i.e. is of type D1 or D2 above), then the domain becomes a permutation of $\{1,3,5,7\}$. For these cases, it is assumed the course will be placed in a double block of 1-2, 3-4, 5-6 or 7-8, meeting with the administration's needs. The possible values of a 3-section course of this type would be $< 1,3,5 >, < 1,3,7 >, < 1,5,7 >$, and $< 3,5,7 >$.

For the first type of domain, semestered courses, we see that there are four distinct domain sizes, $8, 24, 56$ and 70. The size of 8 occurs when m mod 4 is 1, where m is the number of sections, and the size of 24 occurs when m mod 4 is 2, and so on. Courses which have 8 sections exactly are automatically assigned the trivial value $< 1,2,3,4,5,6,7,8 >$. For the "double-block" variables, the domain sizes are 4, 6, 4 and 1, corresponding to courses with $1 \ldots 4$ periods; the size of 4 is induced when m mod 4 is 1, 6 when m mod 4 is 2, and so on.

4.2.3 Constraints

Student course selection constraints

The given input to the problem consists of the student course selections. The resulting final timetable must be one that somehow has the necessary available sections open for all of the student requests. If we wanted to represent the scheduling needs *exactly*, we would have a non-binary constraint for each set of student course requests. Such a non-binary constraint would enforce that there are the available sections open for a student to enroll in all of the chosen courses. For example, suppose the following. a student decides to register in Accounting, Biology and Computer Science; Accounting is a half-credit course offered in the first term, and has 1 section; Biology is double-block course offered in the first term, and has 1 section; and Computing Science is a full year course (takes place over both terms) and has 2 sections. Now, a 3-ary constraint can be employed which guarantees that the available sections are open in these courses, for this student. This constraint would deem, for instance, that if the permutations $< 1 >, < 1, 2 >$ and $< 5, 7 >$ were assigned to Accounting, Biology and Computing Science respectively, this would be unacceptable, since the student would be *forced* to attend Accounting and Biology in period 1. (Note that since Biology is a double-block course, the student attends it in periods 1 and 2). On the other hand, the permutations $< 4 >, < 7, 8 >, < 4, 5 >$ assigned to the three courses would be acceptable. The student would take Accounting in period 4, Biology in periods 7 and 8, and Computing Science in period 5. (Note that Accounting and Biology would be taken only in the first term. while Computing Science is taken in *both* terms in period 5).

We can see that the non-binary constraint responsible for enforcing section availability is complex. The problem quickly becomes over constrained under this formulation. Virtually every course (variable) participates in a non-binary constraint with many other variables. To illustrate, consider a popular course such as English 10. If three hundred students register in this course, then the variable for English 10 becomes constrained with every other course being taken by each of the three hundred students. Thus, when the English 10 variable is assigned a value, all of the three hundred associated constraints must be verified for consistency. If, at some point in the search, this English course has been assigned an incompatible value with the rest of the problem, then backtracking to this variable and changing its value means that 300 constraints need to be re-verified. Changing the value of this variable could mean that other constraints are no longer valid, and the effects of one variable change are propagated to many other variables.

We have developed a set of binary constraints which will *estimate* the section assignment needs. It would not be possible to transform the more exact non-binary constraints directly into binary constraints, thus, we are motivated to find some form of estimation of the exact needs of the student requests. We identify two kinds of binary constraints — *subset* and *intersection* constraints. The subset constraints occur between courses in the same term. This constraint between two courses says that one course's permutation of period values cannot be a subset of the other. The idea of this constraint is to avoid a student being left with the same period as the only open time for two courses that are registered for. For example, suppose a student chooses three courses, A, B and C, the first two having two sections, and the latter with one section. If both A and B are given the permutation $< 4, 7 >$, one might conclude that this was fine, the student could take A in period 4, and B in period 7. However, if C were now given the value $< 4 >$, this is not acceptable. Thus, between *one* pair of these courses there is a subset constraint necessary, which would deem that the two courses' permutation may not be equal to or a subset of the other. With just one subset constraint, together with the natural heuristic of avoiding duplicate courses, we have now guaranteed that the assignment of periods to sections for these three courses will satisfy the student.

If this subset constraint is thought of as a list of valid pairs of values between variables, we may explicitly define the constraint for a number of different cases, i.e., if a subset constraint exists between courses D and E, and D is a 3 section course and E has 2 sections, then the non-permissable values would be:

Course D	Course E
$< 1,2,3 >$	$< 1,2 >$
$< 1,2,3 >$	$< 1,3 >$
$< 1,2,3 >$	$< 2,3 >$
$< 1,2,4 >$	$< 1,2 >$
...	...
$< 6,7,8 >$	$< 7,8 >$

Now suppose course A is a full credit course occurring only in the first or second terms (i.e. is of type F1 [illegible] F2). Course A now takes on values $< 1,3,5 >< 1,3,7 >$ ..., but the sections themselves are actually being scheduled in $< 1,2,3,4,5,6 >$, $< 1,2,3,4,7,8 >$, ... So the non-permissable values would be:

Course A	Course B
$< 1,3,5 >$	$< 1,2 >$
$< 1,3,5 >$	$< 1,3 >$
$< 1,3,5 >$	$< 1,4 >$
...	...

The second kind of mandatory constraint is the intersection constraint. These constraints arise between courses which are taken in *different* terms. These constraints say that one permutation of sections must include at least one similar value with the other permutation. For example, $< 1,2,3 >$ intersects with $< 2,7,8 >$, while $< 1,2,3 >$ does not intersect with $< 6,7,8 >$. While the idea behind the subset constraint was to force flexibility in the section assignments, the intersection constraint works the opposite way. A simple example can demonstrate why this constraint is necessary:

Suppose a student picks 7 full-year courses, one half credit course in the first term, and one half credit course in the second term. The student's timetable will consist of the full 8 courses in each term. Now assume that all of the full year courses have 8 sections and are trivially given a permutation which has one section in each period.

Suppose both of the two half credit courses only have one section each. For the full year courses, we observe that the student must be assigned to sections in 7 different periods. Recall that full-year courses are taken in the *same* time-slot in each of the two terms. Therefore, the two half credit courses *must* be scheduled in the same period, since this period is the only empty slot in both terms, for this student. The intersection constraint enforces this requirement.

Having identified the nature of the two types of binary constraints, subset and intersection, we may now proceed with identifying exactly when these constraints are applied.

First, we will illustrate the cases where it is *mandatory* that the binary constraints exist. In these examples, if the constraints are violated, a student *will* be unable to attend some of the enrolled courses. The rules for identifying mandatory binary constraints are as follows:

A student picks a set of v courses, $C_1 \ldots C_v$.

- Rule 1. Consider each pair (C_i , C_j) of courses. If C_i and C_j both have one section, and both occur in the same term, (i.e. both courses are of type H1, D1, or F; or both courses are of type H2, D2, or F), then a mandatory subset constraint exists between C_i and C_j.

Rule 1 simply states that if two courses chosen by a student have 1 section, and both are taken in the same term, then these courses must *not* be scheduled in the same period. If they were, the student could not attend them both.

- Rule 2a. Suppose the set of courses $C_1 \ldots C_v$ fits the template { F,F,F,F,F,F,F, H1,H2 }, that is, the student has picked 7 courses of type F (full-year), 1 course of type H1 (half-year, first term only), and 1 of type H2. Then, a mandatory intersection constraint exists between the H1 and the H2 courses.

Rule 2a says that the two courses of type H1 and H2 *must* occur at the same time. Since the seven full year courses occupy the same period on the student's schedule for both terms, there must be only one period open on the schedule, which is the same period in each term.

- Rule 2b. Suppose the set of courses $C_1 \ldots C_v$ fits the template { F,F,F,F,F,F, D1,D2 }. A mandatory intersection constraint exists between the D1 and the D2 courses.

- Rule 3. Suppose the set of courses $C_1 \ldots C_v$ fits the template { F,F,F,F,F,F, H1,H1,D2 } Two mandatory intersection constraints are needed, one between the first H1 and the D2, and one between the second H1 and the D2. Rule 1 may or may not have already been needed to create a constraint between the two courses of type H1.

- Rule 3b. For the template { F,F,F,F,F,H2,H2,D1 }, two mandatory constraints are needed, one for each H2 and D1 combination.

These rules illustrate some clearly defined cases where the constraints must be enforced, and the rules tell us exactly which courses need to be constrained. Further to these mandatory constraints, we also identify *estimator* constraints. In the following examples, we show when these constraints need to be applied. However, unlike the mandatory constraints, it is not clear which exact courses need to be constrained.

Consider the following rule:

> **Estimator Rule:** If a student selects a course of n sections, and also selects d courses (in the same term) with n or fewer sections, $d \geq n$, the n-section course is subset-constrained by $d - n + 1$ of the other courses.

This rules tells us the situations where binary constraints are necessary.

- Example: A student picks 3 full year courses, A, B, and C. A has 1 section, B has 2, C has 3. No binary constraints are needed. Any combination of values for A,B and C will allow the student to attend all three courses.

- Example: A student picks 5 full year courses, A through E. A, B, and C each have 3 sections, D has 2. E has 1. Subset constraints are needed between A-B, A-C and B-C. These will guarantee that 5 different periods will appear in the permutations of A, B and C. D and E could then be anything, and are not constrained.

- Example: A student picks 16 half year courses, 8 in each term. Each course has one section. A constraint would exist between each course in the first term, and each in the second term. (A total of 56 constraints in all).

Estimator constraints never exist between half-credit courses occuring in different terms. When two courses are potentially going to be constrained, their course types(a value H1 through F discussed above) must be considered. Constraints can only exist between courses which are of the same type, *or* in the cases where there is some overlap in time between the two courses.

