Titre: Title:	Examination scheduling by constraint programming
Auteur: Author:	Xiangxiu Yang
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Yang, X. (2001). Examination scheduling by constraint programming [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6971/

UNIVERSITÉ DE MONTRÉAL

EXAMINATION SCHEDULING BY CONSTRAINT PROGRAMMING

XIANGXIU YANG

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

AVRIL 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

EXAMINATION SCHEDULING BY CONSTRAINT PROGRAMMING

présenté par : XIANGXIU YANG

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme FARIDA CHERIET,	Ph.D., présidente
M. LOUIS GRANGER,	M.Sc., membre et directeur de recherche
M. GILLES PESANT,	Ph.D., membre et codirecteur de recherche
M. GILBERT LAPORTE,	Ph.D., membre

DEDICATE

To my family

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Professor Louis Granger, my supervisor, for his encouragement, patient guidance, and valuable advice.

I would like to express my special gratitude to Professor Gilles Pesant, my co-director, for his encouragement, support and technical guidance. Without his knowledge and experience, this thesis would not have been possible.

I would like to thank all of the enthusiastic and helpful friends in the constraint research group, for making the past two years memorable.

Finally, I would like to thank my family for their love and support.

RÉSUMÉ

Le problème de confection d'horaires d'examens est un problème combinatoire NP-difficile. Il devient un important défi pour la gestion des opérations des écoles et des universités puisque, à la fin de chaque session ou année, la majorité des institutions d'enseignement doivent planifier la tenue d'un ensemble d'examens. Avec l'expansion de la taille des problèmes et l'augmentation de la complexité des contraintes, la résolution de ce problème nécessite l'investissement d'un effort beaucoup plus grand.

L'objectif de ce mémoire est le développement d'une nouvelle application permettant de résoudre le problème de confection d'horaires d'examens pour l'École Polytechnique de Montréal; l'allocation automatique de locaux pour les examens est aussi incluse. L'algorithme de cette application doit satisfaire l'ensemble des contraintes dures ainsi que le plus grand nombre possible de contraintes molles; la recherche de solution doit de plus être efficace.

Dans le cadre de ce mémoire, la programmation par contraintes constitue la méthode de résolution retenue. Le modèle du problème est tout d'abord construit à l'aide d'une division en deux sous-problèmes: le problème de confection d'horaires des sessions d'examen et le problème de confection d'horaires des locaux d'examen. Les deux problèmes sont résolus indépendamment; dans les deux cas, plusieurs stratégies de recherche, tant statiques que dynamiques, sont développées pour la sélection des variables et des valeurs. Pour une meilleure performance, différentes méthodes sont

proposées permettant de réduire l'espace de recherche aux seuls sous-espaces réellement explorés durant la recherche de solutions. De plus, différents algorithmes contrôlant la recherche sont utilisés pour améliorer davantage la performance et déterminer les endroits où les mauvais choix risquent le plus d'être effectués. Dans ce mémoire, le langage Ilog Solver est l'outil utilisé pour résoudre le problème de confection d'horaires d'examens.

Les résultats expérimentaux montrent que plusieurs algorithmes sont en mesure de trouver de meilleures solutions et que l'un d'eux comporte une exécution plus performante. Cet algorithme est très stable; nous l'avons testé à l'aide de trois ensembles de données. Nos résultats font très bonne figure lorsque comparés à ceux de l'École Polytechnique.

ABSTRACT

The examination scheduling problem is very difficult to be solved and is classified as NP-hard. It is becoming an important operations management problem in schools and universities. At the end of each term or year, most educational institutions must schedule a set of examinations. With the expansion of the size of the problem and with the increase of more complicated constraints, the problem becomes even more difficult and much effort needs to be put into it.

The aim of this thesis is to develop a new application to solve the examination scheduling problem for École Polytechnique de Montréal. The algorithm of the application should satisfy all hard constraints, satisfy soft constraints as much as possible and be efficient in finding better solutions. Also it should allocate rooms for each examination automatically.

