MODELING AN ACADEMIC CURRICULUM PLAN AS A MIXED-INITIATIVE CONSTRAINT SATISFACTION PROBLEM

by

Kun Wu

B.A. Shanghai Maritime University, China, 1994

B.Sc. Simon Fraser University, Canada, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Fall 2005

APPROVAL

Name: Kun Wu

Degree: Master of Science

Title of thesis: Modeling an Academic Curriculum Plan as a Mixed-Initiative Constraint Satisfaction Problem

Examining Committee: Dr. Andrei Bulatov
Chair

Dr. Williams S. Havens,
Department of Computing Science, Simon Fraser University (SFU),
Senior Supervisor

Dr. Fred Popowich,
Department of Computing Science, SFU
Supervisor

Dr. Uwe Glaesser
Department of Computing Science, SFU
External Examiner

Date Approved: Sep 13, 2005

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

Abstract

This thesis describes a mixed-initiative constraint satisfaction system for planning academic schedules of university students. The proposed model is distinguished from traditional planning systems by applying mixed-initiative constraint reasoning algorithms, which provide flexibility in satisfying individual student preferences and needs. The graphical interface emphasizes visualization and direct manipulation capabilities to provide an efficient interactive environment for easy communication between the system and the user. The planning process is split into two phases. The first phase builds an initial plan using a systematic search method. The second phase involves a semi-systematic local search, which supports mixed-initiative user interaction and control of the search process. Part of the challenge in curriculum scheduling is handling multiple possible schedules that are equivalent under symmetry. We show to overcome these symmetries in the search process. Experiments with actual course planning data show that the mixed-initiative system generates effective curriculum plans efficiently.

To my husband Ming

Acknowledgments

I would like to thank Bill Havens, my senior supervisor, for his outstanding guidance, support, and cooperation over the past two years. I am also grateful to the School of Computing Science for funding me during the first year of my graduate study, and my supervisor, Fred Popowich, and examiner, Uwe Glaesser, for their help and constructive comments.

Special thanks to people at Actenum, particularly, Bistra, Junas, Hussein, and Morten, for their great help and letting me use the framework of ConstraintWorks.

Respect and many thanks go to my ISL colleague Lei and my roommate Yang, for their useful comments and discussions. I also would like to give many thanks to Professor Ljiljana Trajkovic who helped me on technical writing.

Finally, I would like to thank my loved one, Ming for his ultimate encouragements and supports. I cannot forget those nights that he stayed up late helping me with the paper. Without him, thesis would not have been possible.

Contents

List of Tables

List of Figures

List of Abbreviations

CPP	Curriculum Planning Problem
CSP	Constraint Satisfaction Problem
MI	Mixed Initiative
SES	Symmetry Exclusion Search
SBDS	Symmetry Breaking During Search
SC	System Constraint
UC	User Constraint
CP	Constraint Programming
IP	Integer Programming
BT	Backtracking
DBT	Dynamic BackTracking
FC	Forward Checking
BJ	BackJumping
MCLS	Min-Conflicts Local Search
SLS	Systematic Local Search
VOH	Value Ordering Heuristic

Chapter 1

Introduction

The field of constraint satisfaction has been developing rapidly during the past 40 years. It has become an important field in computer science. Members of the constraint programming community are usually more familiar with applications such as scheduling than with action planning. Two well-known textbooks [1, 2] on the subject define scheduling as the problem of assigning limited resources to tasks over time to optimize one or more objectives. Reasoning about time and resources is the core activity of scheduling problems. Sport scheduling and timetabling are two very popular scheduling applications that have received many researchers' attention.

Planning is a synthesis task. It involves formulating a course of action to achieve some desired objective or objectives. In practice, the form is often restricted to simple sequences or partial ordering of actions [3]. The objective in a planning problem can encompass many things, including achieving a set of goals or optimizing some objective function. It belongs to a similar domain and can be interpreted as an extension of the scheduling problem [4].

The Curriculum Planning Problem (CPP) is defined as constructing a set of courses for each semester - over a sequence of semesters - in order to satisfy the academic requirements for an undergraduate university degree. Choosing different numbers of courses to take over the semesters and taking on-leave or internship are actions. The goal is to ensure that the actions included in the plan satisfy the requirements for an academic degree and comply with as many of the students' preferences as possible. Like many other planning problems, it

requires the ability to handle complex logical, temporal, and resource-oriented constraints. A mixed-initiative constraint satisfaction system is proposed in this thesis to solve the problem and exploit the advantages of generating plans on-line, making decisions in a reactive fashion over traditional off-line planning.

There are many academic constraints including course availability, prerequisites, breadth requirements, eligibility rules, and so forth. In addition, there are constraints and preferences imposed by students regarding which academic major to pursue and which courses or electives to choose in a particular semester. Thus, curriculum planning is a mixed-initiative (MI) constraint satisfaction problem (CSP) with preferences. Traditional constructive algorithms for solving CSPs do not support MI reasoning well. Hence, in this thesis, a two-phase approach is explored. An initial plan is constructed using constructive backtrack search, which satisfies all the academic constraints. Then a second semi-systematic local search algorithm is applied to the initial solution. This algorithm supports MI interaction by allowing a user to modify the current course plan directly through the GUI while maintaining consistency of the academic constraints.

1.1 Motivation

For university students, especially freshmen, planning a curriculum is like trying to walk in the dark. It is not unusual for students to overlook taking a course that they need as a prerequisite. Also, students often forget about an elective course they needed to satisfy a breadth requirement. Consulting regularly with academic advisors should avoid this confusion. However, waiting to meet with advisors can be time-consuming and advisors are often not available when decisions need to be made.

Another common problem for undergraduate students is course space availability. Since universities do not keep statistical data on the number of students wishing take each course, it is not unusual that the courses offered by schools fail to meet students' demands. As a result, students are often unable to take

a course they need either because the course is not offered or because it has reached the enrollment limit. Currently, students at most universities still use paper and pens to develop their course plans. In this thesis, a mixed-initiative constraint-based planning model is proposed to replaces the error-prone manual procedure with an automated interactive planning procedure.

1.2 Constraint satisfaction problems

Constraints can be found in many places in daily life. Regulations, restrictions, requirements, capacity, and preferences are all constraints. For example, airline companies have to schedule crews for flights and meet aviation regulations and company requirements. In schools, timetables must be generated in such a way that no instructor should teach two different classes at the same time in two different places. In most universities, an undergraduate student must obtain a certain number of credits to be awarded a bachelor's degree.

A classical example of CSP is the n-queens problem. It is a well-known puzzle among computer scientists. The problem is to place n queens to n different squares on a chessboard, which has n rows and n columns, satisfying the constraint that no two queens can threaten each other. A queen can threaten any other queen on the same row, column, or diagonal. Figure 1.1 shows one of many solutions to the 8-queens problem.

We use the 8-queens problem as an example to explain the CSP. A CSP is composed of a finite set of variables, each of which is associated with a domain and a set of constraints that restrict value assignments of variables [5]. In the 8-queens problem, each queen must be on a separate row and column. We can model a column to a variable whose domain is from 1 to 8 - the range of the number of rows. Therefore, there are 8 variables in the problem. The value of a variable corresponds to the row position of the queen in the column. There are 7 constraints from a variable to every other 7 variables, which contain the allowed positions of the involved pair of queens to ensure they do not threaten each other (the queens have to be on different rows and diagonals).

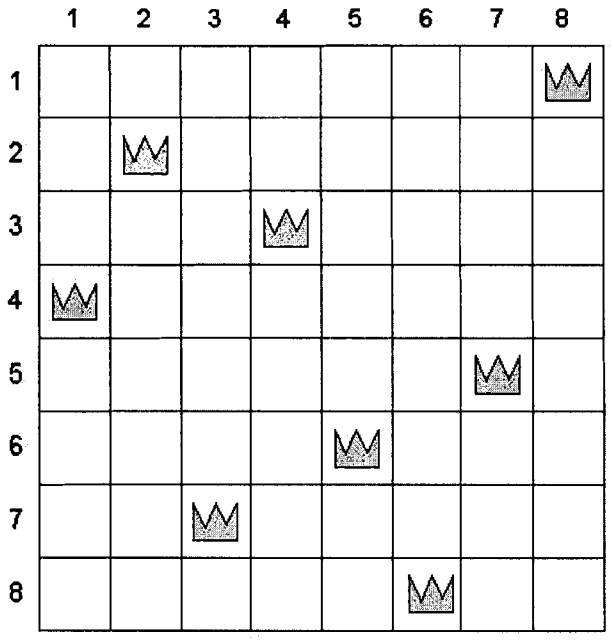

Figure 1.1: A solution for the 8-queens problem.

There are two basic approaches to solve the constraint satisfaction problem:

1. Systematic search: Pick a row position for one queen (a variable) at a time and make sure that no constraint is violated until all eight queens are placed. If at any point, there is no safe place for a queen, move the previous queen whose position has just been chosen to an alternative position that has not been tried. In this way, all positions in the chessboard will be tried if necessary. Thus, a systematic search strategy searches the entire space and is guaranteed to find a solution if one exists.

2. Local search: Randomly choose a position for each queen and then check to see if any queen threatens another. If so, try to move it to a new position. Keep trying to repair the broken constraints in the hope that a solution can be found. The local search method does not explore the

entire search space. Thus, this strategy does not guarantee a solution.

1.3 Overview of the curriculum plan system

The curriculum planning system can be modeled into a CSP as well. Figure 1.2 illustrates a plan produced by the system, with semesters as columns and courses as rows. Each cell of the plan can be seen as a variable, whose domain is the set of available courses. The number of columns in the table is the number of semesters required to fulfill an academic degree based on the course load specified by a user. The maximum number of rows is the maximum number of courses that a student may take in each semester. When an initial plan (see Figure 1.2) is constructed and displayed to a user, the user can directly modify the plan by changing the content of cells in the table. For example, if the user plans to be on-leave for a term (a user preference), he or she simply changes the course load of the term to be zero and then clicks the "Make a New Plan" button. The system will produce a new plan with zero courses assigned in that term and then wait for the user to perform further verification and modifications. Plan construction, revision, and improvement proceed iteratively within a "*decide and commit*" cycle until the user is satisfied with the result.

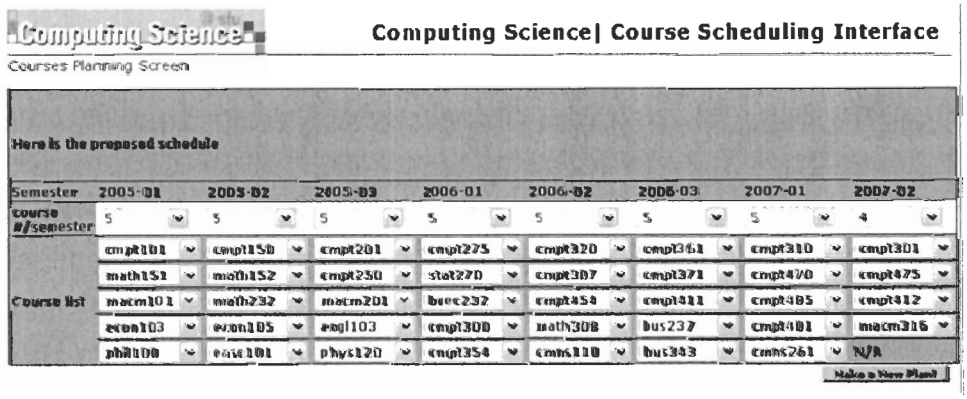

Figure 1.2: Curriculum planning system user interface

In addition to taking requests from users through the interface, the system also inputs data from the database, where system constraints, information about users, academic curriculum, and other related information are stored. Solutions generated by the system must satisfy all system constraints that are transformed from academic regulations. System constraints handled in the system are listed as follows.

- All-different constraint: students should not take the same course twice.

- Prerequisite constraint: some courses must have other courses or a number of credits as prerequisites. For example, in Figure 1.2, 'cmpt300' cannot be planned for a semester unless 'cmpt201' and 'macm201' have been planned in previous semesters. In addition, some prerequisite constraints are defined based on a number of credits. For example, a student cannot take 'cmns261' unless he or she has accumulated 25 credit hours during previous terms.

- Mandatory-requirement constraint: some courses must be taken in order to obtain an academic degree, *e.g.* to obtain a bachelor's degree in computer science, 'cmpt300' and 'cmpt354' must be taken. Hence, a feasible plan must contain these two courses.

- Equivalent-course constraint: a course is equivalent to another course and, thus, students cannot take both courses for credits. For example, 'ensc250' is identical to 'cmpt250'. Therefore, only one of these can be included in a feasible plan.

- Breadth constraint: courses offered in a department may be divided into different academic levels and different areas. A number of courses in certain levels have to be taken from a number of different academic areas to satisfy the breadth requirements. For example, undergraduate courses in computer science are numbered from the 100-level up to the 400-level. Courses numbered at the 300-level or higher are divided into six areas such as Artificial Intelligence (AI), networks, databases and so on.

Students have to take five 300-level computer courses from five areas out of the six. Hence, at least five 300-level computer courses from five different areas have to be included in a feasible plan.

- Depth constraint: A number of courses in a higher academic level from the same subject chosen to satisfy breadth requirements have to be taken, so that students can gain a deeper knowledge in these areas. For example, students in computer science have to take four computer courses numbered at the 400-level. Each of those courses is from four out of the five areas, in which above five 300-level courses are chosen. Then, at least four 400-level courses from four different areas, in each of which a 300-level course has been planned, must be included in a feasible plan.

- Maximum-load constraint: a student's course load can not exceed the maximum course load stated in the student handbook. For example, a student cannot take more than 3 computer courses per semester.

The system uses a two-phase approach to solve the problem. In the first phase, the problem is modeled as a CSP problem. A modified dynamic backtracking (DBT) search method [15] is used to construct an initial plan. In the second phase, a user specifies the requests after reviewing the initial plan. These requests will be added into the system and considered when a new solution is generated for the user. Sometimes, these newly added user requests might make the problem over constrained. For example, because of not being familiar with course regulations, a user requests to take two courses that are equivalent. In this case, no solution can be found to satisfy all system constraints and the user requests simultaneously. When the system suspects that no solution exists, it stops searching, returns the current best solution it has found thus far, and takes the initiative to ask the user for further assistance. The user deletes unsatisfied user constraints and asks the system to verify the plan again. Therefore, the problem at this stage is extended to an optimization problem. The system satisfies all system constraints and optimizes the user's

requests. A systematic local search method, which combines min-conflicts local search with conflict-directed backjumping [21], is used at the second stage in the solving process.

Symmetry occurs in many scheduling, assignment, and routing problems [23]. It occurs in the curriculum-planning problem as well. Courses in a semester (or column) are indistinguishable and can be freely permuted in the semester. Many different solutions found by the search procedure are considered to be equivalent or the same from the logical point of view. Usually we are not interested in getting all these symmetric solutions. If the search algorithm does not exclude symmetries, whenever a solution is found or proven to be inconsistent with the search problem, the algorithm still considers all symmetric solutions, which should not be considered any more.

As an example of the above, there are two solutions S_1, S_2. Assume that they have the same course assignments for most terms except one term T. S_1 contains course A, B, C for term T, and S_2 has course assignments of course B, C, A for term T. Indeed, S_1 and S_2 are equivalent or the same from the logical point of view. If S_1 has been considered inconsistent, then S_2 should not be considered. Consequently, the symmetry problem in the application domain enlarges the search space dramatically. Hence, exclusion of symmetry promises to improve search performance efficiently.