Now, suppose a student's list of choices look like this:

Course	Type	Sections
A	H1	1
B	H1	4
C	D1	1
D	F	4
E	F	3
F	F	15
G	F	2
H	D2	2
I	D2	4

The student has picked a difficult set of selections. He will have the maximum 8 periods occupied in each term. What are the estimator constraints?

First of all, when dealing with these "double-block" courses of type (*ii*) and (*v*), their number of sections is thought of as being twice the actual number, since two periods will actually be occupied. So course C really has 2 sections, H has 4 and I has 8.

Consider the first term (courses A through G).

Course	Sections
F	15
B	4
D	4
E	3
C	2
G	2
A	1

Course F can never be constrained with anything, as it has size > 8. Now, course B has 4 sections. There are 5 other courses with 4 or fewer sections. So in the Estimator Rule, $n = 4, d = 5$, and there are $5 - 4 + 1$ constraints needed on B. These are applied to D and E. The same rule is applied to course D, and it is determined that it must be constrained to courses B and E. So our set of constraints so far is { B-D, B-E, and D-E }. The rule is applied to course E, and it is determined that it must be constrained with course C. Finally, the Estimator Rule tells us that C must be constrained with G. So the constraints are { B-D, B-E, D-E, E-C, C-G}. As for the second term:

Course	Sections
F	15
I	8
D	4
H	4
E	3
G	2

Courses I and F will never be constrained. Furthermore, D has 4 sections, but there are only 3 other courses with 4 or fewer sections. So D, and H by the same logic, will not be constrained. E has 3 sections, but there is only 1 other course with 3 or fewer sections, so E is not constrained. G will not be constrained either. So no new constraints are added when the second term is considered, and the overall set of constraints remains { B-D, B-E, D-E, E-C, C-G}. The motivation for using these estimator constraints is that they will diversify the scheduled periods and, as experiments show, are much more likely to result in student satisfaction. Ultimately, though, there is some luck involved, since there is no guarantee that when all of the constraints are amassed, a solution still remains. Furthermore, we are disregarding the real world constraint of at most 30 students per section, and simply "hoping" that our estimator constraints will appropriately diversify the courses. As we shall see in the next two chapters, it would seem through the use of iteratively adding the binary constraints, satisfactory solutions can indeed be found.

Non-Binary Constraints

The non-binary constraints are never meant to be estimations, instead they are exact. All of the non-binary constraints are included in any solution attempt. There exists a non-binary teacher's constraint, which is designed to ensure the final solution will still allow for successful scheduling of the teachers. There is one such constraint for each of the twelve subjects, and the constraint covers all of the courses in each subject.

If there are f full-time teachers for a given subject, this constraint says that there may not be more than f courses of the subject scheduled at one particular period.

Suppose we have three Chemistry teachers, and six chemistry courses. Let's say these six courses have 5 sections, 4 sections, 4 sections, 3 sections and 1 section respectively.

Thus, if these 5 Chemistry courses took on the values:

- Chemistry 10: $\{1,2,3,4,8\}$
- Chemistry 13: $\{1,3,5,7\}$
- Chemistry 20: $\{1,4,6,8\}$
- Chemistry 25: $\{2,5,7,8\}$

- Chemistry 30: {5}

Then the non-binary constraint is satisfied, because no more than 3 Chemistry courses are scheduled at one time.

Does this guarantee that the teachers are scheduable? The local constraint is that teachers must be working for seven of the eight periods. Note that if the number of sections in a subject is not a multiple of seven, then there will be excess periods which are made up by a part-time teacher (or a teacher of a different subject). There is a simple argument which proves that all of the (full-time) teachers will be able to teach seven periods, conflict free. Think of the number of sections of a subject in each period being summed. In the above example for Chemistry, we can total the number of sections in the 8 periods as:

$$3, 3, 3, 3, 3, 1, 2, 3$$

This subject has three teachers, and since our non-binary constraint is met, no more than three lectures take place at a time. Now, to schedule the teachers in a conflict free manner, simply look at the totals for each period and have each teacher assigned to the seven periods with the most sections currently available, and then decrement these seven totals by 1. Thus, for Chemistry the first teacher would be assigned to work in all of the periods except period 6, (which has the fewest courses available, 1). The totals for the sections in each period would now be:

$$2, 2, 2, 2, 2, 1, 1, 2$$

The next teacher could now choose to have either period six or seven as their "free" period. If this teacher did not want to teach in period seven, the totals are reduced to:

$$1, 1, 1, 1, 1, 1, 0, 1$$

Now, the last teacher gets the remaining courses. This routine for assigning sections to teachers is guaranteed to be conflict-free, *if* the non-binary constraint is met, i.e. in general, if there are k teachers then no period has more than k sections. The periods with exactly k sections are assigned to all teachers, while the other periods are decreased incrementally. At the ith step, there can not be any periods with more than $k - i$ sections still available. Therefore, the routine must end up with seven 1's after the $(k - i + 1)$th step. There will be exactly k steps in the routine since there are k teachers. In the case where the number of sections is not a multiple seven, the routine ends with less than seven 1's, and so these are the periods assigned to a part time teacher.

The other non-binary constraint is the global room constraint. This constraint is meant to enforce that the school cannot exceed its' capacity. If there are r rooms, then there may not be more than r courses scheduled during one particular period.

We have concluded our discussion on this CSP representation of the given HSTTP. However, are there other CSP formulations that arise from the same problem?

4.2.4 An Alternative CSP formulation

- **Variables** represent individual sections of courses.
- The **domains** are simply the period values $1 \ldots 8$.
- Suppose a student chooses courses A and B. A **binary constraint** exists. between one section (variable) of each of A and B
- The same **non-binary constraints** from the original representation would exist

Under this scheme, there are far more variables present in the problem — close to 1000 for a medium sized school. However, the domain sizes are reduced to being all of size 8, or 4 for the double-block case. But notice that we are no longer able to employ the unary constraint of forcing sections of the same course to have different values. In turn, without the unary constraints, we will need **more** binary constraints to enforce section availability for students. For example, if a student picks two courses, each of which have 8 sections, there would be no constraint needed under the original scheme.

However, since it may turn out that all sixteen sections end up being scheduled in same period, we would need a binary constraint between one section of each course to guarantee the student may attend both courses. Since we will need one constraint for *ery* pair of courses chosen by every student, the number of constraints under this formulation will be enormous. As we shall see, the problem is over-constrained as it is, and many constraints will need to be relaxed. So, compared with the original CSP, this alternative one will have more variables, and more constraints, a much bigger search space without the unary constraints, and is therefore considered inferior.

Chapter 5

Solving the High School Timetabling Problem

5.1 Overview

Thus far we have discussed the nature of the scheduling problem and reviewed previous work in this area. We have also described an actual high school timetabling problem. Further, we have given this real world problem a representation as a Constraint Satisfaction Problem.

In this chapter we will examine the process of solving Constraint Satisfaction Problems based on three particular high school timetabling problems. The task at hand is to come up with an instantiation of all of the variables (courses) which satisfy all of the constraints. But what is to be done if such an instantiation does not exist, or is too costly to find? As we shall see, certain constraints will have to be eliminated from the problem.

Further, once all of the courses have been assigned periods, our task is not yet finished, since the students need to be scheduled into these classes. We then measure the success of our timetable by the number of students which are successfully scheduled.

5.2 Three local high schools

Three different high school timetabling problems have been experimented on. Although each of the problems involved the same types of constraints, the resulting CSP's were different in nature. The smallest of these schools, Old Scona Academic, has only 248 students, with 13 different subjects and 45 different courses. A second school, Ross Sheppard Composite, has 1661 students, 33 subjects and 218 courses. The largest of the three schools, Harry Ainley Composite, has 2236 students, 48 subjects and 320 courses. The actual process of scheduling students in these schools begins four months in advance of the beginning of the school year. In April, students select their courses for the term starting in September. At Harry Ainley and Ross Sheppard, the administration will then decide on matters such as the number of teachers for each subject based on the students' selections. The next action is to devise a master timetable, in conjunction with scheduling teachers and students within this timetable. This step requires several man-weeks of effort, and may not be completed until July or August. Allowances need to be made for students who decide to switch programs, or enroll in the school not until late in the summer, or students who enroll but simply never show up. Consequently, once the school year begins there may be additional timetable adjustments that need to be made.

High school timetablers strive for 100 percent satisfaction of student and teacher's timetables. The definition of "satisfaction" with respect to a student's timetable is that all of the requested courses are successfully scheduled within the 8 periods of the two terms. An unsatisfied student request is usually due to some unusual combination of "option" courses, or because of a set of courses which cover the three different grade levels. In the event that the timetabler cannot satisfy a student's course requests, there are three options. First, and most frequently, the student is asked by the administration to select different courses in lieu of those that cannot be scheduled. Second, if the scheduling conflict is due to a course being full, the administration may simply exceed the capacity of a course by registering additional students. Third, if too many students are not being scheduled for a certain course, the administration may open another section for that course. However, this action may be impossible or impractical, since another room and another teacher would be needed. If all of the teachers already have their schedules satisfied, adding another section of a course may require hiring a part-time teacher. At Old Scona, the student satisfaction rate is nearly always 100 percent. Ross Sheppard claims to have at worst 92 percent, and a Harry Ainley scheduler has quoted a similar figure for her school. It is worth noting that Old Scona does not make use of the "double-block" course, as described in the previous chapter, while the other two schools do. Scheduling the teachers is not a goal of our software, since the administration staff prefer to do this themselves. However, by use of the non-binary teacher's constraint, we ensure that scheduling the teachers is guaranteed to be possible.