This thesis adopts constraint programming as the method to solve the problem. First the model of the problem is constructed, then the problem is divided into two sub-problems, and solved in two phases: examination-session timetabling problem and examination-room timetabling problem. For each sub-problem several search strategies including both static and dynamic rules for variable and value selection are investigated. In order to improve the run-time performance, different methods are investigated to reduce the search space so that only the right sub-spaces are explored during the search.

Additionally, different search control algorithms are used to improve the performance further and to find out where the algorithms are likely to make a wrong choice. In this thesis, ILOG SOLVER is used as our tool to solve the examination scheduling problem.

The results of experiments show that several algorithms could find better solutions and one of them had a better run-time performance. The algorithm is very stable, we tested it by using three sets of data. Comparing with the results from École Polytechnique de Montréal, it would seem that our results are very good.

CONDENSÉ EN FRANÇAIS

I. INTRODUCTION

La portée pratique des problèmes d'ordonnancement est énorme. Professeurs et conférenciers doivent organiser leurs présentations, l'organisateur d'un voyage doit gérer les horaires de train et d'autobus, ainsi de suite. De tels problèmes sont très ardus à résoudre et appartiennent à la classe des problèmes NP-difficiles. Les difficultés inhérentes à un problème d'ordonnancement typique proviennent de sa grande échelle, du grand nombre d'exigences, de contraintes et de critères de qualité souvent contradictoires. Bien que de nombreux efforts furent dépensés sur ce sujet, les problèmes d'ordonnancement demeurent un défi pour la recherche opérationnelle et l'informatique. Le problème de confection d'horaires d'examens constitue une sous-classe de ces problèmes. Il devient un important défi pour la gestion des opérations des écoles et des universités; de nombreux efforts méritent donc d'y être consacrés.

Les méthodes habituelles utilisées pour résoudre ce type de problème proviennent du domaine de la recherche opérationnelle, comme les algorithmes génétiques et les procédures de recherche locale. Ces méthodes bien connues sont couramment utilisées et permettent d'obtenir de bons résultats; elles possèdent par contre un défaut inhérent, leur manque de flexibilité. Récemment, la programmation par contraintes est devenue une alternative intéressante pour résoudre les problèmes d'ordonnancement. Elle combine les avantages à la fois de la résolution de contraintes et de la programmation logique et

permet d'obtenir des programmes à la fois expressifs et flexibles. La preuve de l'efficacité de la programmation par contraintes pour résoudre de nombreux problèmes combinatoires n'est plus à faire. En fait, elle constitue un outil très pratique permettant de construire des applications lorsque aucun algorithme général n'est disponible ou que des changements fréquents sont prévus.

Pour résoudre leur problème de confection d'horaires d'examens, l'École Polytechnique de Montréal utilise à l'heure actuelle un programme développé il y a une quinzaine d'années. Ce programme ne semble plus à l'heure actuelle approprié pour attaquer le problème; entre autres, il ne permet pas d'effectuer l'allocation des locaux pour les examens. Au cours de ce projet, une méthode de programmation par contraintes sera utilisée pour générer des horaires d'examens incluant l'allocation des locaux.

II. DESCRIPTION DU PROBLÈME

Le problème de confection d'horaires d'examens repose sur ces deux contraintes fondamentales:

1. Aucun étudiant ne doit avoir plus d'un examen au même moment.
2. Un local ne peut contenir plus d'étudiants que sa capacité.

Il existe aussi d'autres contraintes supplémentaires. Ainsi, il peut être exigé que certains examens aient lieu en même temps, que des examens doivent avoir lieu avant ou après d'autres examens, etc.