In order to break this symmetry and prune the search tree more efficiently in the curriculum-planning problem, partial-order constraints are added into the model, which prevents to search equivalent schedules.

1.4 Thesis outline

The rest of the thesis is organized as follows. In Chapter 2, the works in related areas are discussed. Chapter 3, I formally define the system and system constraints and also describe how to store (or retrieve) system constraints into (or from) the database; Chapter 4 discusses Mixed-initiative constraint reasoning algorithms used in the system and other techniques applied to tune

up the performance of the system. In Chapter 5, I present the experimental results of using different techniques on real course planning data. I conclude this thesis and discuss ideas for future research in Chapter 6.

Chapter 2

Related work

2.1 Curriculum problems

The curriculum-planning problem tackled in the thesis is not a timetabling problem. The timetabling problem is to fix a sequence of meetings between teachers and students in a prefixed period of time with a set of various constraints satisfied [6]. Timetabling is a very popular scheduling application. It schedules courses, instructors and other resources at many different time slots. Typically, it is a weekly schedule, which tells students where and when to take which courses during a semester. Many variants of the timetabling problem are in the literature based on the type of constraints and resources involved. The timetabling problem can be modeled either as a search problem or as an optimization problem. It has been an active research area and many works have been devoted to it.

The curriculum-planning problem that I discuss here is somewhat related to a timetabling problem, since it deals with courses and resources at school as well. The presented system produces curriculum plans for university students to fulfill their academic career. It focuses on choosing actions such as taking courses or being on-leave *etc.*, and decides what actions students should take during academic years when they stay at universities. Thus, the system helps students to achieve the goal of obtaining an academic degree. Few studies

have studied curriculum planning. Most university students still perform their course planning manually with the assistance of academic consultants.

Castro et al. [7] proposed a CSP model on solving curriculum problems. They tackled a slightly different curriculum problem [1]. The problem is to find a balanced 4-year academic plan based on a given set of courses. Their goal is to provide a plan with a balanced course load among semesters if possible with satisfying the prerequisite constraints and the maximum and minimum course load constraints. Their work focused on showing how constraint solving techniques can efficiently solve the curriculum problem that is too hard to solve by using Integer Programming (IP) techniques. They modeled the assignment of semesters to courses by a two-dimensional matrix with the courses as columns and semesters as rows. The academic load constraint of a semester is stated by a weighted column sum. They handle the prerequisite constraints by implying a strict lexicographical ordering between rows.

Hnich et. al. [8] proposed an interesting CP model for the curriculum planning as well in 2002. They model the problem using a one-dimensional matrix indexed by courses and ranging over semesters to achieve the assignment of semesters to courses. The advantage of their model is that the one-dimensional matrix model can easily enforce the prerequisite constraints by enforcing the ordering on courses that have a prerequisite. The disadvantage is that the one-dimensional model requires some constraints, such as course load constraints, to use variables (courses) to index other variables. Such constraints are typically delayed and cannot propagate efficiently resulting in an inefficient model.

Both *Castro et al.* and *Hnich et. al.* tackled the curriculum problem as a combinatorial optimization problem. They focused more on investigating how to use CP to achieve a balanced course load, rather than to perform the planning. The limitation of the two CSP models is that only two types of system constraints - prerequisite and maximum load constraints - are handled. Compared with their work, the model proposed in this thesis tries to solve the curriculum problem in a more complete way. The course set has to be chosen

[1]The detailed problem description can be found at www.csplib.org. It is prob030.

by the system from the available dataset. In addition to the prerequisite and the maximum load constraints, the system handles many other constraints that usually occur in curriculum planning as well. Furthermore, it integrates MI reasoning into the system to take end users' needs and preferences into consideration. In all, the curriculum-planning problem has been studied more completely in this thesis.

2.2 Mixed-initiative

A mixed-initiative (MI) system is one, in which both the system and the user have an active role to play in a dialogue or problem-solving process [9]. The earliest investigations into the design of mixed-initiative dialogue systems were presented in the paper of Whittaker and Walker [9] in 1990. More recently, MI interaction has been considered in the design of AI Planning systems. Burstein and McDermott (1996) [13] discussed the decisions that must be addressed when allowing users and systems to play a more active role in the problem solving process. They described that the objective of introducing Mixed-initiative into an AI system is to explore the productive syntheses of the complementary strengths of both humans and machines to build effective plans more quickly and with greater flexibility [13].

At the early stage of MI research, researchers usually designed MI systems with a concrete model of initiative [9, 10, 11] . They believed that initiative should be equated with the control over the flow of conversation so that the metaphor of conversation is important in designing MI systems. Recently, Miller and Traum [12] questioned whether it is necessary to model initiative in order to design an MI system. They ultimately argue that a MI system can be designed effectively without a concrete model of initiative when application domains are in a task-oriented collaborative planning environment, like advising. Their argument for not modeling initiative explicitly when the application domain is task-oriented, is that, in these application domains, agents already reason in terms of solutions and goals. Thus, there is no need to introduce

the concept of "initiative" into their reasoning process. The model proposed in the thesis supports their view. Experiments indicate that when the application domain is in a task-oriented collaborative planning environment such as advising course planning, it is not important for participants to realize who has the initiative. Thus, it is not necessary to model initiative explicitly, but view initiative narrowly as controlling how a problem is being solved.

2.3 Constraint programming

Constraint programming techniques are widely used to model and solve planning and scheduling problems. Various efficient algorithms have been proposed during the past few decades. They usually fall into three main categories: systematic algorithms, local search algorithms, and hybrid search algorithms.

Systematic searching algorithms are built upon various backtracking mechanisms [14]. The foundation of various backtracking algorithms is the Chronological Backtracking algorithm. It starts with a zero variable assignment, instantiates variables incrementally and explores a search tree, which is based on the possible values for each of the variables of the solved problem, until obtaining a complete assignment that satisfies all the constraints in the problem. When the current variable is assigned a value from its current live domain, consistency checking is performed against every constraint that the current variable relates to. If the consistency checking fails, another value from the live domain is picked and the consistency check is performed again. If no value in the live domain can pass the consistency check, which means no value in the live domain of current variable is consistent with all assigned variables, then backtrack is needed. The variable, which was assigned right before the current variable, is unassigned and becomes the current variable. It will be assigned a new value, which is selected upon a successful consistency check.

The Chronological Backtracking algorithm demonstrates the main concept of all systematic search methods. The biggest problem of such backtracking-based search algorithms is that they are typically plagued by early mistakes

in the search, *i.e.,* a wrong variable assignment can cause a whole sub-tree to be explored with no success. Two ways of reducing the importance of the early-mistake problem - Backjumping (BJ) and Forward Checking (FC)- are well identified [15, 16, 17].

Backjumping (BJ) [14], aims to reduce the number of backtracks and the number of consistency checks. It tries to directly backtrack to a culprit variable. And then, the search continues from that point. Any work from the current point up to the point where it jumps back to is erased. In order to backtrack to a more effective point, BJ needs to use information about the reason for the failure to identify reasonable backtrack points. Hence, the BJ algorithm adds a data set R to maintain the explanations for the eliminated values. R represents the collection of eliminating explanations for all past variables. Every instantiated variable v_i has a R_i to remember the eliminated explanations for v_i. An explanation records a past variable v_j, where v_j is less than v_i according to the variable ordering, and a list of values that have been eliminated from the live domain of v_i because of the current assignment of v_j.

FC is a technique that takes future variables [2] into consideration to reduce the number of backtracks when it selects a value for a current variable [16]. When the FC scheme makes a trial instantiation of the current variable, it looks ahead toward the future variables and removes from the current live domain of those future variables all values that are incompatible with the trial instantiation. Thus, when the search moves forward to the next variable, it can be sure that all values in the live domain of the current variable are consistent with the past variables. The domain pruning information of every future variable for the current variable has to be stored so that the pruning can be undone when the trial instantiation does not work and a new value needs to be tried.

If FC causes any domain-wipe-out of any future variable, then a new trial instantiation is tried for the current variable. If a domain-wipe-out of the current variable occurs, then a backtracking occurs. By now, we can see the

[2] A future variables is a variable that hasn't yet been assigned a value.

goal of FC is to attempt to visit as few nodes as possible during the search by performing more work at each node when making the assignment decision for the node. Clearly, there are tradeoffs of either more work at each node but visiting fewer nodes or less work at each nodes but visiting more nodes. Hence, it is proper to apply the FC scheme when it is sure that a net saving in consistency checks performed can be gained by visiting fewer nodes.

However, a systematic search still requires that a whole sub-tree be searched to find proof of inconsistency, which is far less efficient for applications having a huge domain. Local search algorithms may be preferable for such cases. Local search algorithms are iterative algorithms that move from one solution S to another S' according to some neighborhood structure. Several local search algorithms have been proposed over decades, *e.g.* Simulated Annealing, min-conflict [19], and tabu search [29]). They perform an incomplete exploration of the search space by repairing infeasible complete assignments. Local search algorithms mainly work on a total instantiation of the variables. They usually consist of the following general steps:

1 Initialization: choose an initial solution S to be the current solution and evaluate S based on a predefined objective function $F(S)$;

2 Neighbour generation: select a neighbour $S'$of the current solution S and compute $F(S')$

3 Acceptance test: test whether to accept the move from S to S'. If the move is accepted, then S' replaces S as the current solution; otherwise S is retained as the current solution. Go back to step 2 - the neighbour generation step.

4 Termination test: test whether the algorithm should terminate. If it terminates, it outputs the best solution that has been found so far; otherwise it returns to step 2- the neighbour generation step.

Various local search algorithms apply different techniques to choose an initial solution and a neighborhood and have different sorts of determination

criteria on making moves. Normally they do not suffer from the early-mistake problem because, once a decision is suspected to lead to a dead-end, it can be undone without having anything to prove. Thus, local search algorithms may be more efficient with respect to response time than systematic search algorithms to find a solution especially when the search space is huge [17]. However, they are incomplete and cannot guarantee a solution.

Cooperation between local and systematic search algorithms has been studied [20, 21, 22] as well. These hybrid methods have led to good results on large-scale problems. There are three categories of hybrid search algorithms in the literature:

- performing a local search before or after a systematic search;
- performing an overall local search, but applying systematic search at some point of the search, for example, to select a candidate neighbour or to prune the search space.
- performing a systematic search improved with a local search at some point of the search, *e.g.*, at leaves of the search tree (over the complete assignments) or at nodes in the search tree (on the partial assignments);

Hybrid methods in the first category are intuitive. Algorithms in the second category apply systematic search in the complete problem domain, but introduce the extra flexibility of local search into small areas to facilitate the search and improve the performance overall. The goal of the third category of algorithms is to exploit some filter techniques such as arc consistency (AC) and FC of backtrack based systematic methods for local search algorithm to improve the search behavior. Methods in the second and third categories try to combine advantages from both systematic search and local search into one approach and seem promising. They can start with either a partial solution or a complete solution and have more flexibility on avoiding the early-mistake problem. In this thesis, I take a systematic approach with some enhancements in the first phase and take a hybrid approach that somewhat falls into the second category in the second phase. Indeed, my goal was to show that solving

the curriculum-planning problem at different processing stages using different methods is a good strategy to produce high quality solutions.

2.4 Symmetry breaking

Many CSPs face symmetry problems. Symmetries give rise to many different solutions found by the search procedure, which are all considered to be equivalent. Without excluding symmetries, whenever a solution is found or proven to be inconsistent with the search problem the search algorithm still considers all symmetric solutions. As a consequence, these symmetries amplify, often exponentially, the search space. Breaking symmetry promises to efficiently prune the search tree. Hence, exploiting symmetry in constraint satisfaction problems has become a very popular topic of research. As a result, a growing number of techniques are being reported in the literature.

Most methods of dealing with symmetry in CSP fall in two general categories. One approach is to modify the search method to exclusive symmetry during the search or prune the symmetry branch [18, 25]. *Backofen and Will* [18] proposed the Symmetry Exclusive Search (SES) method, which is based on the notion of symmetric constraints. It introduces the symmetry version of constraints into the search procedure so that it can be used with either the full set of the symmetries or a subset of all symmetries. *Gent and Smith* [25] present Symmetry Breaking During Search (SBDS). This is a similar method to SES. Methods in this category can be applied to arbitrary symmetries and do not restrict the search strategy. It inserts additional constraints at each branching point in the search tree instead of posting constraints at the start of the search.

Arbitrary symmetries exist in the N-queens problems. As an example, consider the 8-queens problem. Elements in the array of queens $[1 \ldots 8]$ take values from 1 to 8. $Q[i] = j$ means that there is a queen on ith row placed on jth column. The symmetries of this problem are rotational symmetries through 180 degrees. Suppose that the first two Queens are assigned with $Q_1 = 2$ and

$Q_2 = 4$. On backtracking from the assignment of Q_2, an alternative choice such that $Q_2 \neq 4$ is made. Should the symmetric equivalent *i.e.* $Q_7 = 5$ be allowed? If $Q_8 = 7$ is eventually placed, which is the symmetric version of $Q_1 = 2$, then the decisions of $Q_2 = 4$ and $Q_7 = 5$ are symmetrical and then $Q_7 = 5$ should not be allowed. On the other hand, if $Q_8 \neq 7$, then $Q_2 = 4$ and $Q_7 = 5$ are not symmetrical, and then $Q_7 = 5$ should be considered when $Q_2 \neq 4$ is chosen. Hence, when the node of the search tree is created with the choice between $Q_2 = 4$ and $Q_2 \neq 4$, a conditional constraint $(Q_1 = 2 \ \& \ Q_2 \neq 4 \ \& \ Q_8 = 7 \ \Rightarrow \ Q_7 \neq 5)$ is inserted. The symmetry exclusive search procedure inserts this type of symmetry exclusive constraint at every branch to avoid the symmetry.

However, symmetry exclusion is very complex and often hard to implement. Thus, their usage was strongly hindered by the complexity [18]. The other approach is to add constraints to the CSP to convert the problem into an asymmetrical one or a less symmetrical one. This type of approach is more commonly used in practice. Since the task we are left with after converting the problem is still to solve a CSP, the search algorithm is, in principle, unaffected. For example, *Crawford et. al.* [26] have demonstrated how constraints can be added to the models in order to break symmetries. *Puget* [28] presented a formal framework that involves the addition of ordering constraints to break symmetries. *Flener et al.* [23] proposed a similar approach of posting global constraints for lexicographical orderings on vectors of variables to break symmetries in matrix models. *Flener et al.* [23] also reminds us that symmetry detection is graph-isomorphism complete in the general case. Approaches in this category are good at handling symmetries in CP, arising from matrices of variables where rows or (and) columns can be swapped. Many of these types of symmetry have been observed [27], such as the rack configuration problem, social golfers problem, the template design problem [3], and as well as the well-known round robin tournament-scheduling problem (weeks and periods in a schedule are indistinguishable).

[3]detailed descriptions of these problems can be found at www.csplib.org.

The approach employed in the model presented by the thesis falls in the second category. Partial-ordering constraints are added to make the CSP model less symmetrical so that the partial solutions that are symmetric to some infeasible ones will not be considered again as well as the symmetrical solutions will not be returned.