5.3 Expressing the timetabling problems as a CSP

Let us summarize the constraints used in our CSP model as described in the previous chapter. Of the binary constraints, there are the not-subset type, which enforce that courses are to have sections at different times. There are also the intersection constraints, which assert that two courses in different terms have to be scheduled in the same period. There are two kinds of non-binary constraints: room and teacher constraints. A complete representation of the problem is one that includes these constraints, using the rules described earlier, on all courses. However including all of these binary constraints leads to an unsolvable, over-constrained CSP. A CSP with no solution is highly undesirable, since there is nothing of value returned to the scheduler. We need some additional rules which dictate the number of constraints to be used by the CSP solver routine.

5.4 The Algorithm

The general algorithm used to solve all school timetabling problems of the nature described in this work must be flexible. Even if all of the constraints are not enforced, a solution still may be returned which ultimately satisfies all of the student requests. The algorithm we have developed operates within several steps. Essentially, the CSP begins with no binary constraints at all, and some instantiation of the variables is found which satisfies the non-binary and unary constraints. Recall that the unary constraints enforced that no tuple of periods would have the same period more than once. This first step, solving with no binary constraints, is an improvement over a completely random assignment, since the non-binary constraints force the sections to be spread mostly evenly over the 8 periods in each term. We perform 10 trials of solving this under-constrained problem, each time randomizing the order of the domains of the variables. The randomization is truly a necessary step, since otherwise all of the initial variables (courses) would have sections assigned to periods 1, or $\{1,2\}$, or in general $\{1,2..m\}$ for an m-section course, until the global capacity of the school's rooms is met. This tactic would result in period 1 being filled immediately, while period 8 would be virtually unused. Consequently, much backtracking would be required to correct this misguided attempt. Working with the best solution found from the 10 trials, the CSP is "repaired" by adding binary constraints based on students who are not satisfied. The process begins again, except now instead of having no binary constraints we add in those constraints which pertain to courses with one section. This value, one section, may be thought of as a threshold. The threshold will be incremented as necessary until there is either 100 percent student satisfaction or no solution is found — and the algorithm halts. We will describe these steps further detail, but first let us summarize the general pseudocode of the algorithm in Figure 5.1.

We have chosen to use a backtracking routine instead of Branch and Bound. On a problem of this size, with this many constraints, Branch and Bound requires an impractical amount of time to arrive at a complete solution [4]. Furthermore, a Branch and Bound solution which minimizes the number of course constraint violations does *not* mean that the solution is optimal, since we evaluate the quality of our solutions by the total number of students who are able to scheduled. In fact, one course

```
TIMETABLING ALGORITHM
0.  determine enrollment matrix
1.  threshold.section ← 0;
2.  threshold.enroll ← 1;
3.  Repeat
4.          GenerateCSP(threshold)
5.          For i=1 to 10 Do
6.                  randomize domains
7.                  Order Variables
8.                  SolveCSP
9.                  Schedule Students
10.         Change thresholds
11. Until no solution exists
```

Figure 5.1: Generic High School timetabling algorithm

constraint violation may result in several students being unable to be scheduled.

Step 0. - Determine Enrollment matrix

The enrollment matrix is used to determine which pairs of courses are taken together by students. This information is then used by the *threshold.enroll* value to determine which courses are included as constraints in the CSP. The matrix is $n \times n$, where n is the number of variables (courses) in the problem. For every pair of courses occuring in a student's registration selections a counter value at the corresponding matrix element is increased; i.e. if a student selects Math 10 and English 10, then $EnrollMatrix(Math_{10}, Eng_{10})$ is incremented by 1. After all of the student requests are examined, the matrix values are then scaled by dividing each value by the total number of students registered in the smallest of the two courses. For example, suppose there are 200 students in Math 10 and 150 in English 10. Now suppose we count that 135 students have selected both Math 10 and English 10. The element $EnrollMatrix(Math_{10}, Eng_{10})$ becomes $135/150 = 0.9$. Therefore, we are left with a matrix of values between 0 and 1. High values indicate strong correlations between two courses, and therefore it is *more* important that these courses be constrained.

Low values indicate that it is less important that these courses be constrained. There are many 0's in this matrix; no student can take two different English courses for example, so all matrix elements between English courses are 0.

Step 4. GenerateCSP(threshold)

This routine is passed the two threshold values, section and enroll. The binary constraints are added to the CSP only if one of the variables being constrained has at most *threshold.section* sections; if it has exactly *threshold.section* sections then the corresponding *EnrollMatrix* element must be greater than or equal to *threshold.enroll*. This step serves to gradually increment the number of constraints in the problem, while hopefully including the constraints that are the most vital to satisfying student selections.

Step 5. for i = 1 to 10

Within this algorithm lies a loop that randomizes the domains and solves the CSP. The number of iterations, ten, was chosen as a means of reducing the chances of starting with poor orderings of the domains. The number ten is somewhat arbitrary, but does keep the total time of solving the largest school to within thirty minutes on a *Sun 3/60 Workstation*. Experiments show that choosing a larger number of test iterations increases the search time without any appreciable effect on the quality of the final solution.

Step 7. Order Variables

The order of the variables refers to the order in which the backtracking routine will assign values to the variables. A good heuristic ordering of the variables can greatly reduce the cost of finding a solution. [18]. In our work, he best ordering strategy that has been found is by domain size, smallest to largest. This means that variables with domain size 1 (an 8 section full year course or a 4 section double-block course) are assigned their values first, while the largest domain size of 70 (a 4 section full year course) will be assigned its value last.

Step 8. Solve CSP

This step involves most of the computational time. SolveCSP could be any existing CSP algorithm, but for our experiments is the backtracking algorithm called *forward – checking*. This particular algorithm is a considerable improvement over chronological backtracking. Experiments on several different problems [19] [17] have shown forward-checking to require the fewest consistency checks on average. Furthermore, many existing software packages such as the CHIP scheduler also use forward checking. Forward checking has been found to be easily adapted to the non-binary case [19]. The general forward checking algorithm works by deleting values from the domains of variables not yet assigned a value. For the non-binary constraints, we simply delete all elements of the domains containing the value of a counter which has equaled its capacity. For example, suppose we have 10 Math courses and 3 teachers. Now suppose we assign the first three Math courses the permutations of periods $< 1,2,4 >, < 1,4 >, < 4,8 >$. Now, period 4 has met its capacity of 3 Math courses. So, the next 7 Math courses will have all permutations which contain a 4 in them deleted from their domains.

For each solution attempt the number of consistency checks is recorded. If no solution is found, the program terminates, since too many constraints have been added. The solution found which satisfies the most students course selections is returned.

Step 9. Scheduling the students

After the CSP has been solved, and we are left with a master timetable, the individual students must be scheduled so that their course requests are met. We have determined that the ordering of the students to be scheduled can make a difference. We choose to order the *most* difficult to schedule students first; a “difficult” student to schedule is one who chooses many courses with few sections in them, thereby causing less flexibility in the student's timetable. If we leave a difficult to schedule student until last, it may turn out that one of the sections he has chosen is full (30 students), and there may not be any other choices. Therefore, we define the difficulty of a student's set of choices by summing the number of sections in each course he has selected, and then dividing by the number of courses chosen. The most difficult to satisfy student would be assigned a value of 1, while any student with a value of 6 or greater will likely be easily scheduled. The actual method chosen for scheduling the students is a simple greedy algorithm. Each student is assigned to the section of the course which currently has the *fewest* students already registered in it.

The number of successfully scheduled students is recorded. If this value exceeds the previous best, the solution is saved, and will be available once the program terminates.

Step 10. Change thresholds

The thresholds are manipulated in order to increase the number of constraints that are in the CSP, and hopefully yield a better solution. At this step the *threshold.enroll* value is decreased by 0.25. The value of 0.25 will roughly increase the number of relevant courses — i.e. those that can participate in a binary constraint — by 10%. If the value of *threshold.enroll* is reduced to 0, the *threshold.section* value is incremented by 1. This means that there will be 4 iterations of the main loop for each increment of the *threshold.section* variable.

5.5 Experimental Results

This algorithm effectively schedules more than 98 percent of the students in the two largest schools, Harry Ainley and Ross Sheppard, and 100 percent of the students in Old Scona. The largest school, Harry Ainley required the most computational time. The best solution was not found until the *threshold.section* variable equaled 4, and

school	students	satisfied	percent	t.section	t.enroll	cc's
Harry Ainley	2236	2199	98.35	4	0.25	179329
Ross Shep.	1661	1641	98.80	3	0.75	88923
Old Scona	248	248	100.00	2	0.75	14008

Figure 5.2: Experimental results on the high schools. The number of satisfied students is shown, along with the final values of the thresholds when the final solution was found. The consistency checks indicate the relative workload. All of these experiments were completed in under five minutes of run time.

the *threshold.enroll* variable was 0.25 . Beyond these values no solution was found. Let us summarize the results of these algorithms in terms of the number of students scheduled in the best case, along with the value of the threshold parameters when the best solution was found, and the number of consistency checks required for finding the successful solution. The results are shown in Figure 5.2.

One drawback with this algorithm is the time required when no solution exists. Once the "fine line" is crossed, and too many constraints exist, the amount of work to prove that no solution exists becomes unmanageable — several hours on a large school such as Harry Ainley. Since we have found that once a solution attempt exceeds one million consistency checks, the probability of the routine actually terminating in a solution approaches zero, we impose this halting limit.