L'objectif premier de la confection d'horaires d'examen est de trouver un horaire ne comportant aucun conflit. De nombreux objectifs secondaires peuvent être fixés: répartir uniformément les examens de chaque étudiant, planifier les examens le plus tôt possible, utiliser l'ensemble des locaux disponibles le plus efficacement possible, etc.

Pour l'École Polytechnique de Montréal, les contraintes suivantes doivent être respectées:

- Aucun étudiant ne doit avoir plus d'un examen au même moment.
- Le nombre d'étudiants durant une période est limité par la capacité totale disponible.
- Plusieurs examens peuvent être alloués dans le même local.
- Un même examen peut se dérouler dans plusieurs locaux.
- Au moins deux examens devraient être prévus dans chaque local.

Il existe ensuite un ensemble de contraintes que l'on doit tenter de satisfaire mais qui peuvent tout de même être violées:

- Éviter de donner deux examens à un même étudiant le même jour.
- Éviter de donner deux examens à un même étudiant lors de journées consécutives.
- Favoriser les examens durant les périodes de l'avant-midi.
- Favoriser la tenue des examens le plus tôt possible.

- Favoriser une répartition équilibrée des examens de chaque étudiant.
- Favoriser une utilisation efficace des locaux durant chaque période.
- Essayer d'équilibrer le nombre d'étudiants dans chaque local.

III. IMPLÉMENTATION

1. Modélisation des données

Des variables ***exams*** appartenant au domaine des entiers sont utilisées pour représenter les examens. Une variable ***cost*** représentant la fonction d'optimisation est définie ainsi:

$$cost=0.5*\sum_i \sum_j examConflict[i][j] * costWeight[distance\ between\ i\ and\ j]$$

où chaque élément de ***examConflict*** représente le nombre d'étudiants communs entre les examens et où chaque élément de ***costWeight*** représente le coût entre deux examens s'ils ont un étudiant en commun. Des variables ensemblistes entières ***roomsForExam*** sont définies pour permettre l'allocation des locaux aux examens. De plus, plusieurs entiers réversibles sont utilisés pour permettre une programmation plus aisée et efficace.

2. Représentation des contraintes

La première contrainte C1 est représentée par la fonction ***IlcAllDiff***(exams). La contrainte C2 est représentée par ***numStudForExam[i] <= *sessionSize[j]***. tandis que les contraintes C3 et C4 doivent satisfaire la condition:

$$numStudLeft[exam] <= \sum_{room} roomCapacity[period][room]$$

En ce qui concerne la contrainte C5, le local est divisé en deux parties que l'on considère comme des locaux distincts ne pouvant faire partie du même ensemble. Les contraintes

molles C6 à C10 doivent être satisfaites durant la stratégie de recherche selon leur priorité. Les contraintes C11 et C12 peuvent être satisfaites par l'algorithme d'allocation des locaux.

3. Stratégies de recherche

Le problème de confection d'horaires d'examens comprend deux sous-problèmes: allouer une période à chaque examen sans conflit d'horaire et allouer un ou plusieurs locaux à chaque examen. Dans ce projet, nous résolvons ces deux sous-problèmes en deux phases.

3.1 Stratégies de recherche pour la première phase: problème de confection d'horaires des sessions d'examen

Cinq méthodes sont examinées pour choisir la prochaine variable: ***la méthode statique***, où la liste des examens est triée initialement; ***la méthode des plus petits domaines***, qui consiste à sélectionner l'examen avec le plus petit domaine; ***la méthode aléatoire***, où le prochain examen est choisi au hasard; ***la méthode du moindre regret***, où l'examen sélectionné est celui pour lequel le deuxième meilleur choix entraîne un coût de beaucoup supérieur à celui du meilleur choix; ***la méthode du plus grand nombre de conflits***, qui sélectionne l'examen possédant le plus grand nombre de conflits avec les examens déjà fixés.