Chapter 3

Curriculum planning modeling

The complete curriculum planning model consists of three parts: the system process, the interface, and the database. Figure 3.1 outlines the model. The system process is responsible for initializations, constraint propagations, and decision-making. The interface is the place where users interact with the system, such as reviewing system proposed solutions, inputting their preferences and constraints and giving feedbacks. The details of the system process are discussed in 3.1. The formal definitions of constraints handled in the system are given in 3.2. The database stores data, which is needed to initialize and construct the system, variables, and constraints such as course information, user information, and relationships among related constraints. The relation between the system process and the database and how to store constraints in the database are described in 3.3.

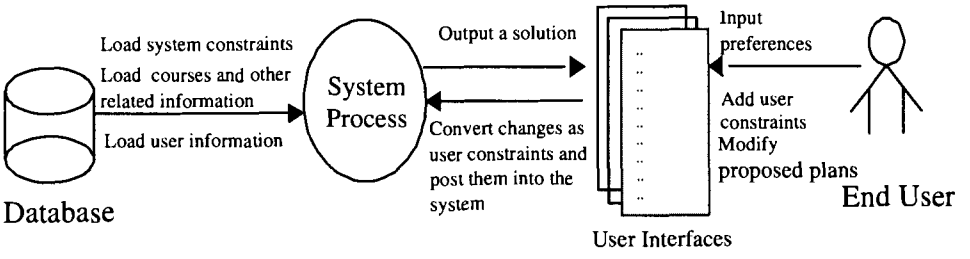

Figure 3.1: Curriculum planning model outline

3.1 System Process

The system process is the control process in charge of solving the problem. The problem is solved in two phases. The structure of the system process is slightly different at each phase. Figure 3.2 illustrates the structure of the system process in phase one. The system process - the component circled by the dotted lines, contains a system controller and a user agent. The system controller is responsible for the creation and propagation of variables and system constraints. The user agent is responsible for collecting and maintaining user related information.

3.1.1 System controller and user agent

When the system starts, a system controller will be created for the logged-in user. In turn, the controller will create variables and system constraints based on regulations and information retrieved from the database or input by the user. System constraints are not shared with the user agent, but only controlled by the system controller. The system controller is also responsible for maintaining a current set of feasible solutions, propagating the consequences of decisions, and constructing the final solution incrementally. A feasible solution is a plan that satisfies all system constraints in the model. The role of the user agent is to create and maintain the current set of users and their specific user constraints, and to retract requests that have proved unsatisfactory. The system controller has the control over the user agent. In phase two, requests from users are collected through the interface and transformed into user constraints, which are maintained by the user agent and shared with the system agent. User constraints are unary and also retractable. They are kept active until being retracted.

3.1.2 Problem solving process

As mentioned in Chapter 1, the problem is treated as a CSP and solved using constraint-programming techniques. There are three major steps for solving this CSP:

1. Gather the information required for the creation of variables and constraints;

2. Create variables and constraints;

3. Perform the search.

Data required to create variables are collected from the database and (or) interface if needed. When the system starts, it waits for a user to log in. Once a student logs into the system, he (or she) faces an interface to ask him (or her) to check the profile including the coordinates, courses that have been taken, the type of the major, interested areas and so on, while a system process dedicated to this user is created, which contains a user agent and a system controller. If the information in the profile is obsolete, he (or she) has to perform an update before he (or she) can proceed. If no related information is displayed, the user needs to input the information and perform an update so that the input information will be saved in the system.

I would like to walk through the planning process using a student who is an intended major in computer science as an example. Assume that he is a new student and it is his first time using the system. Thus, he should have no record of taking any course in the system. He needs to verify this. If no major type is recorded, he has to specify the type of the major. The information he has input is saved into the database and temporarily cached in the user agent so that the system controller can use it later during the solving process. For existing users, most information has been stored in the database. They need to verify and make changes if necessary. Upon confirmation, he would be asked to select areas that sound interesting to him. For a computing science major, there are 6 available areas. Assume he selects the first option, Artificial

Intelligence. He can specify the average course load that he would like to have, otherwise the default minimum course load will be used. He also can state the semesters that he likes to be off from the school and so on. After specifying the preferences, he can simply ask for a plan.

When the system controller receives the request, it creates variables and system constraints based on the number of taken courses and the total number of credit hours required for a program. The variables are the basic elements in the plan. The number of variables represents the number of courses that the user needs to take to finish the degree. In the example, the new student needs 40 courses, which will have a total of 120 credit hours, provided all courses have the same 3 credit hours.

3.1.3 Variables and their domains

Figure 3.2 is an instance of the system process having n variables. In our example, the value of n is 40. The system controller creates 40 variables in a chronological variable ordering. Fewer variables would be created if the user in the example was a student who had taken a few courses. For example, if the user has had 12 credit hours, then only 108 credit hours are needed to graduate, thus, only 36 variables will be created. The number 40 (or the number 36) is the maximum number of variables that may be needed, because the system assumes that all courses planned to take have the minimum 3 credit hours. If several variables are assigned with courses that have 4 or more credit hours, the system controller does not need to assign courses to 40 (or 36) variables anymore. However, we do not know what values will be assigned to variables and cannot foresee the exact number of variables needed, we always create the maximum number of variables and at the end of the search, some variable(s) may not be assigned a value representing a course, which depends on the assignments of the rest variables. These variables will be assigned a null value.

Every variable has a static domain, which is a finite collection of continuous integers. Those integers are called values in the domain. Each value in the

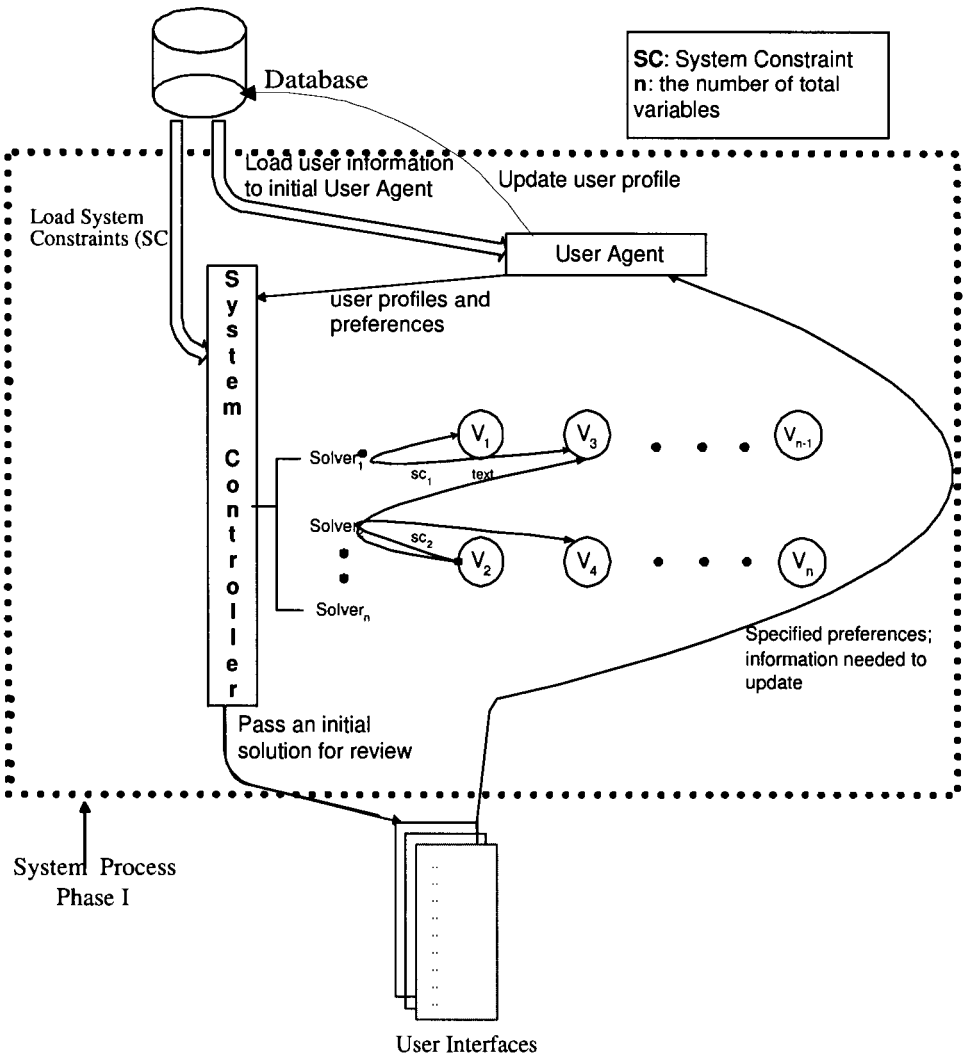

Figure 3.2: System process model structure phase I

domain represents an available course stored in the database. All variables in the model have the same static domain, which is the complete list of all available courses in the database. An alternative method is to give variables representing courses in different school years, a different static domain, such as to exclude lower division courses from the static domain of a variable in a senior

school year. The alternative method sounds more reasonable and efficient, but unfortunately, it will exclude reasonable solutions.

It is true that usually a fourth-year computer science student probably won't take a first or second year computing science course, but it is still possible, if the course is a non-mandatory course and does not violate any equivalent constraint, which means students still can get the credit hours by taking that course. For instance, a fourth year computer science major student can still take a 200-level c++ programming course *e.g.* 'cmpt212' and get the credits. There are students who do so even though it is not popular. However, if the model applies the alternative method and then the static domain of variables in the senior school year does not include the lower division courses, the method will eliminate the possibility discussed above. When users would like to make this type of request in their plans, the requests would not be able to be approved by the system, which is unreasonable and should not happen. Hence, the alternative method is not used. Every variable in the model has the same static domain and leaves the system process to prune and decide the value through constraint propagation.

3.1.4 Constraints and solvers

When a variable is assigned a value, it can be constrained to assume a value from its current domain, which is usually performed by a type of entity called a solver. A solver enforces a constraint on what value(s) a variable can take. It also can constrain the variable to only take a value from a subset of the initial domain. Thus, when creating variables, the system controller creates a solver for each variable and compiles them with the fixed total ordering among them. After creating the variables and solvers, the system controller creates system constraints. When creating a constraint, the controller needs to decide the related variables and propagation directions of each constraint based on the type of the constraint and the information controlled in the user agent, such as the intended or approved major, interested subject areas and the number of courses that the user intends to take per semester and so on. Each constraint

has to be registered with the solver of the related variables so that whenever a trail assignment is initiated, the solver invokes propagations of all constraints related to its variable.

In our example, this new computer science major student needs to take at least 8 lower division and 5 upper division computer and mathematic courses. Hence, there would be a total of 13 mandatory constraints created. The lower division mandatory constraints are prerequisites for most of the upper level courses, and will normally be taken in junior school years normally. Thus, lower-division mandatory constraints will be related only to the junior school years (the first half of columns in the plan). There are also breadth constraints, (*e.g.* taking one upper division course from 5 out of 6 areas), and depth constraints. In addition, the all-different, prerequisite and maximum-load constraints, which apply to students in any major, are created as well.

3.1.5 Creation of system constraints

Constraints are created in a chronological order. Constraints related to lower division are created before constraints related to upper division. They are formally defined in the next section to ease the understanding of the constraints and their implementation. Furthermore, the number of system constraints are various for different types of majors and/or subject areas, *e.g.*, taking 'macm401' is a mandatory constraint for a student joint major in computer science and math but not for a student major in computer science.

In Figure 3.2, two examples of system constraints are shown among variables. One is an equivalent constraint (the system constraint sc_1 in Figure 3.2). It is posted on two variables v_1 *and* v_3. This type of constraint is called a binary constraint since it is related to two variables. This constraint is registered with both solvers that control two variables respectively so that the propagation direction can be dual direction. Since the propagation is dual-direction, a line with arrows at both ends is used to illustrate the dual-direction propagation (see sc_1 in Figure 3.2). Whenever the value of v_1 (or v_3) is changed, a verification on live domain of v_3 (or v_1) will be performed.

The other typical example of system constraint is a prerequisite constraint (see sc_2 in Figure 3.2), which involves 3 variables, v_2, v_3, *and* v_4. The propagation directions are one way, always from v_2 to v_3 and from v_2 to v_4. This constraint only needs to be registered with the solver that controls v_2 since this type of constraints propagates in one direction only. A line with an arrow at one end pointing to the direction that the propagation goes into (See Figure 3.2) is used to represent the constraint. At this moment, there are only system constraints in the system and no user constraints. The user agent is only responsible for caching the preference data collected from the interface and (or) loaded from the database.

Upon finishing the creation of system constraints and the registration with corresponding solvers, the system controller starts to find the first feasible solution - a solution that satisfies all system constraints. The system is in the first phase and uses the first search algorithm, which will be discussed in 4.2. Once a solution is found, the controller posts it through the interface for the user to review. A sample feasible solution may look like Figure 3.3. From the figure, we can see that there are a total of 40 cells in the plan (8 x 5). Each cell holds a variable. All cells have been filled with an abbreviation of a course except the last one, where a "*N/A*" is filled in. It is because three (or more but less than five) courses assigned in the plan have a credit hour greater than three. Thus, only 39 variables need to be assigned a value (an available course in the database). The last one does not need to be assigned any real value but a null value, which does not represent anything meaningful.

3.1.6 User requests

After reviewing the initial plan, the user can directly modify it through the interface for any unsatisfied places in the plan. In our example, assume the user does not like to take *cmpt361*, which is a computer graphic course. He clicks on the cell in the table and picks another choice that he likes from the drop down list. (See Figure 3.4). The user can make as many changes as they like. These modifications are called user requests.

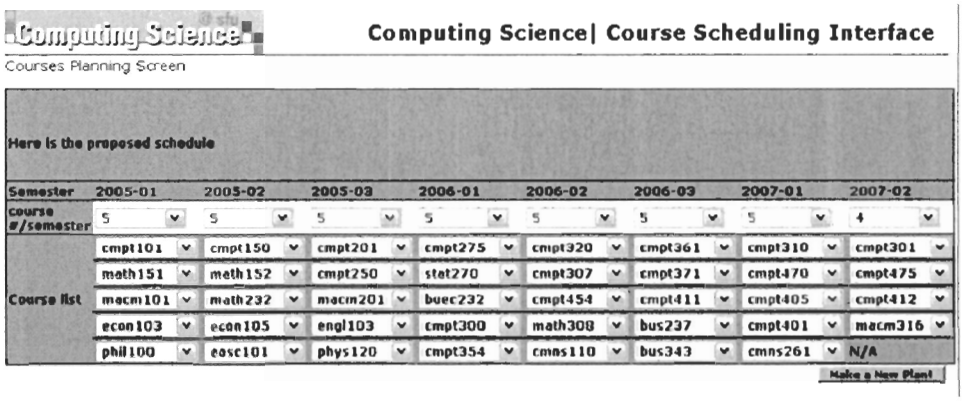

Figure 3.3: Curriculum planning system user interface I

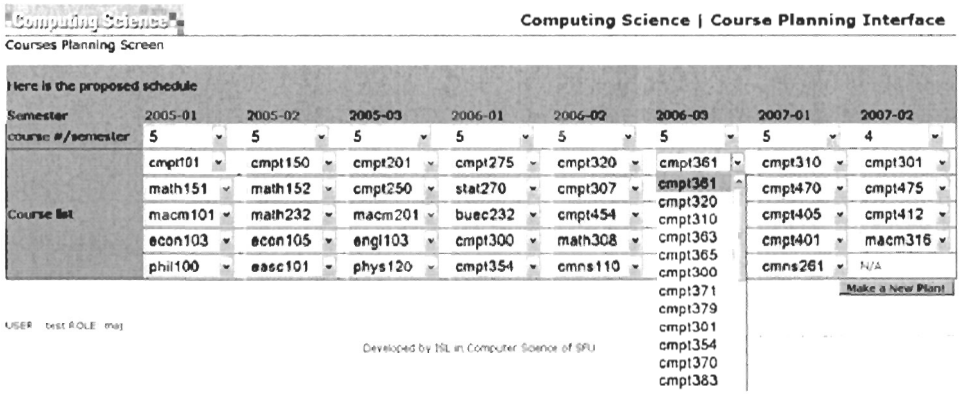

Figure 3.4: Curriculum planning system user interface II

If the user increases or decreases the number of courses to take in a semester of the solution, the system agent will increase or decrease the number of variables in that semester accordingly. If the user requests to take on-leave, which does not take any course for a semester, then NULL values will be filled in for that semester. If the user is willing to take a practicum working in industry in a semester, which is also represented in a short term as "co-op", she or he selects the "co-op" choice provided in the interface (see Figure 3.5), then the

system agent creates only one variable for the semester.