5.5.1 Unsatisfied students

In the two large schools, roughly two percent of the students cannot be satisfied. The source of difficulty in student scheduling is either "clashing" times allocated for time slots, or a course having thirty or more students. The latter case actually occurs infrequently. Of the 37 students that were not scheduled at Harry Ainley, only eight were not scheduled because of sections being full; at Ross Sheppard the fraction was only three of twenty. At any rate, this problem is not too serious in reality. The school administration is willing to be flexible on the upper limit of thirty students in a room, if absolutely necessary. The issue of time clashes for courses is more difficult. Below is an example of a partial incompatible schedule for a student:

Course	Periods offered
Chemistry	1,5,6
Accounting	1,6
Drama	5
Spanish	6

In this example, the student will be unable to register in one of the four courses. It is required, by law, that students receive instruction in any *core* course in which they have registered. Chemistry is such a course, while the other three are *options*. The scheduling algorithm can ensure that registering in a core course always takes precedence over the options. Therefore, we have found that *all* of the students in the three schools have schedules which allow attendance in all of the desired core courses. For the options, the student may either take the course by correspondence, or simply register in a different course.

Thus far we have shown that our algorithm performs well on three schools' data. In the following chapter, we will show the results of more experiments on a wide testbed of randomly generated school timetabling problems, and identify the conditions for hard timetabling problems.

Chapter 6

A testbed of random high school timetabling problems

6.1 Overview

In the previous chapter we saw that the CSP representation of the timetabling problem was easily solved, and gave satisfactory scheduling results for the three high schools. However, does our algorithm only perform well on these three specific problem instances? We seek to validate our CSP model by measuring its performance on general high school timetabling problems. In this chapter we propose a model for a random high school timetabling problem generator. We then proceed to test our CSP algorithm on many possible experimental cases, and determine that certain problems result in constraint networks which are much more difficult to solve than others.

In most previous work on solving constraint satisfaction problems, researchers have tested new algorithms or heuristics on random binary CSP's. A random binary CSP can be described by the tuple $< n, k, p1, p2 >$, where n is the number of variables and k is some uniform domain size; p_1 and p_2 are define as follows: The variable p_1 is a measure of the number of binary arcs (constraints) on the constraint network vs. the total number of possible constraints. Note that for a CSP with n variables, there are $n \times (n-1)/2$ total possible binary constraints. A p_1 value of 0 describes a CSP with no constraints. A CSP which has all of its variables constrained by every other variable has a p_1 value of 1. In our problem there are two different types of binary constraints, subset and intersection. Does this imply that we need a separate p_1 value measures for each of the binary constraints? No, since it is impossible for a pair of variables to participate in both a subset constraint and a intersection constraint, the two sets of constraints are disjoint. Therefore, one p_1 value appropriately measured the total number of constraints versus the maximum possible total. A second measurement of the CSP's difficulty is the value p_2, which indicates the "tightness" of the constraints. A tight constraint is said to be one which is difficult to satisfy. In other words, the constraint has many non-permissable pairs of values. The variable p_2, then, is exactly defined by the ratio of tuples allowed out of all possible tuples

Given the tuple $< n, k, p1, p2 >$ it is a simple matter to construct a random CSP which conforms to these parameters. However, a drawback in this approach is that these random problems bear no semblance to any real world situation. Our own timetabling problems are far from "random" in this manner. If we think of the constraints in our problem as matrices of 0's and 1's, (permissable and non-permissable pairs of values) they would not be characterized as a haphazard array of values, such as:

$$\begin{pmatrix}
0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{pmatrix}$$

This example, showing a p_2 value of 0.5, has no meaning and would never arise in our timetabling problem. If we consider the simple case of two single section courses which may not be scheduled in the same period, then the resulting constraint matrix would be:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & & 1 & 1 & 1 & 0 \end{pmatrix}$$

The reader should realize that these Boolean matrices are really just an implementational detail, but the point is that real constraints are highly *structured* and would not be accurately modeled by one simple value, p_2. Furthermore, the parameter p_1 does not accurately describe the distribution of binary constraints in a real timetabling problem. The constraints themselves are highly localized around particular variables in the problem — usually single section courses — while many other variables remain unconstrained altogether. A further drawback seen in the typical work on random binary constraint problems is the use of k, the uniform domain size. Certainly, our own problem features variables with domain sizes ranging from 4 to 70, so a single value k is too restrictive for our purposes. By far the most interesting property of our timetabling constraint networks, though, is the inclusion of non-binary constraints. The simple binary model $(n, k, p1, p2)$ cannot easily be adapted to include random non-binary constraints, since there are so many different possible ways a non-binary constraint can be formulated. Clearly, then, some other method of problem generation is needed. Also seen in the literature is the use of such "toy" problems as the zebra problem or the n-queens problem. These problem instances are of course not applicable in our work since only an extremely narrow range of CSP's, which bear no resemblance with timetabling constraint networks, would be seen.

We desire a random timetable problem generator which will provide us with a supply of problems with binary *and* non-binary constraints of varying quantity and tightness. The goal is to provide evidence that our search routine is as generalized as possible, and can successfully create a timetable for any school which uses this particular weekly scheduling system. A further motivation in creating a random problem generator is in identifying "hard" timetabling problem instances. It is an interesting problem in itself to understand what makes a problem hard, since we can then predict in the future which other problem instances may be hard. There is a particular phenomenon seen in random binary constraint problems known as the phase transition [16]. This transition refers to the small region of problems which are exponentially difficult to solve, or to at least prove that no solution exists. Normally, constraint satisfaction problems with many solutions are solved quickly, since the search routine may halt when just one solution is found (as our algorithm does). These are known as under constrained problems. Over constrained problems are those that are proven to have no solution after an exhaustive search. Prosser [12] and others have found that the phase transition of binary constraint satisfaction problems occurs between these areas of under constrained and over constrained problems. Here, the expected number of solutions will likely be extremely small, and the solutions may be "clustered" in one particular area of the search tree. As far as is currently known, this region of difficult problems does *not* depend on the search algorithm being used; finding a solution will likely be difficult regardless of the method.

Our random problem generator is able to create a broad range of timetabling problems, including some that are particularly difficult to solve. However, their is no sharply pronounced transition from easy to extremely difficult to solve problems as seen in the binary CSP. There are some distinctly difficult problems which occur when the capacity constraint for the number of rooms in the school approaches a minimum. Other difficult problems exist, but not with any predictable fashion.

6.2 A realistic problem generator

The emphasis of this chapter is on creating a wide range of realistic data. Further, we wish to understand what are the properties of the CSP we are solving; in other words, what makes them different from truly random CSP's. The presence of non-binary constraints makes this work particularly interesting, as does the use of non-uniform domain sizes of the variables. Additionally, we wish to examine what effect varying the real world parameters of our problem has on the time required to solve the problem, and the overall success (number of students scheduled) of the problems.

We first identify three critical parameters of the high school timetabling problem at hand:

- The numbers of students in the school (n)
- The number of courses offered by the school (c)
- The number of rooms (global capacity) in the school (r)

In an effort to cover a broad range of problems, in our generator the number of students will vary from 200 to 2500, the number of courses ranges from 20 to 400; the number of rooms is a function of the capacity of the school. Given the n students, who choose from the c courses, the minimum value of r is the smallest number of rooms which can accommodate all of the resulting classes. We chose the ranges of the parameters to be values which represent the extremes of the high schools found in Edmonton, Alberta.

To generate a random timetabling problem, one needs a random set of student course selections. From these selections, the binary constraints can be formulated, as described in Chapter 4. But how are these random course selections created in a realistic manner? First off we notice that the actual number of courses selected by each student is a random variable in itself. Each student can take a maximum of 8 periods of instructions per term. Since there are 2 terms, there are in fact 16 "blocks" that a student can fill. Full year courses count as two blocks, occupying the same period in each term; double-block courses are also two blocks, while half credit courses are occupy one block. While 16 is the maximum, few students at the non-academic schools elect to spend all of their time in the classroom. Some students only take just enough courses to graduate. If all 16 periods were occupied, there would be more constraints generated, and additionally the students would be more difficult to schedule once a master timetable was created. The point of this discussion is that our random course selections should correspond in number per student as closely as poss:ble to reality. After analyzing the schools' data, we may plot the number of "blocks" of instruction time per student as a histogram, as shown below.

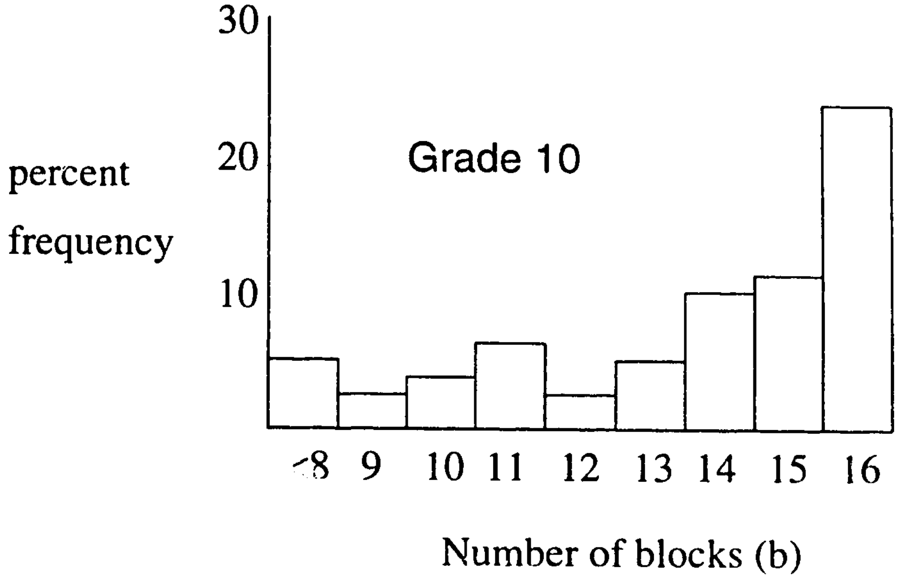

Figure 6.1: Frequenc... of students taking indicated number of blocks of courses. Y-axis indicates the percent of the total students taking the given number (b) of blocks of instruction time. There are different histograms for each grade. Grade 10 students are more inclined to be academical'y orientated, meaning that in grades 11 and 12, there are fewer students who select 16 blocks of instruction.