Trois méthodes sont examinées pour la sélection des valeurs: choisir la période entraînant le plus petit coût avec les examens déjà fixés; choisir la période pour laquelle

le nombre d'examens non-fixés qui pourraient rentrer en conflit avec l'examen considéré est le plus petit; choisir une période aléatoirement.

3.2 Stratégies de recherche pour la deuxième phase: problème de confection d'horaires des locaux d'examen

Deux règles statiques sont utilisées pour sélectionner le prochain examen: choisir le prochain examen en fonction de sa taille ou en fonction de la taille de la section.

Trois règles sont examinées pour la sélection des valeurs: ***la méthode de la plus grande valeur en premier***, selon laquelle le plus grand local ou les plus grands locaux doivent être alloués immédiatement à l'examen considéré; ***la méthode du premier ajustement***, qui ne fixe pas l'examen considéré immédiatement du premier coup mais résout plutôt le problème partiellement à chaque fois que l'examen est sélectionné; ***la méthode du meilleur ajustement***, où un ensemble des locaux les mieux ajustés sont alloués à l'examen considéré.

4. Réduction de l'espace de recherche

Trois fonctions de coût sont évaluées pour l'ordonnancement des périodes: les deux premières utilisent les même principes que les deux premières méthodes de sélection des valeurs de la section 3.1, tandis que la troisième, plus complexe, alloue chaque période disponible à l'examen pour trouver une solution réalisable en accord avec la stratégie de recherche. Elle trie ensuite les périodes en ordre croissant du coût de ces solutions. Une

fois ordonnées, on ne considérera qu'une certaine tranche initiale de ces périodes afin de réduire l'espace de recherche.

5. Algorithme de contrôle de la recherche

Trois algorithmes sont utilisés pour diriger la recherche: la recherche en profondeur d'abord explore en premier les derniers noeuds générés; la recherche à discordance limitée divise l'arbre de recherche en bandes selon le nombre de discordances et les explore une à une; la recherche à discordance limitée selon la profondeur progressive, une variante de la précédente, divise l'arbre de recherche en strates.

IV. RÉSULTATS

1. Résultats du problème de confection d'horaires des sessions d'examen

1.1 Comparaisons des différentes méthodes de sélection des variables

Cinq implantations ont été construites. Toutes utilisent la même règle de sélection des valeurs: choisir la période qui engendre le plus petit coût avec les examens fixés. Les méthodes de sélection des variables décrites à la section 3.1 sont utilisées pour chaque implantation. La première adopte la règle du plus grand nombre de conflits, la deuxième celle du plus petit domaine, la troisième celle du moindre regret, la quatrième la règle statique et la cinquième ordonne la liste aléatoirement.

La première implantation obtient la première meilleure solution après 56 secondes, sans échec. Dans cette solution, il y a respectivement 86 et 1783 étudiants qui se retrouvent

avec deux examens la même journée ou durant des journées consécutives. La deuxième implantation prend environ 44 secondes; ces résultats sont inférieurs à la première mais demeurent au même niveau que sont actuellement utilisés par Polytechnique. La troisième implantation prend environ 480 secondes, et permet d'obtenir des résultats équivalents à ceux de la première. La quatrième nécessite environ 50 secondes et un échec, avec des résultats équivalents à ceux de la deuxième. Finalement, les résultats et la durée de traitement de la cinquième implantation sont médiocres.

1.2 Comparaisons des différentes méthodes de sélection des valeurs

Trois implantations ont été construites: toutes utilisent la même règle de sélection des variables, celle du plus grand nombre de conflits. Les méthodes de sélection des valeurs décrites à la section 3.1 sont utilisées. La première trouve une première solution après 0.3 secondes; ces résultats sont d'une qualité comparable à ceux de l'École Polytechnique. La deuxième nécessite 0.35 secondes; ces résultats sont inférieurs. La troisième prend 0.3 secondes, mais ces résultats sont encore moins intéressants.