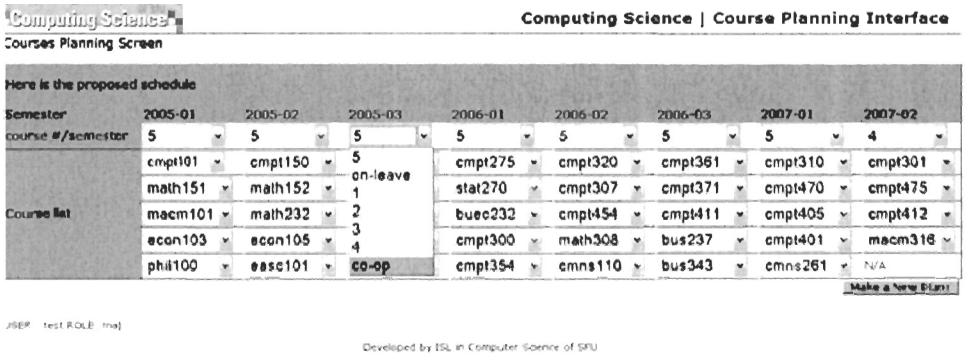

Figure 3.5: Curriculum planning system user interface III

Once the user has made desired changes, he takes the initiative to ask the system to find out if the changed plan is feasible by clicking on the button of "Make a new plan". These changes are reported to the user agent. The user agent, then, transforms the changes into user constraints and posts them on the corresponding variables. Meanwhile, it notifies the system controller of the newly added constraints. Now the system enters the second phase.

When receiving the notice, the system agent remodels the problem based on the requests, which includes creating new variables, re-constructing the constraints and posting them on corresponding variables if necessary. Figure 3.6 shows changes to the structure of the system process when the user makes changes on two cells relative to Figure 3.2. These two cells contain variables V_1 and V_{n-1}. In Figure 3.6 the user agent transforms the requests made on the two cells into two user constraints UC_1 and UC_2, and then engages them with corresponding variables V_1 and V_{n-1} belonging to the two cells. Because a user constraint is a unary constraint, it only acts on a single variable. A loop starting from a variable and going back to the same variable is used to illustrate the user constraint. In addition, since both user constraints are registered with the system controller, the two lines linking the two variables to

the system controller are drawn to indicate the registrations. (see Figure 3.6).

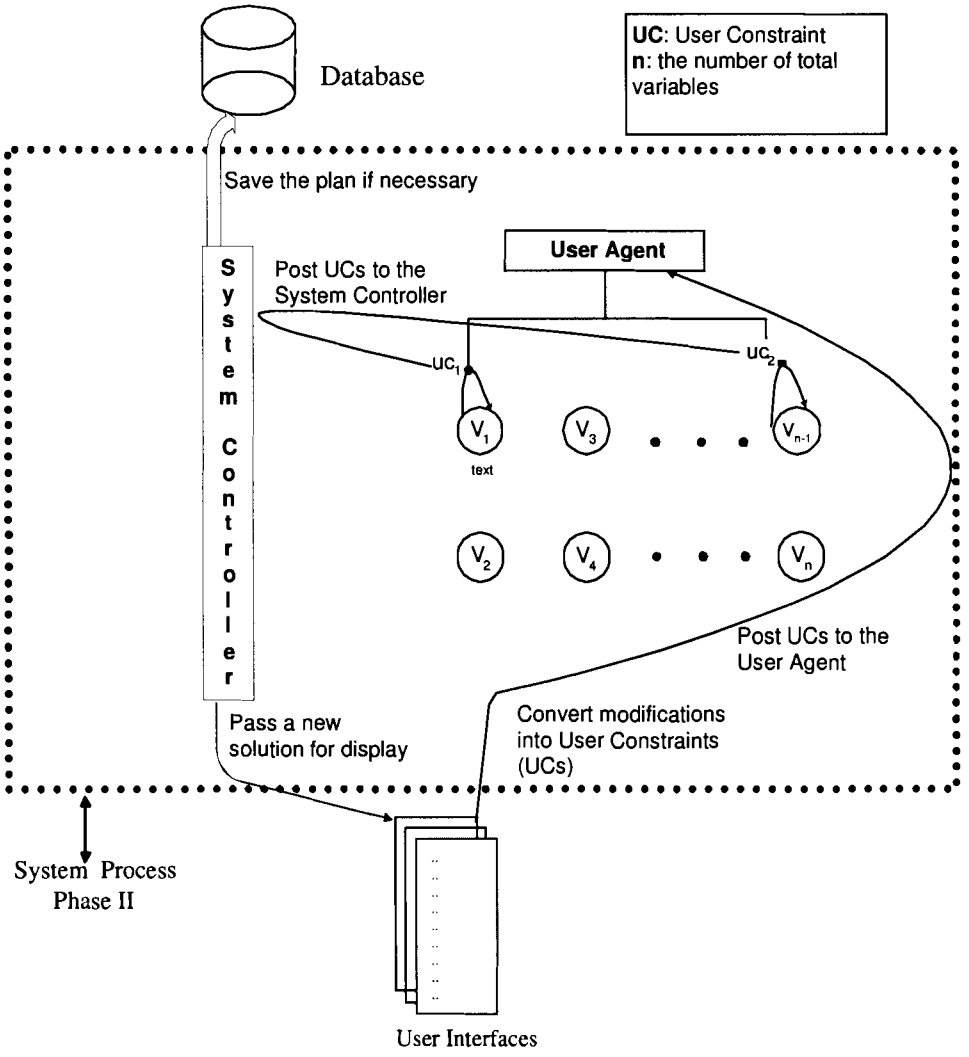

Figure 3.6: System process model structure phase II

After finishing the reconstruction, the system controller uses the solution provided by the first phase as a starting point and initiates a systematic local search to find out if the changed plan is feasible. If the changed plan complies with system constraints and newly added user constraints, then the solution

is confirmed with the user. If it does not, the controller tries to find a solution that does. The search algorithm will be discussed in 4.3. When the system controller finds a new feasible solution, it displays the solution in the interface for the user to review.

Sometimes, the user may put a request that does not comply with school regulations, *e.g.* a student wants to do a co-op in the third semester, which is usually not suggested by the school. This is the case where a user constraint conflicts with a system constraint, and then the user constraint has to yield the right to the system constraint. The controller will notify the user agent of the infeasibility of this user constraint and ask the user agent to retract the user constraint from the system. The user agent will retract the constraint and also send a notification to the interface to notify the user about the failure of fulfilling the request. If the user chooses to enforce the request, then the system will halt since it cannot solve the confliction and generate any solution.

If user decides to change the preferences, they can click the back button on the browser to go back to the previous user profile/preference screen, reset the preferences and then ask for another plan. In this case, the system controller will go back to the phase one and restart the search using the first algorithm and then go through the refining steps depending on user's actions. The system and the user work together in a cycle of searching, reviewing, and modifying iteratively to construct a plan that the user is satisfied with. The final plan will be saved into the database and also emailed to the user for reference.

3.2 Definition of the system constraints

The Curriculum Planning Problem (CPP) is modeled as a Constraint Satisfactory Problem (CSP) problem, defined as a triple of *(V, D, C)*. V is a set of ordered variables, corresponding to the slots in the plan that we need to assign values to. D is the domain, a set of possible values associated with a variable v, where $v \in V$. In CPP, the domain D is the complete list of available courses. C, a set of constraints that restrict value assignments of variables, represents

the collection of the academic constraints in CPP.

In our model, each cell in the table contains a variable. V is organized as a matrix of variables with p columns and r rows (Figure 3.3). The number of columns represents the number of semesters planned. The number of rows represents the number of courses planned in a semester. p is used to represent the number of possible semesters that the user may take to finish an academic degree, and r is used to represent the maximum number of courses that can be taken per semester. The collection of available courses is the domain D. The size of D is the number of all available courses, which we use m to represent. For all $v_{ij} \in V$, an assignment pair (v_{ij}, d_j) indicates that the value d_j is assigned to the variable v_{ij}, which belongs to the cell located at the ith row and the jth column in the table. A solution is denoted as a matrix of variable assignment pairs that satisfy the set of constraints.

The system constraints have been described informally in Chapter 1. They are described formally in this section. Symbols used in the definitions are listed in Table 3.1.

Table 3.1: List of symbols

V:	the collection of all variables.
D:	the collection of all possible domain values for a variable.
C:	the collection of all system constraints managed in the problem.
v_{ij}:	a variable at ith row and jth column in the matrix of variables.
p :	the number of periods (or semester) in the plan.
r :	the maximum number of rows among all periods.
n :	the total number of variables in the problem domain.
m :	the number of courses or domain elements in the variable domain D.

The definitions of the constraints are stated as follows:

All-different Constraint: Students can not take the same course twice and get credited twice on the same course. For any two variables v_i, v_j, where

$v_i \in V$ *and* $v_j \in V$. If $v_i \neq v_j$, and then for the corresponding variable assignment pairs of (v_i, d_i) *and* (v_j, d_j) *with* $d_i \in D$ *and* $d_j \in D$ in the solution, it is always true that $d_i \neq d_j$.

Prerequisite constraint: A prerequisite of a course can be either a list of courses or a number of credit hours. When the prerequisite in a prerequisite constraint is a list of courses, this prerequisite constraint is defined as follows. Let D' represent the prerequisite for a course d. Thus, D' is a set of course values, *s.t.* $D' \subset D$ and $d \in D$ *but* $d \notin D'$. Let V_i represent the set of variables at the ith column, where $1 < i \leq p$. A variable v_i, $v_i \in V_i$, can have a variable assignment pair of (v_i, d), if and only if, it is true that for every $d_k, d_k \in D'$, a variable assignment pair (v, d_k) exists in the solution where $v \in V'$, $V' = V_1 \cup \ldots \cup V_j$, and $1 \leq j < i \leq p$. When a prerequisite is a number of credit hours, the prerequisite constraint can be described as follows. Suppose a course d, $d \in D$, has a prerequisite of a number of credit hours. We use ϵ to represent the number of credit hours. The course d can not be assigned to any variable v_j, $v_j \in V'$, $V' = V_j \cup \ldots \cup V_p$, until the number of the accumulated credit hours is equal to ϵ after the variable assignment (v_i, d_i) is made, where v_i is a variable at the ith column, $v_i \in V_i$, and $1 \leq i < j \leq p$.

Mandatory-requirement constraint: A mandatory-requirement constraint requires that a student must take a course d, $d \in D$ in order to obtain a degree. We use D' to be the collection of the mandatory courses. Clearly, $D' \subset D$ is true. Thus, we can say that a mandatory requirement constraint is satisfied if and only if $\forall d \in D'$, a variable assignment pair (v, d) exists in the solution.

Equivalent-course constraint: A student can not take two courses that are logically equivalent to each other and be credited on both. We define an equivalent-course constraint as follows. Let's use D', $D' \subset D$, represent a set of courses that are logically equal. If a variable assignment (v, d), *with* $v \in V$ *and* $d \in D'$, exists in the solution, and then, for every

variable x, $x \in V$ with a variable assignment (x, d_x), $d_x \notin D'$ is always true unless x and v are the same variable.

Breadth/Depth constraint: A breadth/depth constraint requires a student to take courses from different areas and different academic levels. It is defined as follows. A subset of domain values D' is divided into k groups, $s.t.\ D' = D'_1 \cup D'_2, \cup \ldots \cup D'_k$. We say that the constraint is satisfied if for every D'_i, $D'_i \in D'$, a variable assignment (v, d) $with\ v \in V and\ d \in D'_i$ exists in the solution.

Maximum-load constraint: The course load of a student in a semester is restricted by the maximum-load constraint. It is defined as follows. Given a group of variables $V' : \{v_1, \ldots, v_i\}$ and their corresponding assignment pairs $\{(v_1, d_1), \ldots, (v_i, d_i)\}$ in the solution, where $1 \leq i \leq r$, it is always true that

$$\sum_{r=1}^{i} c_r \leq Max$$

Here, $c_1, \ldots, c_i$ represents the number of credits for $d_1, \ldots, d_i$ respectively and Max represents the maximum course load allowed in each semester defined in the system.

3.3 Storing constraints in the database

Every university maintains databases to store data for courses and students. The basic information about courses may include course name, subject area to which it belongs, the number of credits, and etc.; basic student information may include name, contact information, year enrolled, and any courses previously taken (and grades, if any). The database may be extended to store the constraints for processing the course planning. More tables need to be added to store system constraints, such as prerequisite constraints, mandatory constraints, equivalent constraints, breadth constraints and depth constraints.

A complicated constraint like prerequisite constraint is represented in a tree structure in a database table. The table may have three columns: *course*

id, *prerequisite course id*, and *relationship*, which indicates the prerequisite relationship between the two courses. There may be 5 types of relationships:

- the prerequisite course is the sole request for a course, which means students may take this course as long as they have taken the prerequisite.
- the prerequisite course is a co-prerequisite for a course, which means students still cannot take the course if they have just taken this prerequisite.
- the prerequisite is a number of course credits; either lower level or upper level course credits can be counted.
- the prerequisite is a number of course credits from upper level courses.
- the prerequisite is a number of course credits from lower level courses.

Related prerequisites are stored like a tree structure in the table. For example, if course A has prerequisites of B or "C and D"; and C has prerequisites of E or F; B has prerequisite of "G and H", then prerequisites for A are stored like a tree structure in the table (Figure 3.7). Before a student is eligible to take course A, she or he either has to take both G and H, and then B; or has to take one from E and F and then take both C and D. When loading the constraints into the system, a query for finding all courses that have prerequisites will be executed and results of the query will be used for processing a solution.

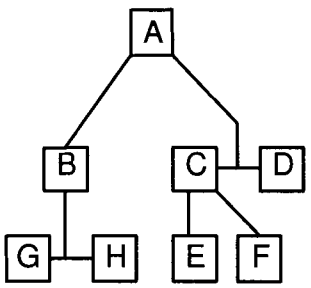

Figure 3.7: A example of a prerequisite tree structure

We can either add columns into the existing course table or create a new table to store mandatory constraints. Because sometimes *OR* relationships may also exist among mandatory courses. For example, computer science students must take either *statistics 273* or *business 254*. Hence, it would be easier to add, delete, and maintain the constraints if we create a new table to store them. The table needs to remember which department, what degree, and what level (upper or lower division) the mandatory requirement is for.