We may use these graphs for choosing the number of courses that each particular student will select. A random number of blocks of instruction can be assigned to a student by considering the individual probabilities of the student choosing a particular number of blocks.

Now, our random model also must accurately describe which courses are taken by the students. In other words, courses such as English, which virtually all students must take, should be selected in an appropriate frequency relative to obscure courses such as Beauty Culture. This seems simply to be an issue of measuring the actual enrollment in each course, determining what percent of the students take which courses, and then randomly giving students various course selections. However, we must also consider that the random model features as a variable the total number of courses offered by the school (C). So we must ensure that no more than C different courses appear in the students' course selections. To incorporate this fact, we have proceeded as follows. First, the maximum value of C in our generator is 404, which corresponds to the total number of different courses offered by Harry Ainley — which happens to be the largest high school in Western Canada. Using the real student course selection data of this school, all courses are measured by the number of students which register in each. The number of students enrolled becomes the course weighting, w_i. The values w_1 through w_{404} are sorted in decreasing order. Now, suppose we ask for a random problem from a school with $C = 150$ courses. We would then discard the weightings w_{151} through w_{404}, and let TW be the sum of the weights w_1 to w_{150}. Next, each student, given a particular random number of blocks of instruction b as described above, must now be given a random set of course selections. For each course, a random number between 1 and TW is selected, and then this number is converted into a course based on the course weightings. For example, suppose the values of the weightings were:

1 English 10, $w_1 = 520$

2 Math 10, $w_2 = 480$

3 French 10, $w_3 = 450$

.

.

150 Law 10, $w_{150} = 95$

$TW = 5032$

A random number x between 1 and 5032 is chosen; if x is between 1 and 520 the random course is English 10, if x is between 521 and 1000 the random course is Math 10, and so on. Once we are given the number b of blocks for each of the n students, we can proceed to choose the courses in the manner. Duplicate courses (the same course picked twice by a student) would not be permitted.

A less important random variable in the random model is the actual ratio of students in each grade. The students are not split amongst the three grades evenly. Using data from the two composite high schools, the frequency of grade 10, 11 and 12 students respectively are 36%, 30%, and 34%. However, what makes this fact more interesting is that students do not necessarily take all courses of their specific grade level. For example, most Grade 11 students would be expected to take "20" level courses, such as English 20, Math 23, Biology 20 and so on. In fact, some Grade 11 students may be seen taking Grade 10 option courses that were not taken the year before. Or, a Grade 11 student may be excelling in some subject and is already at the "30" level of the course. Further, students who fail to complete a core course (such as English) may be asked to retake that course the next year. At any rate, were it not for this fact our timetabling problem could be much more simple, since the course selections would be three disjoint sets, one for each grade. To accurately model this important detail of students taking courses either above or below their normal level, the database of course selections for the two composite schools needed to be scanned for such occurances. Each course will have a grade "offset" associated with it which indicates the probability that the course is being taken by a student of a different grade level. Once a course is chosen using the method described above, the offset is used and it is randomly decided if the student will actually be in the same course at a grade below or above. Most courses normally expect 1 or 2 students to be from a different grade, with the exception being Band 30 which includes students evenly throughout the three grades.

We may now summarize the steps involved in creating a random timetabling problem consisting of n students and c courses. This routine assumes that the weightings w_1 through w_{404} for the courses have been calculated.

The first step in verifying the correctness of this random model is to experiment with parameters of n and c corresponding to the three real schools. In other words,

```
RANDOM-TIMETABLING-PROBLEM($n, c$)
1.  For each course $i$, $i = 1$ to $c$ do
         $TW = TW + w_i$;
2.  For each student $s$, $s = 1$ to $n$ do
         $x = random(0, 1)$;
         calculate $b$ from histogram of associated grade of student
3.       For each course, 1 to $b$ do
4.            repeat
                   $x = random(1, TW)$
                   find course $C$ from weightings
              until $C$ not already in Course.Select[s]
              $x = random(0, 1)$
              if $x < offset[C]$ add lower grade course $C$ to $Course.Select[s]$
              else add course $C$ to $Course.Select[s]$
```

Figure 6.2: Algorithm which returns n random sets of course selections in the set $Course.Select[1 \ldots n]$ — a random set of course selections for a timetabling problem

how well does the random model predict the actual data? We may simply call the above procedure with the parameters n = 248, c = 45 to represent Old Scona, n = 1661, c = 351 to represent Ross Sheppard, and n = 2236 and c = 404 for Harry Ainley. The table below summarizes the experimental results and comparisons with the results on the actual data. For this test we generated 100 trials for each of the three parameter settings and recorded the average number of students satisfied and the average number of consistency checks. The results are shown in Figure FIGURE:reef.

From these observations, the random problems clearly have a strong correspondence with reality. In general, there are fewer students being satisfied on the random data than on the actual course selections. There may be a simple reason: the "random" students are more likely to choose strange, i.e. truly random combinations of courses. In reality we may assume that students have particular interests and choose courses which reflect this fact. Courses such as Physics, Chemistry and Biology may appear together frequently in a student's course selections. In other words,

School	n	c	a.% sat.	r.% sat.	a.# c.c's	r.# c.c.'s
Harry Ainley	2236	404	98.35	97.93	179329	184593
Ross Sheppard	1661	351	98.80	98.40	88923	104772
Old Scona	248	48	100	99.00	14008	21095

Figure 6.3: Experimental results. The parameters n and c represent the number of students and number of courses in the school. The average percentage of students satisfied (a.% sat.) on the *actual* school data is shown, followed by the same percentage (r.% sat.) on the *randomly* generated school data for the parameters n and c. These numbers are the averages of 100 trials. The results are also shown for the actual number of consistency checks (a.# c.c's) required, along with the number of consistency checks for the random school data.

the course selections are not *independent* events. There may be fewer constraints generated when the course selections come in a more predicatable (realistic) way, but clearly the effect is not overly significant. The algorithm's failure to duplicate the 100 percent satisfaction on the students of Old Scona was due to the inclusion of the "offset" of students taking a course at a grade level different from their own.

6.2.1 Experiments

Having provided evidence that the r[illegible] generates valid data, we may now proceed with experiments on a wide range of random problems. Here, we introduce the number of rooms parameter r. This value r in the CSP corresponds to the global capacity of the non-binary rooms constraint; no period may have more than r courses scheduled at a time. Once a random problem is created, we may calculate the minimum value of rooms by summing up the total blocks of classroom time and dividing by eight. If it were possible to schedule such a school, every room would be full for each of the eight periods per week. Experiments quickly show that there is no problem that can be solved with such a rigid constraint, however. Normally there needed to be at least three empty rooms for a reasonably sized school. Usually a small school (less than 500 students) could successfully schedule its students with one room more than the necessary. For our experiments we calculate the minimum value of r once the random school is generated, and then proceed to vary r from this minimum value to some upper limit. We now present the code fragment for generating a wide range of timetabling CSP's.

```
RANDOM-TIMETABLING-GENERATOR
1. For $n$ = 200 to 2500 by 100 do
2.         For $c$ = 24 to 404 by 25 do
3.                 P = random-timetabling-problem(n,c)
4.                 calculate $R_{min}$
5.                 For $r = R_{min}$ to $R_{min} + 10$
6.                         solve(P,r)
```

To make the experiments more manageable, the search routine halts if no satisfactory solution was found after one million consistency checks. Previous experiments demonstrated that it was extremely unlikely (only one occurance in over two hundred attempts) that any search that went beyond one million checks would ever terminate with a solution; the end result was ultimately that no solution was found. The tolerance for a successful solution was deemed to be 98% student satisfaction; at this point the search would also terminate.

To examine the properties of the three critical parameters — students (n), courses (c) and rooms(r) — we must first consider each variable separately. To proceed we first consider the effect of varying the number of students in the model by holding c at a constant value of 200 courses and r at $R_{min} + 5$. We are searching for the effect of varying the number of students on both the overall search time in consistency checks, and the actual quality of the solution in number of students satisfied. For this experiment we have solved 100 random problems at each of the 20 student values 500 through 2500 (the number of students is incremented by 100). The results are shown in Figure 6.4.

As the number of students increased, the model became increasingly difficult to reach 98% student satisfaction. However, this effect tapers off as the number of students increases beyond 1800. On the one hand, as more students are added to the problem, there are more binary constraints from the students' course selections,

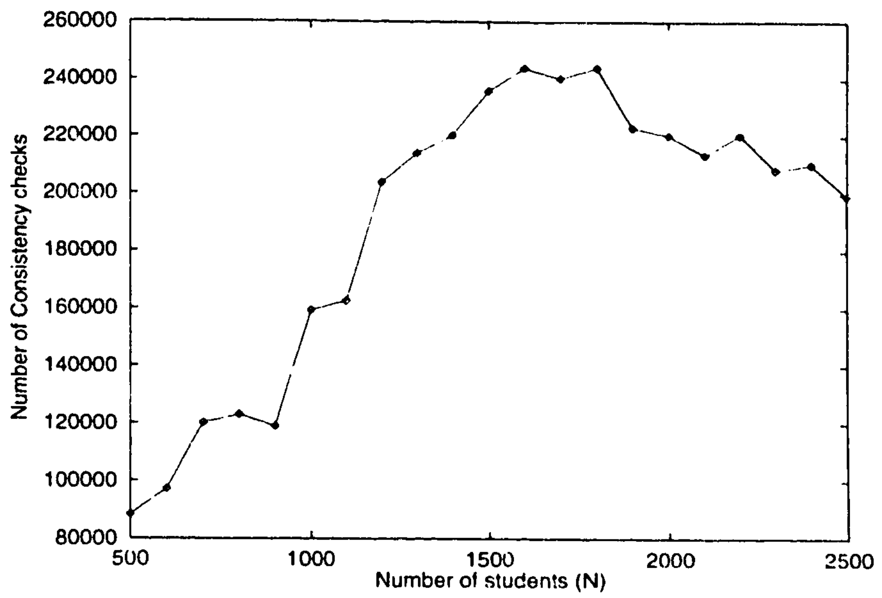

Figure 6.4: Effect of number of students on search time. Here, the parameters are $C = 200$ courses, and $r = R_{min} + 5$ rooms. Each data point represents the average number of consistency checks for one hundred different random problems.

and the problem is harder to solve. [illegible] nally, the final scheduling of students is more difficult since there are more c[illegible]ses with close to thirty students. Satisfying the global constraint also becomes more difficult as more students are added to the problem; since r is held at $R_{min} + 5$, and R_{min} is increasing, the fraction of available rooms decreases. However, as the number of students becomes very large, the number of "singleton" courses (those with 1 section only) goes down, since more students means that more sections are required. Courses with many sections (6 or more) are almost always easy to schedule; due to the chosen representation of the domains of the CSP, the courses (variables) are forced to have only one of each of the 8 periods in their instantiated values, so there is little choice. The singleton courses are most difficult to schedule because they participate in the most "mandatory constraints" as described in chapter 2. All of these factors influence the amount of work needed to find a solution. But what of the student satisfaction rate? We have noted that the algorithm will halt after one million consistency checks if no satisfactory solution of 98% is found. In fact, all of the 100 tests on each of the 20 student values 500 through 2500 were able to find an acceptable solution. This fact, however, most likely suggests that the chosen values of the constants, $C = 200$ and $r = R_{min} + 5$ are "easy" in the sense that no matter what the number of students, a solution can be found.

A more difficult set of problems can arise with different parameter settings. If we let $c = 400$ and $r = R_{min} + 5$, and now vary the number of students from 1000 to 2500. Under these conditions the satisfaction tolerance rate of 98% is much more difficult to reach. Now, recording the average number of consistency checks is much less meaningful, since we are intentionally halting after one million. We instead consider the ratio of trials which are able to successfully find a schedule that satisfies 98% of the students. In the graph below, we have plotted the effect of varying the number of students on the *probability* that the algorithm returns a successful solution, based on 100 instances at each of the 15 values of 1000 through 2500 students. These results are shown in Figure 6.5.

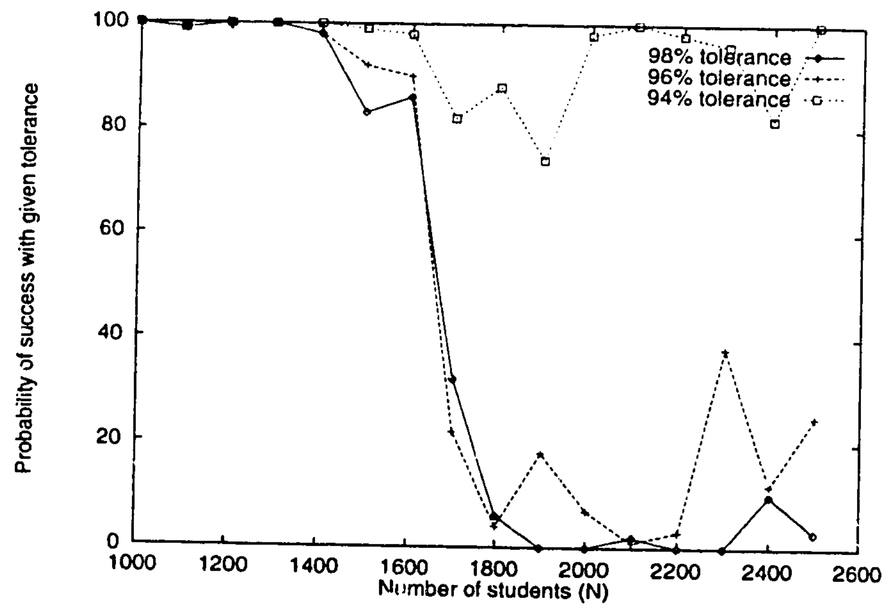

Figure 6.5: The Effect of varying number of students on probability of success with a given tolerance. For example, At a tolerance rate of 98%, the algorithm continues searching until 98% of the students are satisfied, or one million consistency checks are made. For these experiments, the parameters are $c = 400$ courses, and $r = R_{min} + 5$ rooms. Each data point represents the average probability of success over 100 experiments.

As more students are added, the problems becomes increasingly difficult up to a certain point: but again, with more and more students there are fewer "singleton" courses, which are always a burden to the search routine. As the number of students approaches 1900 the probability of solving the problem becomes nearly zero with a tolerance rate of 98%.

Note that we could arbitrarily change the tolerance of 98%. If we were to use 92%, for example, there would be complete success from the algorithm. In this event, the number of consistency checks (c.c.'s) would also be much less. Of course, 92% student satisfaction is generally not acceptable in a real high school. For this same set of difficult problems, we may simply record the best satisfaction rate found. Therefore, for this test we eliminate the halting criteria of 98% and the search routine continues until either 100% satisfaction is reached or one million c.c's have been made. The average of 100 trials is taken, and these results are shown in Figure 6.6.

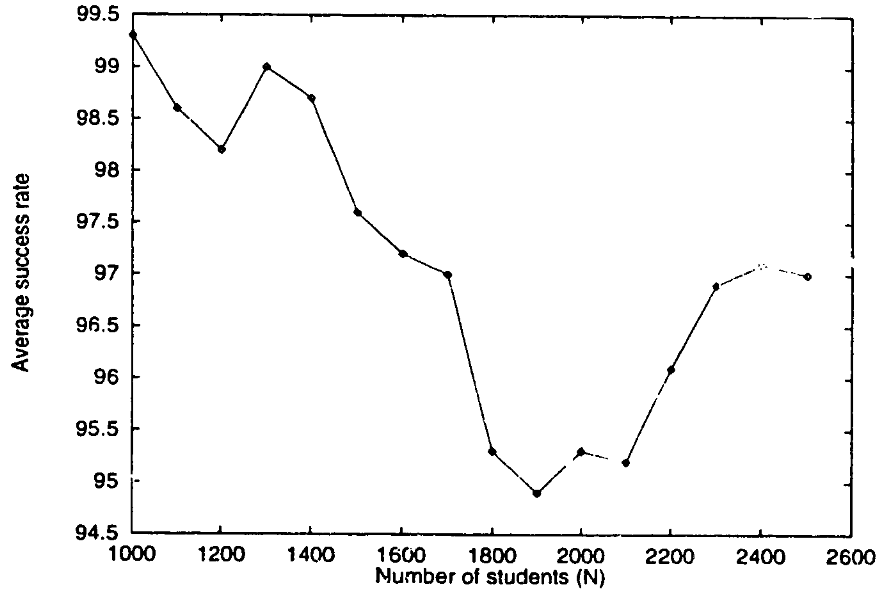

Figure 6.6: Number of students vs. average best satisfaction rate. The parameters are $c = 400$ courses and $r = R_{min} + 5$ rooms. Each data point represents the average of 100 student success rates. Each experiment terminated returning the best schedule found.

In general, for any set of random problems described by the pair of values $< c, r >$, where $c \in \{200 \ldots 2500\}$ courses, $r \in \{R_{min} \ldots R_{min} + 5\}$ rooms, as the number of students increases we see two effects. First, the difficulty (measured by consistency checks) tends to increase up to a certain point, after which the difficulty remains constant and slightly deteriorates. This phenomenon is likely due to the decreasing number of "singleton" courses with more students in the school. Second, the probability of finding a satisfactory solution begins at 100%, for a school of very few students, and then decreases, reaching 0 if there are many courses or there is a tight global constraint.

We may continue experimenting by considering the effect of varying the number of courses, C in the random model. Here, we are directly manipulating the number of *variables* in the CSP. If we use constant values of $n = 1500$ students, and $r = R_{min}+5$ rooms, an appropriate range for c is 150 to 400 courses. Here, there is a direct correspondence between increasing the number of courses and increasing the difficulty of the problem. Under these conditions the average number of sections per course *decreases*, since we are giving the students more courses to choose from. As a result, there are more binary constraints needed, since there are more courses with a small number of sections (between 1 and 3). It is worth noting that, in reality, if the number of students registering in a course is fewer than some minimum value — fifteen for example — then the course may simply be cancelled. Another option may be to "merge" two small courses into the same classroom, receiving instruction from the same teacher. Our random model does not take these actions into consideration, however, and when the number of courses becomes large, there may be some courses with very few students registering in them. Not only does this waste "space" in the school by allocating an entire room for these few students, but it adds another singleton course which may participate in many binary constraints. At any rate, it is clear from these results that increasing the number of courses increases the difficulty of the problem. This property of the random model is true *regardless* of the constant values chosen for n and r.

The third variable in the random model is the global capacity, (r), which indicates the number of available rooms in the school. In general, there may be no more than

r courses scheduled in any one period, per term. For any reasonably sized school, setting $r = R_{min}$, i.e. the minimum number of rooms in the school which can schedule all of the students at a time, results in no solution to the CSP. Let us first consider a typical school of $n = 1000$ students with $c = 200$ courses to choose from. Depending on the random course selections, the value of R_{min} will be between 56 and 58. Let us vary the number of rooms from R_{min} to $R_{min} + 9$ and examine the success rate of the algorithm. For these experiments we again choose 100 random problem instances for each of the 10 values of r. However, in the spirit of reducing the variance of the results, for these experiments we generate the *same* set of 100 random problems, only changing the constant r each time. The graph in Figure 6.7 plots the effect of varying r versus the probability of finding a schedule satisfying 98% of the students. On the horizontal axis is the variable k, where the number of rooms is $r_{min} + k$, and the vertical axis shows the resulting probability of success.