Equivalent constraints are very easily implemented in the database if a course has no more than one equivalent course. This condition is applied in the experiment model. This type of constraint is stored by adding an alias column into the existing course table. If a course has an equivalent one, then fill the field with the equivalent course id. If a course has more than one equivalent courses, then a dedicated table needs to be added in to remember the mapping from a course to one of its equivalent course.

How to store breadth (or depth) constraints depends on detailed requirements of the school. In my experiments, the breadth requirements require computer science majors to take one 300 level course from five different areas out of the six areas. Thus, I have a column in the *course* table to indicate the area that a course is in. In addition, a dedicated table is created to remember all areas and their descriptions. In this way, it is easy to delete or add new areas, in turn, it will be easier to update the change when breadth (or depth) constraints are changed.

The actual data used for the experiment was from the Computing Science Department of Simon Fraser University. Descriptions of database tables, which include names of fields in the table, their data type, and default value used in the experiment for storing constraints are listed as Table 3.2 to Table 3.9. .

The *Course* table (Table 3.2) forms the static domain of variables. The column of *alias_id* is to identify an equivalent constraint. It stores the cid of an equivalent course if it has one. The column *cid* will be used in the *Course prerequisite* table (Table 3.3) to represent courses. It is a unique string representation for each course, for example, the *cid* of *Operating System I* is

cmpt300. The *priority_id* indicates the offering frequency. It is a single-digit number defined in the *Course priority* table (Table 3.5). The higher the value of the digit, the less priority it represents. The column *area_code* indicates the subject area to which the course belongs. A boolean flag *hasPre_req* indicates if the course has any prerequisite, which is for performance only and can be omitted. The column *capacity* stores the number of seats available for an offered course.

Table 3.2: Course table

Field	Type	Default Value
cid	varchar(8)	NOT NULL' '
name	varchar(50)	NULL
type	enum('lower','upper','grad')	NULL
priority_id	char(1)	NULL
area_code	char(2)	NULL
alia_id	varchar(8)	NULL
hasPre_req	enum('1','0')	'0'
capacity	int(11)	NULL

Table 3.3: Course prerequisite table

Field	Type	Default Value
cid	varchar(8)	NOT NULL ' '
pre_cid	varchar(8)	NOT NULL ' '
relation_id	enum('3','2','1','0')	'0'

The *Relation meaning*table (Table 3.4) maps a digit to the definition of a

prerequisite relationship. The *Course area description* table (Table 3.6) lists all subject areas that the Computing Science Department uses to categorize the offered computing courses. The column *id* is the integer representation for an area. It is used for breadth constraints and depth constraints.

Table 3.4: Relation meaning table

Field	Type	Default Value
relation_id	enum('3','2','1','0')	NULL
description	varchar(50)	NULL

Table 3.5: Course priority table

Field	Type	Default Value
id	char(1)	NOT NULL ' '
name	varchar(50)	NOT NULL ' '

Table 3.6: Course area description table

Field	Type	Default Value
id	char(1)	NULL
name	varchar(50)	NULL
nickname	varchar(10)	NULL

Courses offered in the Computing Science Department are also grouped into different tables. Breadth and depth requirements require that students should take courses from different tables as well. Hence, a course mapping

from an area to a table has to be established. *Course area description* table (Table 3.7) maps an *area* to a *table.*

Table 3.7: Course area description table

Field	Type	Default Value
area_code	char(2)	NOT NULL ' '
coursetable	char(1)	NOT NULL ' '

The *Mandatory courses* table (Table 3.8) is used to store mandatory constraints. The "role_id" in the table indicates which type of students the mandatory requirement is for, *e.g.* a mandatory requirement for a major may not be a requirement for a minor. Some mandatory courses have "OR" relation, *e.g.* course 'cmpt101' is required for a computer science major, however if 'cmpt104' is taken, then 'cmpt101' is not required anymore. Then 'cmpt101' has a "OR" relation with 'cmpt104' in the table. The relation only affects the courses that have the same "group_id". The "group_id" is from the *Group* table (Table 3.9).

Courses in the *course* table are divided into groups based on the school regulations. Every rule defined by the school forms a group and has a unique *group_id.* The column *dept* defines who makes the rule and *relation* defines the "*AND/OR*" relationships among rows that are in the same group. The mapping from group id to group definition is in the *Group* table (Table 3.9).

Table 3.8: Mandatory courses table

Field	Type	Default Value
cid	varchar(8)	NULL
relation	char(3)	NULL
group_id	char(2)	NULL
dept	varchar(5)	NOT NULL' '
role_id	varchar(5)	NULL

Table 3.9: Group table

Field	Type	Default Value
group_id	char(1)	NULL
group_name	char(10)	NULL

Chapter 4

Search algorithms and solving techniques

A two-phase approach is applied to generate a final solution iteratively. Two different search algorithms are employed based on the different characteristics of each search phase. A systematic search algorithm is used in phase one to find an initial solution. A systematic local search algorithm is used to find a final solution interactively under the user's guidance. Part of the challenge in curriculum scheduling is handling multiple possible plans, which are equivalent under symmetry. An ordering technique is applied to handle the symmetry so that equivalent infeasible solutions will not be considered more than once. Variable and value ordering is also an important factor impacted on search performance. Thus, in this chapter, related terminologies and definitions used in the algorithms are introduced first, followed by the detailed discussion of the two algorithms. An example is given to explain when and how the system uses the two algorithms during the search. Finally, it is a discussion of symmetry breaking and variable and value ordering techniques used to speed up the search procedure.

4.1 Preliminaries

I would like to define some concepts and coding conventions used throughout this chapter.

Variable ordering: The system has n (see Table 3.1) variables. Those variables are instantiated in a certain order. This is called variable ordering.

Variable: v with either one or two subscripts is used to represent a variable. If a variable has one subscript, the subscript represents its order in the variable ordering. If a variable has two subscripts, the two subscripts represent the row and column positions of the variable in the plan. When we discuss a variable at the solver level, the variable has one subscript, which represents the order of the variable in the variable ordering. When we discuss how a variable is related to a constraint, the variable has two subscripts, which represent the position of the variable in the two-dimension model.

Current variable: The variable currently chosen for instantiation is the current variable. Usually, v_i represents the current variable in the algorithm, unless explicitly expressed.

Variable assignment pair: is a pair (v, d) that contains a variable and a value that has been assigned to that variable.

Variables assignment set: is a set that collects variable assignment pairs. Each pair consists of a variable and its consistent assignment. A is used to represent the collection of all assigned variables and assignments. Sometimes, these assigned variables are also called past variables. A becomes a solution when it contains the assignment pairs for all variables.

Future variables: are variables that have **not** been assigned a consistent value. U is the collection of all future variables.

Live Domain: is a collection of possible domain values that have not yet been shown to be inconsistent with respect to the ongoing search process. D_i is used to represent the current live domain of the current variable v_i.

A label: A set of variable assignment pairs is also called a label.

noGood: is a label, in which some variable assignment pairs are precluded from any global solution.

Culprit: is the variable in a noGood, whose assignment is disallowed. A culprit in a noGood can be any variable in it.

4.2 Modified dynamic backtracking

In this section, a modified dynamic backtracking algorithm is developed to find the initial solution in phase one. I start with justification for the applied algorithm, and then present the algorithm used in the model in detail.

4.2.1 Algorithm justification

A systematic search method is chosen to be used in the first phase because of the characteristics of the curriculum-planning problem. A university freshman always starts with an empty academic record. The record will be filled out semester by semester after the student starts to take courses. Hence, it is intuitive to choose a search method that generates a curriculum plan in the way of mimicking the procedure of a student's taking courses gradually. Also, students graduate when they have achieved all academic requirements for a degree, which indicates many solutions are guaranteed to exist. Thus, a systematic searching method, which starts with an empty variable assignment applying a chronological variable ordering, will be a good search method to find an initial solution for this problem domain. Therefore, a modified Dynamic Backtracking search method is used in the first phase.

The modified Dynamic Backtracking search method employs a backtracking mechanism that is based on Backjumping (BJ) [14] with some inspirations from Ginsberg's [15] Dynamic Backtracking (DBJ). Forward-checking (FC) [14] is added into the search method as well. I will also discuss the justification while explaining the algorithm.

4.2.2 Modified dynamic backtracking

In this section, I present the modified DBT algorithm. It uses a backtracking mechanism that is based on BJ with an improvement inspired by Ginsberg's DBT [15], combined with FC. We add FC into the search algorithm, because it helps to improve the search performance for this application domain. We have mentioned in Chapter 2 that there is a type of system constraint - prerequisite constraints, which is a partial-ordering constraint applied on all variables except those in the first column. They are one-way propagation constraints. Some prerequisite constraints involve over thirty variables and heavily impacts the search procedure. When FC is applied, it can take advantage of the characteristic of one-way propagation of prerequisite constraints to help in preventing an "early" mistake and ensures that every course assigned satisfies the prerequisite constraints. Therefore, the number of backtracks can be reduced. The experimental results presented in Chapter 5 have shown this effect.

It is obvious that for a large problem domain, we cannot use the simple BT, which is too naive and slow. BJ is a more "intelligent" backtracking mechanism and will be a good choice. However, it has one aspect that is not suitable to the planning application domain. In the BJ algorithm, when a backtracking occurs, the search procedure jumps back to the culprit and starts the search from there again. Any work between the culprit and the previous current variable is erased. There may be a great deal of work deleted, which should be avoided in most cases.

It is particularly desired not to erase all the work in the planning domain. For example, suppose that the current variable is in the first row and the third column, and the backjumping point is at the same row but at the first column.

According to BJ, those variable assignments are all removed, although most of them should be kept. This is because courses taken in the same semester usually will not affect each other, which means that the assignment of a variable usually does not affect the assignment of other variables in the same column. Therefore, some of the assignments could be retained. Some care needs to be taken to not erase meaningful work in BJ.

In Ginsberg's DBT [15], it describes a technique that successfully keeps the work, not modifying the values of variables between the current variable and the backjumping variable, by moving the backjumping variable to the end of the partial solution to replace its value. Variable re-ordering is a way of saving the work. However, it is not suited to this application. It is because of the characteristics of the partial-order prerequisite constraints as well. As stated earlier, these constraints propagate in one direction from earlier semesters to later semesters. Thus, when all cells in previous semesters have been assigned, the prerequisite constraint posed on variables in the current semester can propagate effectively.

Prerequisite constraints cannot propagate backwards to prune the domain of variables in previous columns. Therefore, it is strongly desired that the variable ordering be lexicographic ordering. Any other variable reordering that makes the prerequisite constraints hard to propagate is not desired. The approach we decide to take is to backjump to the culprit variable but keep the variable assignments that follow the backjumping spot, and only redo the variable assignment when a constraint violation has occurred.

The algorithm is listed in Figure 4.1. Given a set of variables V, system constraints C, and a static domain D, the algorithm returns the first solution A, which satisfies each constraint in C. The algorithm starts from an empty variable assignment with each variable having the full domain D (Line 1) as the live domain. The loop starting at line 2 is repeated until all variables have been assigned and U becomes empty and then a solution is returned (line 3). While U is not empty, a variable v_i is chosen from U based on the variable ordering rule (line 5). Update the live domain D_i of v_i, and remember the eliminating

explanations R_i for v_i (line 6). R_i remembers reasons of pruning values out of the live domain D_i for v_i(See definition of R_i in Figure 4.1). If the live domain D_i is not empty, forward checking is performed on the future variables, which are those unassigned variables in U. Forward checking is performed at line 8 by trying to instantiate v_i repeatedly until a trail instantiation is found, which ensures no annihilation of the live domain of every future variable. With a successful instantiation of v_i, v_i is removed from U, the assignment of v_i is added into A (line 10 and 11). If a domain-wipe-out occurs when pruning live domain D_i of v_i or when performing forward checking on the future variables, backtracking from v_i to v_j is performed, where v_j is the last assigned variable that occurs in R_i (line 16). If backtracking is needed while the set of eliminating explanations becomes empty, the algorithm returns a failure (line 13).

In our model, a solution is always returned since there exists solutions and the systematic search method is guaranteed to find one.

4.3 Systematic local search

In this section, I describe the algorithm used in the second search phase. It is the algorithm that the system uses to shape the initial solution based on the users' requests. I first discuss the algorithm justification and then present the algorithm in detail.

4.3.1 Algorithm justification

When the system finds the initial solution, it is at a stage where every variable has an assignment consistent with system constraints. Next, it should find a new solution that can keep the consistency of system constraints and at the same time satisfy users' requests. As a MI system, the system should stop the search at a certain time during the process of reasoning, providing the user with the best solution found so far and take the initiative to request further assistance from the user if necessary. A systematic local search algorithm is

V: the set of all variables.

v_i: represents a variable $\in V$ at position i according to the variable ordering.

A: the set stores all variables that have been assigned a consistent value.

U: the set stores all variables that have **not** been assigned a value.

C_i: the set of constraints on variable v_i.

D: the live domain of variables. D_i represents the live domain for variable v_i.

R: the set of elimination explanations. Every element in R is associated with a variable. R_i, remembers all reasons for eliminating certain values from the live domain D_i for the variable v_i. Each reason is represented as a pair composed of a value and a list of variables $d, (v_x, \ldots, v_y)$, where $d \in D,\ v_x \in V,\ and\ v_y \in V$. The intended meaning is that v_i can not take the value d because of the current assignments of variables $v_x, \ldots, v_y$.

```
begin
1.    Set A = φ, U = V, and then set R_x = φ, and D_x = D for ∀v_x ∈ V
2.    Begin loop
3.    if U = φ, returnA.
4.    else{
5.        Select a variable v_i ∈ U according to the variable ordering,
6.        Set R_i based on C_i and A,
7.        update live domain D_i of i based on R_i.
8.        if D_i ≠ φ {
9.            if ∃d ∈ D_i s.t. D_x ≠ φ for ∀v_x ∈ U, then {
10.               Update live domain D_x for ∀v_x ∈ U and v_x ≠ v_i.
11.               Add (v_i, d) to A,
12.               Remove v_i from U, go back to Line 3.
13.           }
14.       }else{
15.           if R = φ, return no solution.
16.           else {
17.              find (v_j, d_j) be the last entry in A, s.t. v_j ∈ R_i.
18.              Remove (v_j, d_j) from A and add v_j to U
19.              Update R_j, and R_x for every variable v_x assigned after v_j.
20.              Set i = j, go back to Line 6.
21.           }
22.       }
23.   }
24.   End loop
end
```


Figure 4.1: Modified dynamic backtracking algorithm

used here instead of the algorithm used in the first phase because this method fits into the second phase of the application domain better.

First, the initial solution provides a perfect starting place for a local search. Second, we observe that a department has hundreds of students and provides a couple of hundred available courses for students to take. The search space is huge, for example, if the number of courses available is 100, and number of variables is 40, then the search space is 40^{100}. We say that two students have one overlap spot in their academic records if they take a course in the same semester. Two records are similar if they have many overlap spots. If two students have taken the same courses at the same semesters together during all school years, which rarely happen, then they have identical records. It is not easy to find two students who have identical curriculum records, but it is easy to find two students whose records are similar and have many overlaps. Students in a department have different academic records, in which there may be many overlaps with the records of many other students. It indicates that many feasible solutions may be clustered together in the search space.