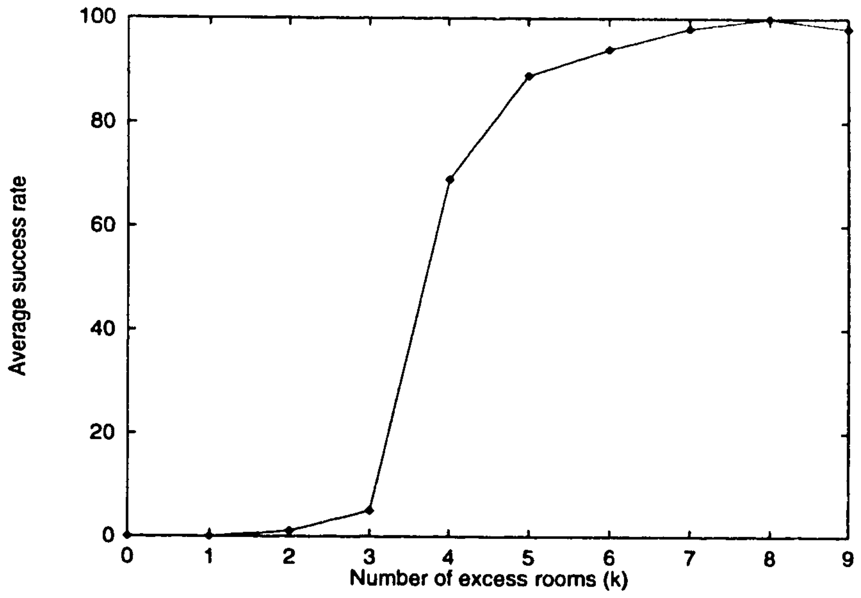

Figure 6.7: Effect of number of excess rooms on success rate, c=200, n=1000. Each point represents the average success rate of 100 problems. The tolerance rate is 98%.

We see a sharp breakpoint between problems with no solution and those that can be solved within one million consistency checks. At $R_{min} + 2$, none of the problems have solutions, while at $R_{min} + 4$ virtually all do.

It would seem, then, that although there exists a theoretical minimum value R_{min} which can be calculated, there also exists a more "practical" minimum value for r. If the number of rooms is lower than this practical minimum, the chances of finding a solution may be extremely small. It is also important to note that we set a halting limit of one million consistency checks; the actual workload to prove that no solution exists is much larger. Allowing the forward checking routine to perform an exhaustive search usually results in at least one hundred million consistency checks, if r is chosen to be slightly greater than R_{min}. Therefore, knowing the practical minimum of a particular problem would be useful. If we say that the practical minimum is $R_{min} + k$, then the value k tends to increase as the size of the problem increases. Consider another set of problems, now with $n = 2000$ students and $c = 200$ courses. We again demonstrate that there exists a practical minimum value of r in Figures 6.8.

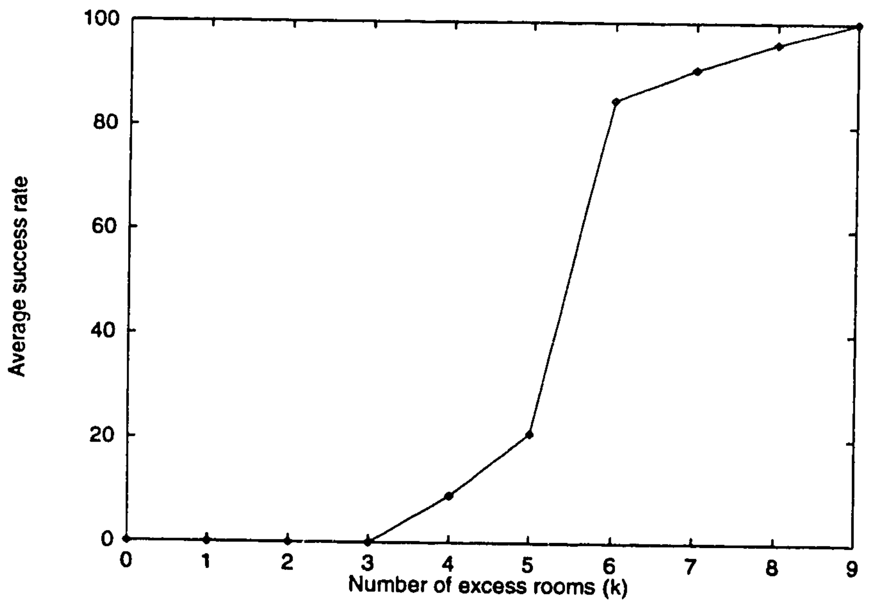

Figure 6.8: Effect of number of excess rooms on success rate, c=200, n=2000. The tolerance rate is 98%.

It is not clear how one could calculate the practical minimum value of r, at present it can only be estimated by considering experimental results.

What makes the timetabling problem difficult? Since there are two main types of constraints in this problem, binary and non-binary, it is interesting to consider what effect each are having. Let us return our attention to the experimental results shown in Figure 6.4. There, we had constant values of $c = 200$ courses and a global capacity of $R_{min} + 5$. Let us repeat these experiments, only now *without* any reference to the non-binary constraints — the global capacity constraint and the teachers' constraints for each subject. We again record the average number of consistency checks from 100 trials. (We are generating the same 100 random problems as used for the data in Figure 6.4). The results for these experiments are shown in Figure refFIGURE:g6.

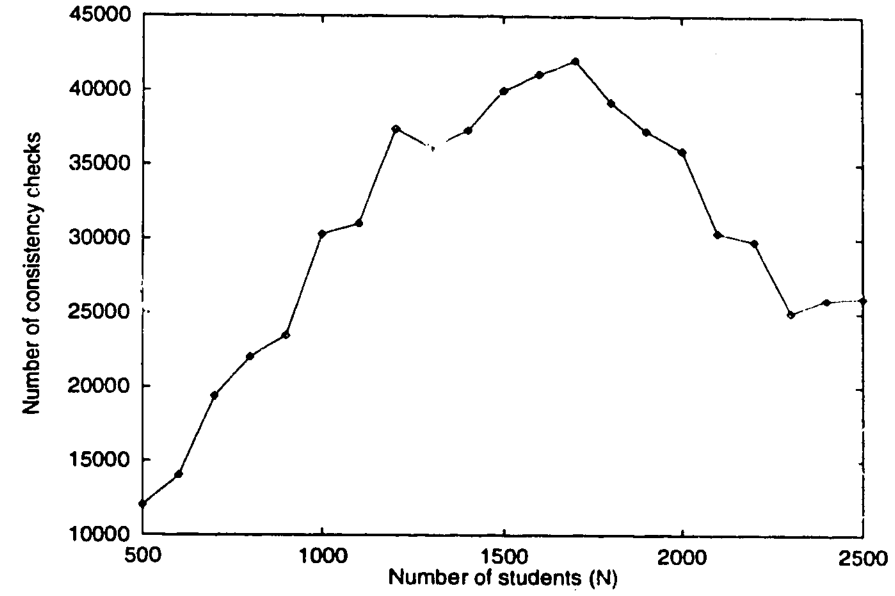

Figure 6.9: Number of students vs. c.c's, no non-binary constraints included. $c = 200$ courses, $r = R_{min} + 5$ rooms. The tolerance rate is 98%.

We repeat these experiments, only now including the non-binary constraints and omitting any reference to the *binary* constraints. These results are shown in Figure 6.10.

The given set of problems proves to be much less difficult when either the non-binary or binary constraints are not included. Comparing the consistency check totals in these Figures 6.9 and 6.10 with Figure 6.4, we see that they are certainly not cumulative. That is, if one were to add the consistency check totals for the problems with only binary constraints with the non-binary problems, the total is of course

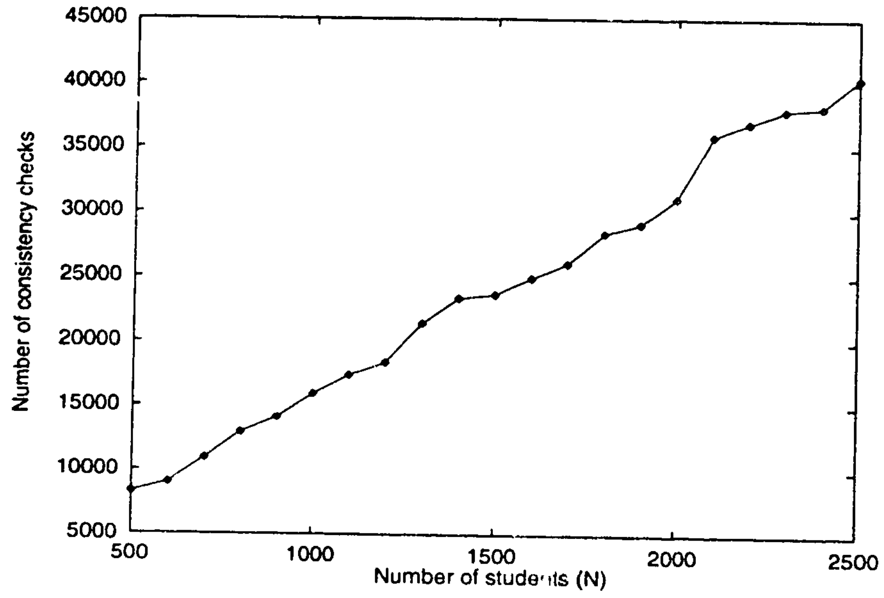

Figure 6.10: Number of students vs. c.c's, no binary constraints. $c = 200$ courses, $r = R_{min} + 5$ rooms, tolerance rate is 98%.

much less than the consistency check totals of the problems with both binary *and* non-binary constraints. This fact is not surprising. It is certainly the combination of both types of constraints that makes the timetabling problem so difficult. To illustrate, let us consider the master timetable created when there are no non-binary constraints. Recall that (in general) no student can pick more than one course of the same subject. For example, a student cannot be taking English 20 and English 30, since the former is a prerequisite to the latter. Therefore, there are *never* any binary constraints between courses of the same subject. Thus, a master timetable with no non-binary constraints would tend to have more courses of the same subject scheduled in the same period. However, once we include the non-binary teachers' constraint, this tactic no longer returns a valid schedule. Since there is a finite number of English teachers, it would be impossible to schedule all of the courses in the same period.