A good initial solution has been found in the first phase, where many other alternative feasible solutions are probably close by. It is reasonable to perform a local search, rather than a systematic search that starts from scratch again with a zero variable assignment. Especially, when user's requests on a proposed plan do not break any constraints, then local search can confirm with much less work - only verifying the variables that have user constraints posted. However, systematic search has to redo the whole planning search.

Furthermore, in the second phase optimization is introduced into the model. It is well known that usually a local search will have much better performance with respect to response time and may reach a far better quality in a given time frame [22]. The reason to extend the model into an optimization problem in the second phase is because we would like to optimize to the number of satisfied user constraints. As the example mentioned in Section 3.1, users may post some constraints that have conflicts with some system constraints, such that no solution exists to satisfy both the system constraints and the

newly posted user constraints. In a case like this, user constraints need to be sacrificed. A solution that satisfies all system constraints and optimizes the number of satisfied user constraint will be a desired one.

When optimization is added, the system returns an optimal solution that has been found so far in a given time frame. Then, the system shows this optimal solution and takes the initiative to ask user's assistance. If the system still applies the same search method used in phase one at this stage and does not add optimization, then it needs to search the complete search tree to return a failure, which probably takes a very long time and is not desired. It usually cannot detect a failure (no solution) in a given time frame. Instead, the system has to stop the search, rejects all user constraints and returns with the previous solution since it fails to find one. Clearly, the approach with the optimization extension behaves more reasonably. Hence, optimization is introduced into the model and a systematic local search method is applied. Experiments are done on both approaches for comparison. More details are discussed in Chapter 5.

In the following subsection, a systematic local search method that is used in phase two is discussed. The systematic local search method combines desirable aspects of systematic backtrack search and heuristic local search.

4.3.2 Systematic local search

In this subsection, I present the systematic local search algorithm (see figure 4.2), which is based on the search method described in [21]. It is mentioned in previous section that optimization is introduced into the model to optimize the user constraints in the second phase. The objective of the optimization is to satisfy as many user constraints as possible while keeping all system constraints satisfied. The objective function is $f(v) = \alpha * s + (1-\alpha) * u$; $where\ v$ represents the full assignments of all the variables, s is the number of satisfied system constraints under the current variable assignments whilst u is the number of satisfied user constraints and finally alpha is a parameter and $\alpha \in (0, 1)$.

The parameter of α is used to control the weight allocated between the system constraint and the user constraint. The assignment of the weight must

be chosen in such a way that the solution must satisfy all the system constraints while take the user constraint into consideration as well. Clearly if α is set to 1, the solution will only try to satisfy all the system constraints while if α is set to 0, the solution will only try to maximize the number of user constraints satisfied. In our implementation, α is set to 0.95. This setting can always ensure that the system constraints are all satisfied and the user constraints are also reasonably satisfied when the algorithm tries to optimize the objective function, as evidently shown in our experiments in Section 5.

The algorithm looks for maximal solutions for a set of variables, which is a subset of all variables. A solution is maximal for a variable set if every variable in the variable set is chosen a maximal assignment. A variable v has a maximal assignment d if the defined evaluation function f, not $\exists a \in D,\ s.t.\ f(d) \leq f(a)$. D is the live domain of v. Every time the algorithm reaches a maximal solution, it verifies that the solution satisfies all constraints. If so, it returns the solution. Otherwise, it keeps looking for the next maximal solution. It remembers the one with the best quality. Because the system is an interactive system, the response time is crucial. Once the predefined search time is exceeded, the current best solution is returned and the system takes the initiative to ask for help with the search from the user.

In the current model, the system takes the initiative based on the searching time only. Actually, the criteria can be much more complex and more practical so that the user can make good decisions easily on the next step. This topic should be more fully discussed. However, this is beyond the scope of this thesis. We will discuss this in the future work in Chapter 6.

The algorithm operates as follows. It initializes the current solution A and the best solution B with variable assignments obtained from phase one. The set of variables V contains the variables that have user constraints. The loop starting at line 2 is repeated until A is a maximal solution for the variable set of V. Then it verifies that A is consistent (line 7). If so, it returns A, otherwise it remembers the better one between A and B, fails A as a noGood (line 8). Then it updates V by adding those variables whose values have been changed

because of the changed assignments of variables in V and goes back to line 3 to look for the next maximal solution (line 9). While A is not maximal, a variable v is chosen based on the variable ordering rule (line 3). Then v is assigned based on the value ordering heuristics and the maximal assignment rule (line 4). Whenever an empty noGood is derived or the predefined search time is up, the current best solution is returned (line 5).

uc: the set of user constraints that currently registered with the user agent.

R: a collection of noGood.

A: the list of variable assignment pairs, initialized with the assignments from the GUI.

B: the best inconsistent solution that has found so far, initially $B = A$.

V: the set of variables that either have user constraints or their assignment have been changed since the last Maximal solution has been found.

```
begin
1.    initialize A, and set B=A
2.    loop
3.        pick v from V and mark v as 'picked';
4.        assign a value d to v;
5.        if an empty noGood is derived or
6.            predefined searching time is exceeded
7.                return B;
7.    end loop when (A is a maximal solution for V)
8.    if A is consistent with all C and uc, return A
9.    else
11.       let B to be the better one between A and B,
12.       set A as a noGood , add it to R;
13.       update V for another loop;
14.       go back to line 3
end
```

Figure 4.2: Systematic local search algorithm

Let's go through an example to illustrate how the two algorithms cooperate.

When a user accesses the system, he or she specifies his or her preference and then takes initiative to ask the system to find a plan. At this stage, the system invokes the search using the modified DBT method. When a feasible solution is found, the system displays the solution and waits for the user's feedback on it. The user inspects the plan, makes some changes and then asks the system to provide a new plan that can satisfy the changes as well as the general constraints stored in the system. This time the system performs the search using the SLS method. When a result is found or predefined termination criteria is reached, the system stops the search and takes initiative to ask user for further assistance. The user has three choices. The first one is to choose to accept the result. The second is that the user makes more changes on the current result and asks the system to generate another one. In this case, the system continues the search using the systematic local search method. The third choice is that user chooses to discard the current result, resets the preferences and asks the system to start again. In this case, the system goes back to phase one and uses the modified DBT algorithm to perform the search.

4.4 Other techniques

In this section, the technique used to overcome the symmetry in the planning problem is described and followed by the discussion of the importance of variable and value ordering during the problem solving and what variable and value ordering are used.

4.4.1 Symmetry breaking

The curriculum-planning problem, like many other scheduling problems, encounters the symmetry problem as well. There is symmetry among courses in the column. Rows are undistinguishable and can be freely permuted in a semester. In Chapter 2, two general ways of handling symmetry problems are discussed. The approach of defining symmetry versions of constraints requires adjustment of the search method and is hard to apply to application domains

that have many complex constraints. Because we have only one-dimension symmetry in the planning problem, it is more straightforward to take the other approach of adding constraints to the CSP to convert the problem into an asymmetrical one. The symmetry constraints are treated as the system constraints as well. By taking this approach, we do not need to make any changes on the search algorithm .

The method involves adding a partial-order constraint on the variables that are in the same column to break the row symmetry. Simply adding the lexicographical ordering constraints on symmetry breaking does not work well. Lexicographical constraints can solve the symmetry problem but they may hinder the search in some cases in this application. A detailed explanation is discussed below. Hence, adding partial-ordering constraints onto the variables in columns breaks the symmetry occurred in the curriculum-planning problem.

In the model, the courses in the variable domain are grouped into classes according to their subjects. Each class has an associated value, which is used to determine if a course has a higher order than another one. For example, all mathematics courses belong to the class of 'math.' All computer science courses belong to the class of 'cmpt.' The system defines that the 'cmpt' class has a higher order than the 'math' class. Thus, a computer course has a higher order than a math course.

The symmetry-breaking constraint enforces the partial ordering on variables in a column by the class value of the variables rather than the lexicographical ordering by domain values. This is because the lexicographical constraints may be too strict under certain circumstances and can hinder the search by causing unnecessary backtracks in the second search phase.

For example, suppose that the symmetry-breaking constraint directly uses domain values to evaluate the ordering on those variables. There are three variables in the column i (v_{1i}, v_{2i}, v_{3i}), four domain values with $d_1 < d_2 < d_3 < d_4$, and three classes $A_1 < A_2 < A_3$, where $d_1 \in A_1$, $d_2 \in A_2$, *and* $d_3, d_4 \in A_3$. The three variables have the assignment pairs of $\{(v_{1i}, d_1), (v_{2i}, d_2), (v_{3i}, d_3), \ldots\}$ for a column i, where $1 \leq i \leq p$. When the user changes the assignment of

the variable v_{2i} from one course d_2 to d_4, provided no system constraints are violated by the change, the partial order on the three variables evaluated by domain values ($d_1 \leq d_4 > d_3$) is broken. Thus, the constraint is violated after the change, and then backtracking has to be performed. However, the ordering on class values ($A_1 \leq A_3 \leq A_3$) exists, and the backtracking can be avoided if the constraint uses class values to evaluate the partial ordering on variables.

Hence, the better approach to break the symmetry is to let symmetry-breaking constraints use class values to check the ordering on variables in a column. In this way, the efficiency of pruning the search space is ensured, meanwhile, unnecessary backtracks are avoided.

4.4.2 Variable and value ordering

It is known that, when using symmetry-breaking constraints, the variable and value ordering are very important [24]. In particular, if variable ordering moves from a direction that increases conflicts with the symmetry-breaking constraint, we can expect to gain from both the lower complexity and increased pruning [24]. Also, there are system constraints such as prerequisite constraints, preferring special variable ordering, aside from the fact that variable ordering affects the search performance significantly. Therefore, variable ordering should be carefully selected and maintained in this application.

In the model, variables are grouped by semesters and ordered from top to bottom within a semester, and chronologically among semesters in phase one. In phase two, variables are divided into two groups. One group contains variables that have user constraints posted and has higher ordering. The other group contains variables that have no user constraints posted, thus has lower ordering. Inside each group, variables are ordered lexicographically.

When selecting values for variables, the value ordering heuristics are applied first. The heuristics are defined based on the variable position and related user information. Heuristics simulate what a human advisor would suggest. When students make their plans manually, they usually choose courses under the guidance of an academic advisor. These rules are transformed into value

ordering heuristics stored in the system to guide the search. For those variables that have user constraints, the value ordering will be little different. The value specified by the user has the highest priority.

When there are no related value-ordering heuristics to apply, values in a live domain are ordered as follows. Courses in a variable's live domain are divided into different classes. If two courses are in the same class, these two courses are ordered by the value of their integer representations; otherwise, the course that has a lower class level is less than the one that has a higher-class level. Through the experimental results presented in Chapter 5, we found that it can speed up the search.

Chapter 5

Experimental results

The model is implemented into a Web Service using Java [1], JSP [2] and Servlets [3] on Apache Tomcat server [4]. JSP and JavaScript handle the presentation layer - the interface of the model. The Servlets handle the navigation flow and data access. The system process (the service part) is implemented in Java. The program also uses a Java-based Constraint Programming (CP) library called ConstraintWorks [30], which provides the procedure of the systematic local search algorithm. The database is implemented in MySql server [5]. The

[1] Java technology is a portfolio of products that are based on the power of networks and the idea that the same software should run on many different kinds of systems and devices. JDK1.3.2 is used in implementation. Related documentation can be found at http://java.sun.com/j2se/1.3/docs/api/.

[2] JavaServer Pages (JSP) technology provides a simplified, fast way to create dynamic web content. It enables rapid development of web-based applications that are server and platform independent. JSP specification is developed by Sun Microsystems, which can be found at http://java.sun.com/products/jsp/.

[3] Java Servlet technology was created as a portable way to provide dynamic, user-oriented content. Their specifications are developed by Sun under the Java Community Process. Detailed information is available at http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html.

[4] Apache Tomcat is the Servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. Binaries and documentations can be found at http://jakarta.apache.org/tomcat/.

[5] MySql database server is an open source database. It can be downloaded at http://dev.mysql.com/downloads/.

application uses JDBC [6] driver provided by MySql [7] to connect the database.

The experimental data is from the Computing Science Department at Simon Fraser University (SFU), British Columbia. All course related information stated in the calendar, which includes course name, credit hours, subject area, prerequisites, equivalent courses and course level (upper or lower division), is stored in the testing database. Other requirements needed to finish a degree such as mandatory upper or lower division requirements, and breadth or depth requirements for different types of programs offered in the School of Computing Science are also defined clearly in the testing database. The testing database includes available courses from the school calendar that computer science students may take. There are plenty of courses in SFU school calendar that computer science student would never take, and thus, those courses are excluded from the database. The implemented system limits its service to undergraduate students in computing science only.

Users used in the testing are fabricated and so is any user related information. There are total 9 types of typical computer science students in the School of the Computing Science at SFU: students who are major in computing science and interested in five different subject areas (Artificial Intelligent, Graphics, Network, Database and Theory); students intended to have a math minor; students intended to have a business minor; students intended to complete the co-op program (Students have to finish at least four co-op terms during their undergraduate study in order to complete the co-op program); students intended to graduate with software engineer special program. Hence, I created 9 made-up users, one for each typical type of user group. Test cases are run against these nine users.

In addition to computer science courses, available courses stored in the testing database include elective courses from many departments, such as Art,

[6]Java Database Connection (JDBC) technology is an API that provides cross-Database Management System connectivity to a wide range of SQL databases and access to other tabular data sources, such as spreadsheets or flat files

[7]MySql provides a native JDBC driver called MySQL Connector/J. The binaries and documentation can be found at http://www.mysql.com/products/connector/j/.

English, Economics, Business, Natural science, Philosophy and so on. The maximum number of available courses used for experiments is 120 courses, among which 48 are computing science courses. The remaining courses are from departments mentioned above. The number of available courses used as the static domain for variables varies from 60, 80 to 120. The model is validated in following four aspects.

1. Compare the performance of with or without FC during the search.

2. Compare the performance of with or without symmetry breaking.

3. Perform experiments with different domain sizes, number of constraints on different systematic search method to show the modified DBT with heuristics has the best performance overall.

4. Compare the performance in the second phase. The systematic local search can reach a better solution with respect to search time than the systematic search method in most of the test runs.

Experiments were run on a windows machine with Intel Pentium 4 Mobile CPU 1.60 GHz and RAM 512 MB. When the result is measured by searching time (seconds), the time is rounded to whole seconds, i.e., if it takes less than one second, it counts as one second. The experiment configuration used to verify the performance of using FC is as follows. The domain size of a variable m [8] is 60. The total number of system constraints is 28. The search method is the systematic search method with no heuristics applied. A total of 18 test runs were performed against all typical users. Results of test runs are plotted into charts (Figure 5.1 and 5.2). The X-axis represents test runs for individual users. The Y-axis is either the number of backtracks or the search time in seconds.