Chapter 7

Summary and Future Work

7.1 Conclusions

This thesis has encompassed the high school timetabling problem. We have devel oped a constraint satisfaction model that captures all of the real-world attributes of an actual timetabling problem. Although the problem is over-constrained, we have devised a met[illegible] of incrementally adding constraints to the problem and building on th[illegible] solution until no further improvements can be made. Our resulti[illegible] good or better than those developed by school administr[illegible] few minutes of computational time to generate. We have a[illegible]etabling problem generator which supplies an adequate te[illegible]his generator to verify the success of our algorithm and iden[illegible] to solve problems.

The [illegible]etabling problem appears frequently in many real world situations. For this reason, there has been considerable effort devoted to automated timetabling. The major approaches to solving these complex problems are integer linear programming, seen in the Operations Research community, and the constraint logic programming methods of Artificial Intelligence researchers. A diverse range of timetabling problems have been studied in the literature. This diversity is due to the many scheduling goals and constraints which can be in place. Some constraints may be seen as maximum or minimum limits, such as the number of hours worked by an employee; some constraints are issues of resource [illegible]ring, such as a classroom not being available to two classes at the same time; other constraints are global in nature, such as requiring a locomotive network to be scheduled with at most ten trains in service. Furthermore, some constraints are subject to being satisfied with a degree of preference, satisfied rigidly, or perhaps optimized. Because of the many different timetabling and scheduling problems which can arise, many different solution techniques have been developed.

Our work has been centred on applying the constraint satisfaction problem (CSP) to a local high school timetabling problem. The problem we have studied involves an assignment of eight weekly periods to the sections of school courses. Each of the students provides a list of course selections which must be satisfied. The sections of the course are scheduled such that there is at least one section for each course available to the student. However, with as many as 2000 students registered, we have found it becomes impossible to satisfy all of the student's demands. Therefore, we view the course constraints as being *relaxable*, in the sense that not all of them need to satisfied. The *rigid* constraints in our problem involve the teachers and rooms of the school. Both of these constraints pertain to a finite capacity; there are a limited number of teachers for each subject and a limited number of rooms in each school.

We have proposed a constraint satisfaction model for the high school timetabling problem. The problem is formulated with courses as variables, and tuples of period values as domains. Both binary and non-binary constraints are used to model student course requests and school capacity and teacher constraints. Our CSP formulation of the problem leads to a network with a manageable number of variables, and, though over constrained, can result in finding adequate solutions. Other potential formulations of the problem, such as having the variables represent each section, lead to having too many variables — there would be more than a thousand for a large high school — and even more constraints than our own over constrained network.

A natural heuristic for the domains we have discovered is to exclude the possibility of duplicate periods appearing in the corresponding tuple of values. Certainly, this idea corresponds to the steps taken by real life schedulers, who initially seek to diversify a timetable as much as possible. Further, eliminating duplication greatly reduces the size of the domains and results in a much smaller search space.

The forward-checking algorithm successfully solves the formulated CSP. While human schedulers are sometimes happy with 95% student satisfaction, we have reached at least 98% on three local high schools. Additionally, school administrators report spending several man-weeks on the timetabling project, while our program requires at most two hours, but generally much less. The innovation in our solution method is the process of *iteratively* adding constraints to the network. Iterative solutions may be poor at first but will improve to some upper limit, until no solution can be found. The main advantage of our iterative method is there will always be a timetable output to the scheduler . Because of the iterative constraint addition, a "best" solution always exists at any point in the search. Our algorithm is one of a classification known as "anytime" algorithms, those whose utility is a function of computation time, wherein a solution exists at any point in time. In general, tree search routines will terminate if no solution is found, and there is nothing of value left to the user, except the knowledge that the constraints were too strong. Additionally, we have designed a greedy algorithm which schedules the students once the master timetable is completed. The scheduling of the teachers and assigning classes to rooms are problems left to the administration of the schools, since that is their preference. The timetables generated by our software have have been validated by the schools.

Finally, in this thesis we have proposed a random model of the school timetabling problem. By identifying the three critical parameters of the number students, number of classrooms and number of courses, we have created a diverse testbed of *realistic* timetabling problems. Since three high schools alone would be insufficient testing for our algorithm, the primary motive for our random timetabling model was to supply a sufficiently large set of realistic problem instances. Our timetabling model is superior to the typical CSP random binary model of $< n, k, p_1, p_2 >$ for our purposes. Since our search algorithm solves timetabling problems, it is pointless for it to be tested on problems with no intrinsic real world meaning generated by the typical random binary CSP. Furthermore, we have identified some particularly difficult to solve timetabling instances. These particular instances can be *recreated*, by giving the generator the same parameters and same random seed, so that comparisons can be made with other, improved timetabling algorithms.

7.2 Future work

The software we have developed is currently only a batch scheduling system. That is, there is some human, pre-determined input, which the user feeds into a "black box", and then a master timetable is returned. In reality, the scheduler would need to be *interactive* rather than batch. There may be many levels of preference within a schedule that the constraints do not adequately model. Certain courses may, for whatever internal reasons, simply need to be scheduled at certain times. The idea behind interactive scheduling is that the user would work in tandem with the computer to solve the problem. A human scheduler may wish to manipulate things such as:

- The number of sections in a course.
- The number of teachers available for a particular subject.
- The semesters in which a course is offered.
- The availability of rooms at certain times.

Reasonably, it may not be apparent what changes need to be made until after a few iterations of the scheduling process. For example, if a large number of students are not being satisfied, opening up another section of a course may prove to be beneficial. Or, it may be realized that more teachers need to be hired if the demand for a course is higher than expected. These human decisions can be greatly influence the success rate of a timetable. The CSP model can be extended to accommodate some these interactive decisions. For example, deciding that a particular course must be scheduled at a certain period could be handled as a unary constraint which eliminates all non-desirable period values.

Furthermore, an interactive school timetabling system would need to handle the frequent "last minute" changes. Generally, in September the high schools face students who are registering late, or making changes to their submitted course requests. An interactive package would allow a master timetable to be "repaired" to accept these changes without beginning the entire scheduling process over again. Beginning again would prove to be catastrophic to the administration, since most of the students already have their individual timetables.

There are many interesting issues within the software engineering aspect of timetabling. The graphical interface of an interactive scheduling package would require much ingenuity. The display would require the whole master timetable, the list of unsatisfied students and their courses, as well as the consumed resources for each period, i.e. the available rooms and teachers. Such an interface would probably best realized under the popular "point and click" paradigm.

The most important future work that should be done is a theoretical look at other applications for the CSP in scheduling. There is no end to scheduling and timetabling problems found in the world. Employee shift timetabling, examination scheduling and so on are all worth considering. At any rate, the timetabling problem will continue to be a source of research interest in the years come.

Bibliography

[1] J. Aubin and J.A. Ferland. A large scale timetabling problem. *Computers & Operations Research*, 16:67–77, 1989.

[2] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. *ACM SIGPLAN Notices*, 22(12):48–60, 1987.

[3] D. de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19:151–162, 1985.

[4] V. Dhar and N. Ranganathan. Integer programming vs. expert systems: an experimental comparison. *Communications of the ACM*, 33:323–336, 1990.

[5] M. Yoshikawa et. al. A constraint-based approach to high-school timetabling problems: a case study. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1111–16, Seattle, Wa., 1994.

[6] R. Feldman and M. C. Golumbic. Optimization algorithms for scheduling via constraint satisfiability. Technical report, IBM Israel, 1989.

[7] Schmidt G. and Strohlein T. Timetable construction - an annotated bibliography. *The Computer Journal*, 23:307–316, 1979.

[8] C. Le Pape. Implementation of resource constraints in ilog schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3:55–66, 1994.

[9] A. Meisels, J. El-Saana, and E. Gudes. Decomposing and solving timetabling constraint networks. *The AI Journal*, Submitted August 1993. Revised November 1994.

[10] G. A. Neufeld and J. Tartar. Graph coloring conditions for the existence of solutions to the timetable problem. *Communications of the ACM*, 17(8):450–453, 1974.

[11] P.Jacques. Solving a manpower planning problem using constraint programming. *Belgian Journal of Operations Research, Statistics and Computer Science*, 33, 1990.

[12] Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *AIJ: Artificial Intelligence*, 81, 1996.

[13] P. Prosser. Binary constraint satisfaction problems: some are harder than others. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 95–99, Amsterdam, 1994.

[14] Jean-Francois Puget. Object oriented constraint programming for transportation problems. *ILOG Schedule Collected Papers*, 1994.

[15] H. Simonis. The chip system and its applications. pages 643–646, Cassis, France, 1995.

[16] B. Smith, S. Brailsford, P. Hubbard, and H. Williams. The progressive party problem: Integer linear programming and constraint programming compared. pages 36–51, Cassis, France, 1995. Available as: Springer Lecture Notes in Computer Science 976.

[17] B. M. Smith. How to solve the zebra problem, or path consistency the easy way. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 36–37, Vienna, 1992.

[18] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[19] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1989.