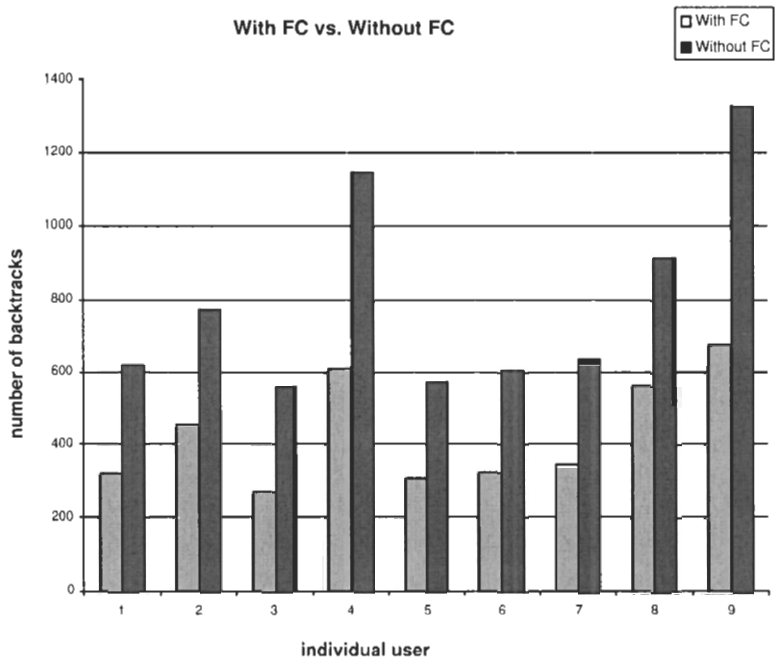

Figure 5.1: Number of backtracks comparison chart of searching with or without FC with the domain size of 80

5.1 Forward checking

The experimental data of applying FC and not applying FC is measured in the number of backtracks against test runs for each individual user, and is plotted into a chart (see Figure 5.1). The comparison is performed under another measurement - search time as well. The test results are plotted in Figure 5.2. All Test runs were performed in an incomplete system environment

[8]m: the domain size of variables, see table 3.1

with partial system constraints posted. The charts from both Figure 5.1 and Figure 5.2 show that the method with FC applied has more net savings in both the number of backtracks and search time. In some test runs, the savings were about approximately 2 times or more even. The results indicate that applying FC into the search procedure can improve the search performance and is suitable for the course planning application.

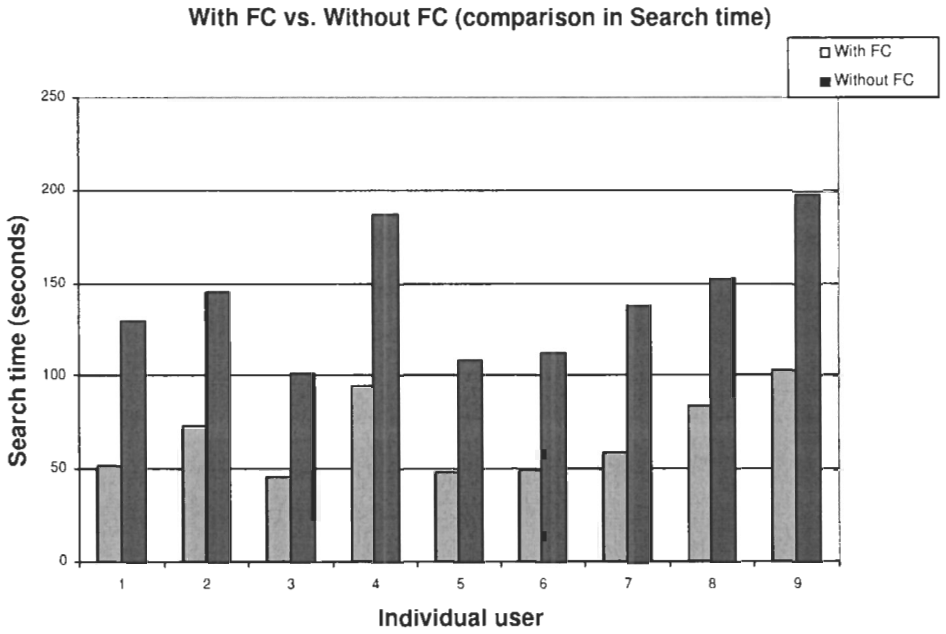

Figure 5.2: Search time comparison chart of searching with or without FC with the domain size of 80

5.2 Symmetry breaking

Next, we look at the experimental results of breaking or not breaking symmetry. The experiment configuration used to verify the performance of using symmetry breaking is as follows. The domain sizes are 60, 80 and 120. The

Table 5.1: Performance evaluation on symmetry breaking

Domain size	With symmetry breaking # of backtracks	search time (seconds)	No Symmetry breaking # of backtracks	search time (seconds)
60	217	33	385	59
80	372	51	524	76
120	841	110	1537	249

total number of system constraints is 28, excluding the symmetry-breaking constraints. The maximum number of course load is 12 credit hours. The search method is the systematic search method with no heuristics applied. This test case ran against all 9 users. A typical set of the experimental results is listed in the table 5.1. They are for the user who is an intended major in computer science and interested in the database area.

Instead of listing the experimental data on this test case for the remaining 8 users into 8 tables, the data are plotted into the two charts (see Figure 5.3 and Figure 5.4). The first chart (Figure 5.3) shows the comparison of the number of backtracks used when the symmetry breaking technique is applied versus when symmetry breaking is not applied and the number of backtracks was less when the symmetry technique was used in the test runs. The second chart (Figure 5.4) shows the comparison of the search time. The runs, in which the symmetry breaking was applied, used less search time. From the two charts, it clearly shows that the search method with the symmetry-breaking technique has better performance with respect to the number of backtracks and the search time. Hence, we can conclude that it is better to apply symmetry breaking techniques in the curriculum problem solving for good performance.

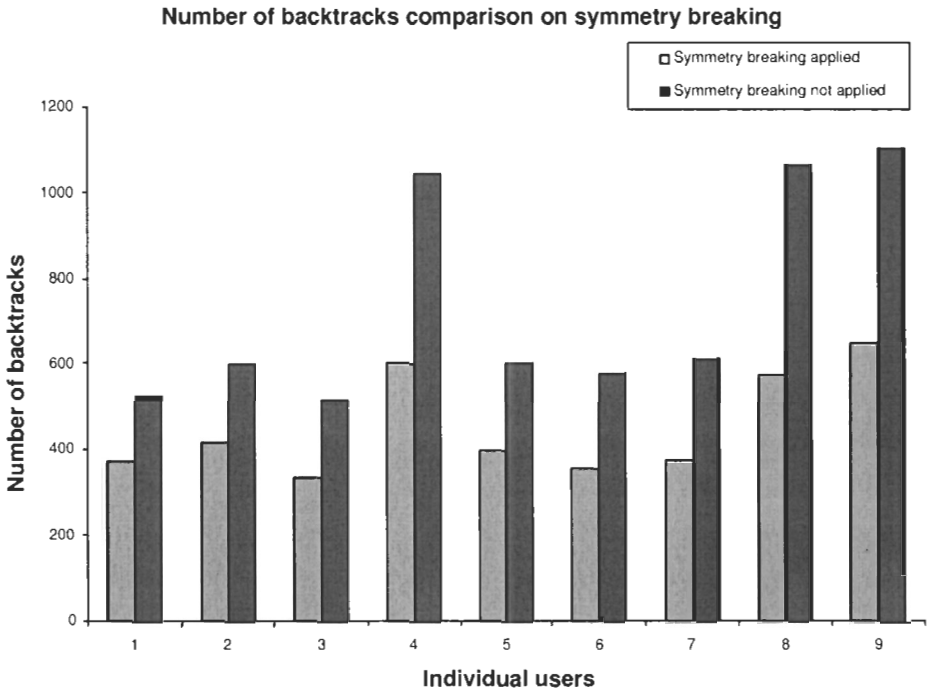

Figure 5.3: Number of backtracks comparison chart on symmetry breaking techniques with the domain size of 80

5.3 Comparison of search methods

Next, we inspect the performance comparison among three different search methods: systematic search with no heuristics, systematic local search, and the systematic search with heuristic. Experimental results are presented in Tables 5.2, 5.3, and 5.4. All system constraints are loaded into the system for this set of test runs. I compared the performance of the three different search methods at a domain size changing from 60, 80, up to 120. A total of 18 test runs were executed to compare the performance among the three search methods. Similar test results were collected. One set of test results is listed in the tables below (5.2, 5.3, and 5.4). We use the number of backtracks and the number of iterations to evaluate the performance of the method. The

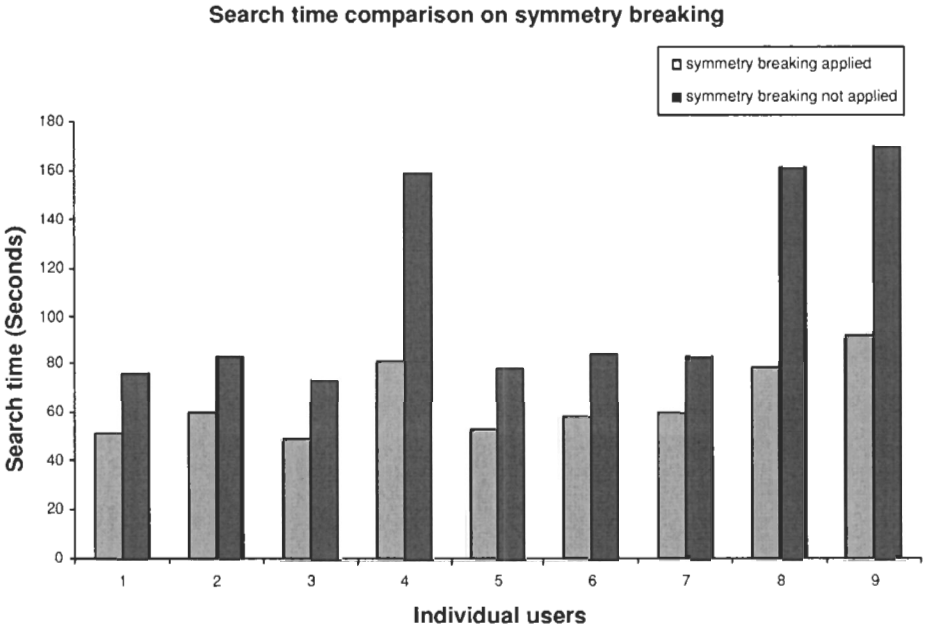

Figure 5.4: Search time comparison chart on symmetry breaking techniques with the domain size of 80

number of backtracks is the number of times trying to un-assign an assigned variable. The number of iterations is the number of times trying to assign a future variables during the search. The three tables (5.2, 5.3, and 5.4) show that as domain size increases, the number of backtracks and the number of iterations of both the local search method and the systematic search method with no heuristics increase dramatically. In contrast, the systematic search with heuristics can find a solution without backtracking in all 3 cases. It maintains good performance as the domain size increases. Hence, results confirm that a systematic search method needs good heuristics to have good performance, especially when the size of the domain becomes large.

Table 5.2: Search method performance comparison with m=60

Search method	# of backtracks	# of Iterations
Systematic search with no heuristic	7189	4435
Systematic search with VOH[9]	0	39
Local search with no heuristic	740	10

Table 5.3: Search method performance comparison with m=80

Search method	# of backtracks	# of Iterations
Systematic search with no heuristic	12991	8997
Systematic search with VOH	0	39
Local search with no heuristic	6740	983

5.4 Phase two performance evaluation

As for the second phase, we study the performance of local search method when the user's requests break different numbers of constraints at different positions in the schedule. A position in a plan is defined as follows: for a plan with p number of columns, a slot S is at column c with $c/p < 30\%$, then we say S is at an early position of the plan; if $30\% < c/p < 70\%$, then S is posted at a middle position, otherwise it is posted at a late stage of the schedule. In order to compare the performance of the systematic local search method, we also run the second phase using the search algorithm from the first phase.

Table 5.4: Search method performance comparison with m=120

Search method	# of backtracks	# of Iterations
Systematic search with no heuristic	99760	31216
Systematic search with VOH	0	39
Local search with no heuristic	11811	5585

Where the second phase is solved only as a CSP, the algorithm either returns a solution satisfying all constraints including the newly added user constraints or a failure. In the case of a failure, the system keeps the old solution and discards all newly added user constraints. The experiment configuration used to evaluate the performance of systematic local search (SLS) and systematic search (modified DBT search) at the second stage is as follows. The number of courses available is 120. Maximum course load is 12 credit hours. The search time limit for the systematic local search method is 3 minutes. The following cases are tested on all 9 users. The precondition of running these experiments is that the system has entered phase two and it has proposed an initial plan.

- Test case one: Modifications made in the proposed plan do not break any constraints, *e.g.* replacing an English course with a Philosophy course, which does not violate any system constraint. Make this type of change at an early, middle or late position respectively.

- Test case two: Modifications made in the proposed plan break one constraint. For example, replacing a mandatory computer course with a non-mandatory computer course. Thus, a mandatory constraint is broken, provided the newly chosen computer course does not break any other constraint. Then, the system needs to fix this broken constraint by replacing this mandatory course at some other place in the plan with all constraints satisfied. Another typical test case is like this: the plan suggests a course in the third semester, a change is made by the user to take that course in the second semester provided there is no prerequisite or other constraint violated. This change leads to two cells having the same value. Thus, the all-different constraint is broken and the system needs to provide a good suggestion to reassign a new course to the slot in the third semester to repair the all-different constraint. Make this type of change that violates either a mandatory or (exclusive or) the all-different constraint at an early, middle or late position in the plan.

- Test case three: Modifications made in the proposed plan break two constraints, *e.g.* change the contents of two slots in the proposed plan, which breaks one prerequisite and one mandatory constraints. Choose the two slots from two early positions, two middle positions and two late positions respectively.

- Test case four: Modifications made in the proposed plan break three constraints, *e.g.* change the contents of three slots in the proposed plan so that a prerequisite, a mandatory and the all-different constraints are broken. Choose the three slots at three early positions and late positions respectively.

- Test case five: Changes made at two slots in the proposed plan break three constraints, a prerequisite, an equivalent and the all-different constraints. The change is to request taking two equivalent courses, which should not be satisfied. Choose the two slots at two middle positions. This is a special case.

Table 5.5 lists the test results for a user who is a computer science major and wants to have a business minor. We observe that when the user's requests (the modifications committed by the user) do not violate any system constraints, the SLS search returns the confirmation within a second and the modified DBT method takes 11 seconds to return. The SLS is much faster. When the user requests break from one to three constraints, local search has shorter search time than the systematic search in seven out of nine cases. For the special case that no feasible plan can satisfy the user's requests, it takes the systematic method a very long time to return a failure and report the culprit user constraints, which really are the two modifications made to take two equivalent courses. The SLS method stops with one user constraint satisfied when the predefined search time is up. When the systematic method stops searching at the predefined time, it stops with the old solution returned but no report of finding culprit user constraints. Instead, it rejects all user constraints. SLS reaches a better solution in the given time frame.

Table 5.5: Performance comparison between the systematical Local search and the modified DBT search

# of broken constraints	Position in the plan	Systematic Local Search		Modified DBT search	
		# of iterations	time used (seconds)	# of iterations	time used (seconds)
0	Early	2	1	39	11
	Middle	3	1	39	11
	Late	4	1	39	11
1	Early	222	39	1222	171
	Middle	31	44	910	79
	Late	10	16	62	10
2	Early	1838	151	1638	193
	Middle	40	13	379	64
	Late	194	25	70	11
3	Early	13	9	93	29
	Middle	2024	180	8324*	6478
	Late	3	2	3	4

*note: In this test case, we did not let the systematic search method stop at the predefined search time limit, because we would like to see how long to take it to find a failure.

The first four sets of test cases listed above were run against the remaining 8 test users as well. Thus, the total number of users is nine. Each user runs four sets of test cases at three different positions. The total number of test runs is 108. Instead of listing all experimental data in a table for the remaining 8 users, I plotted the data into figures so that we can compare the performance with respect to search time between the two methods more easily. Three figures are included below. Figure 5.5 displays the time used by two methods for 9 different users when users make changes at early positions in the plan. Data used to plot Figure 5.5 are the result of running the second set of test cases (changes made break one system constraint). The SLS method used less search time in 9 out of 9 runs. Figure 5.6 shows the time used by two methods for 9 different users when users make changes that breaks two system constraints,

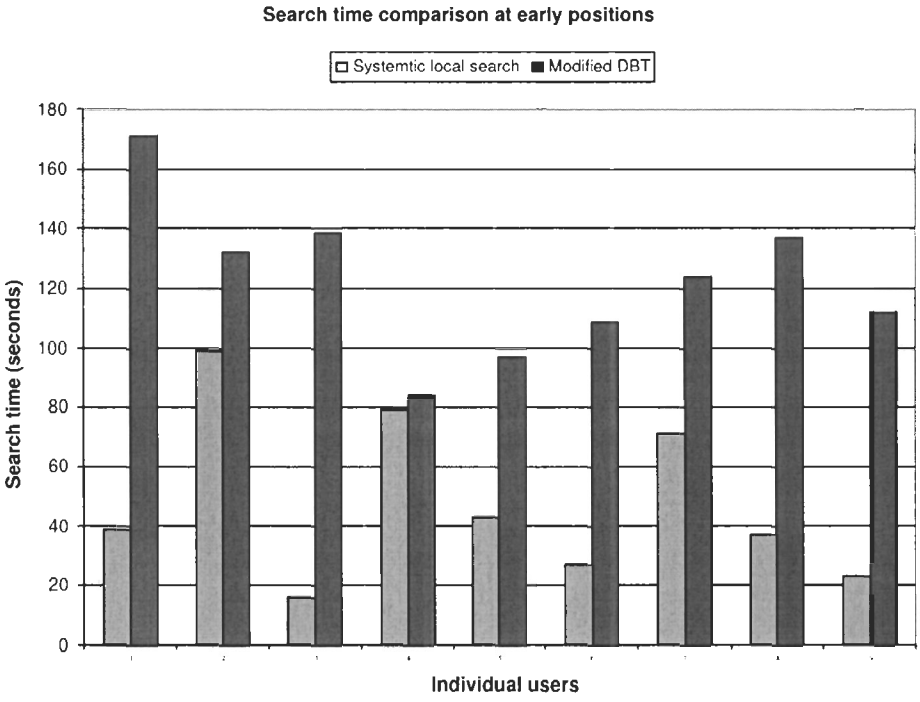

Figure 5.5: Search time comparison at early positions

at middle positions in the plan. The SLS method outperformed the modified DBT method in 8 out of 9 runs.

Figure 5.7 plots the time used by two methods for 9 different users when users make changes at late positions in the plan and the test runs fell in the category of the test case four (the changes break three system constraints, the special case is not included). The SLS method used less time in 7 out of 9 runs. Overall, there were 84 out of 108 runs that SLS solved problem and returned the answer within a minutes. There were 79 out of 108 that the SLS method used less search time to return the solution than the modified DBT.

Hence, from above experimental results, it is observed that the SLS method usually is able to solve the problem within a minute in most of cases and overall has better performance with respect to the response time than modified DBT.

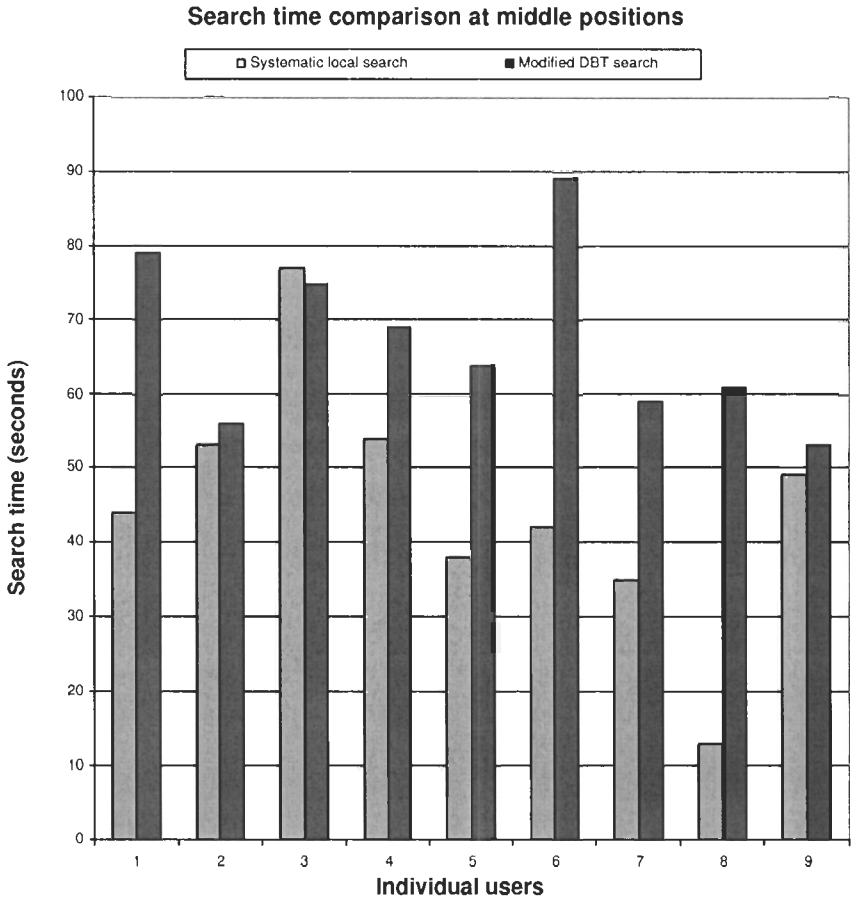

Figure 5.6: Search time comparison at middle positions

The SLS method can return a better quality solution within the given time frame in the case that user requests make the problem over constrained.

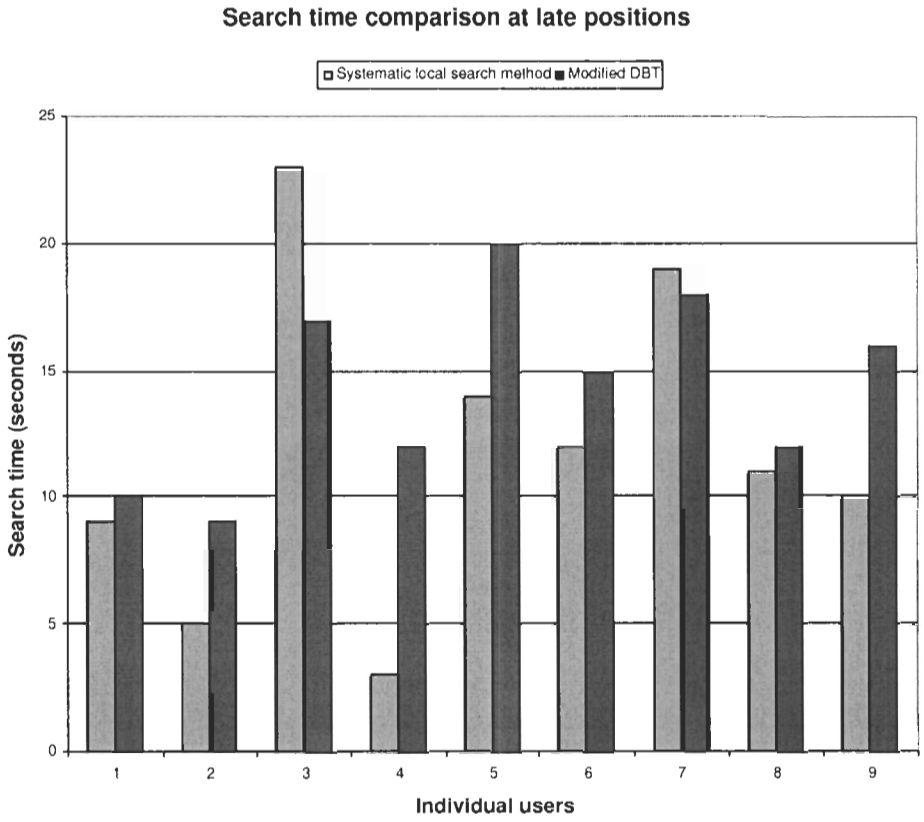

Figure 5.7: Search time comparison at late positions

Chapter 6

Conclusion

6.1 Summary

In this thesis, a two-phase CSP based reasoning model is developed to solve the curriculum-planning problem. Due to the diversity in the characteristics and preferences of different users, mixed-initiative is integrated into the planning system, which provides a direct-manipulation environment for the efficient communication between a user and the system. Through a cycle of plan construction, revision, and improvement, the mixed-initiative system can serve various users effectively.

The CSP based planning model solves the problem using two phases. A backtrack-based systematic search method is used in the first phase to produce an initial solution. In the second phase, the user can take the initiative to post requests and ask for new solutions. Then the system starts to search. It either returns a solution or asks for help with an optimal solution, which some user constraints are not satisfied. It depends on the user to take further actions. The systematic local search method is employed to construct new plans iteratively, under the interactive guidance from the user.

Because symmetry exists in the course-planning problem, an approach of adding partial-ordering constraints is employed to break the symmetry arising in the model. Value ordering heuristics are used as well to speed up the search.

Results from the experiments with actual curriculum data are promising and show that the system generates effective curriculum plans efficiently.

Experiments with actual course planning data illustrate that constraint programming techniques with proper variable ordering and value ordering heuristics can yield impressive results when solving curriculum-planning problems. The promising results indicate that it is a proper approach to model the curriculum-planning problem into a CP substrate and solve it using CP techniques. The success of modeling the planning problem into a CSP demonstrates connections between planning and constraint satisfaction as well.

6.2 Future work

One idea for future work is to determine what the better criteria could be used for the system to stop the search in the second phase. Should all solutions found be returned to the user or only the optimal one be returned? To solve these questions, it requires further investigate and also more input from users. Indeed, sometimes a "worse solution" may be easier for a user to correct or perhaps even preferred by the user. The knowledge and ability of individual users should be taken into consideration to solve these questions and improve the system.

The response time collected in the second phase during the experiments is a little bit long for an interactive system. Another place for future work would be considering to include compilation methods that transform the original problem into a data structure to allow a short response time for interactive solving [31].

Since the goal of the thesis is not to discuss how to build software to solve the curriculum-planning problem but concentrate on presenting an efficient CP model. Thus, scalability, generalization and deployability are beyond the scope of this thesis and can be considered as future work.

Bibliography

[1] K, Baker. Elements of Sequencing and Scheduling. Wiley, 1998.

[2] M. Pinedo. Scheduling Theory, Algorithms and Systems. Prentice Hall. 1995.

[3] D.E. Smith, J. Frank and A. K. Jonsson. Bridging the gap between planning and scheduling. Knowledge Engineering Review, Vol. 15(1) 2000.

[4] N. Alexander and K. Subbarao. Introduction to the Special Issue on Planning: Research Issues at the Intersection of Planning and Constraint Programming. Constraints, Vol. 8(4) (1997) pp. 99-118.

[5] E.P.K. Tsang. Foundations of Constraint Satisfaction. Academic Press: London and San Diego. 1993.

[6] A. Schaerf. A survey of automated timetabling. Artificial Intelligence Review, Vol. 13(2) (1999) pp. 87-127.

[7] C. Castro and S. Manzano. Variable and Value Ordering: When Solving Balanced Academic Curriculum Problems. In *Proceedings of 6th Workshop of the ERCIM WG on Constraints*, Prague. June, 2001.

[8] B. Hnich, Zeynep Kizitan, and T. Walsh. Modelling a Balanced Academic Curriculum Problem. In *Proceedings of CP-AI-OR02*, 2002.

[9] M. Walker. and S. Whittaker. Mixed-initiative in dialogue: an investiation into discourse segmentation. In *Proceedings of ACL90*, (Pittsburgh, PA, 1990) pp. 70-76.

[10] C. Guinn. Mechanisms for mixed-initiative human-computer collaborative discourse. In *Proceedings of ACL96*, (Santa Cruz, CA, 1996) pp. 27-205.

[11] J. Allen. Mixed-initiative planning: position paper. Presented at ARPA/Rome Labs Planning Initiative Workshop, 1994.

[12] B. Miller. Is explicit representation of initiative desirable? Working Notes of AAAI97 Spring Symposium on Mixed Initiative Interaction. Stanford, CA, 1997.

[13] M. Burstein and D. McDermott. Issues in the development of human-computer mixed-initiative planning. In B. Gorayska & J. L. Mey(Ed.), *Cognitive Technology: In Search of A Human Interface*, (North-Holland, 1996) pp. 285-303.

[14] P.Prosser. Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence, Vol. 9(3) (1993) pp. 268-299.

[15] M. L. Ginsberg. Dynamic backtracking. Journal of Artificial Intelligence Research, Vol. 1 (1993) pp. 25-46.

[16] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, Vol. 14 (1980) pp. 210-215.

[17] R. Dechter. Enhancement schemes for constraint processing: backjumping, learn- ing, and cutset decomposition. Artificial Intelligence, Vol. 41(3) (1990) pp. 273-312.

[18] R Backofen and S. Will. Excluding Symmetries in Constraint-Based Search. Constraints, Vol. 7 (2002) pp. 333-349.

[19] M. Johnston, A. Philips, and P Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence, Vol. 58 (1992) pp. 161-206.

[20] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of fourth Conference on Principles and practice of Constraint Programming*, (Pisa, 1998) pp. 417-431.

[21] W. S. Havens, Bistra N. Dilkina. A Hybrid Schema for Systematic Local Search. In *Proceedings of Canadian Conference on AI 2004*, (London, ON, Canada, 2004) pp. 248-260.

[22] N. Jussien, and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. Journal of Artificial Intelligence, Vol. 139 (2002) pp. 21-45.

[23] P. Flener, A. Frisch et. al. Breaking row and column symmetries in matrix models. In *Proceedings of CP'2002*, (Springer, 2002) pp. 94-107.

[24] A. Frisch, B. Hnich et. al. Global Constraints for Lexicographic Orderings. Constraint Programming, (2002) pp. 93-108.

[25] I. Gent, and B. Smith. Symmetry breaking during search in constraint programming. In *Proceedings of ECAI 2000*, (2000) pp. 599-603.

[26] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *Proceedings of the AAAI-92 Workshop on Tractable Reasoning*, 1992.

[27] P.Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel and T. Walsh. Matrix modelling. In *Proceedings of Formul'01, the CP')1 Workshop on Modelling and Problem Formulation*, 2001.

[28] J. F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Proceedings of ISMIS'93*, (Springer-Verlag, 1993) pp. 350-361.

[29] F. Glover and M. Laguna. Tabu search, in: *C. Reeves (Ed.), Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publishing*, (Oxford, 1993) pp. 70-141.

[30] Actenum Corporation, ConstraintWorks, Vancouver, British Columbia, Canada. http://www.actenum.com.

[31] H. Fargier and M. Vilarem. Compiling CSPs into Tree-Driven Automata or Interactive Solving. Constraints, Vol. 9 (2004) pp. 263-287.