1.3 Réduction de l'espace de recherche et algorithmes de contrôle de la recherche

Trois implantations ont été construites selon les fonctions décrites à la section 4; les résultats de la troisième sont les meilleurs. Il apparaît que chacune des trois stratégies de contrôle de la recherche permet de trouver la meilleure solution, mais la stratégie de recherche à discordance limitée prend le moins de temps.

2. Résultats du problème de confection d'horaires des locaux d'examen

Deux alternatives se présentent pour allouer les locaux aux examens: considérer chaque examen de façon entière ou plutôt considérer la section comme unité de base.

2.1 Comparaisons des différentes méthodes de sélection des valeurs

La première alternative est adoptée. La sélection des variables consiste à toujours choisir l'examen le plus important en premier. Trois implantations sont construites; elles utilisent respectivement la méthode de la plus grande valeur en premier, la méthode du premier ajustement et la méthode du meilleur ajustement. Les résultats montrent qu'en ce qui concerne l'utilisation des locaux la règle du meilleur ajustement est la meilleure, suivie du premier ajustement et de la plus grande valeur en premier. Le niveau moyen d'utilisation des salles est respectivement de 96.6%, 90.4% et 76.6%. En ce qui concerne l'utilisation du plus petit nombre de salles, la règle du premier ajustement est la meilleure.

2.2 Ordonnancement des locaux selon la section

L'étape élémentaire consiste à choisir un examen, essayer d'allouer une section par local puis, si ce n'est pas possible, essayer d'allouer une moitié de la section dans un local et l'autre moitié dans un autre local, et ainsi de suite. Deux implantations ont été construites. Elles utilisent la même règle de sélection des valeurs, la règle du meilleur ajustement, mais diffèrent par la règle de sélection des variables, l'une utilisant la règle du plus grand examen et l'autre de la plus grande section. Les résultats montrent qu'en terme de l'utilisation des locaux, la première règle est légèrement supérieure. Mais si nous

considérons que de nombreuses sections sont séparées dans plusieurs locaux différents, la deuxième règle devient préférable.

V. CONCLUSION

1. Contributions

Nous avons divisé le problème en deux sous-problèmes, ce qui a permis une résolution en deux phases. Pour chaque sous-problème nous avons développé plusieurs stratégies de recherche. Différentes techniques furent utilisées pour réduire la taille de l'espace de recherche et différents algorithmes de contrôle pour améliorer la performance. Toutes les contraintes dures furent satisfaites dans chaque algorithme, tandis que les contraintes molles avec les plus hautes priorités furent satisfaites avant celle de plus basse priorité. Chaque implantation est en mesure d'effectuer automatiquement l'allocation des locaux. Les contraintes, comme par exemple les contraintes de préséance, peuvent facilement être ajoutées à l'application durant l'exécution en les lisant dans un fichier texte. Les résultats de notre étude prouvent que la programmation par contraintes est très puissante en ce qui concerne le problème de confection d'horaires d'examens.

2. Discussion et travaux futurs

La sélection des variables joue un rôle très important dans la recherche d'une meilleure solution. Dans ce projet plusieurs règles de sélection furent étudiés, mais toutes ne considéraient que l'impact des examens déjà fixés. Idéalement, il devrait être possible de considérer à la fois l'impact sur les examens fixés et non-fixés lors de la sélection du

prochain examen. Notre méthode de tri de la liste des périodes disponibles est coûteuse; le développement d'une méthode plus simple et efficace permettrait de réduire l'espace de recherche rapidement et donc de diminuer le temps d'exécution. Les stratégies d'allocation des locaux utilisées dans ce projet sont relativement simples; nous tentons simplement de trouver une première solution. Il est possible de définir une fonction de coût associée avec chaque solution pour ainsi permettre de trouver une solution optimale. Finalement, nous pourrions ajouter certaines options de visualisation à l'application, comme par exemple une interface graphique permettant à l'usager d'ajouter ou de retirer certaines contraintes, de sélectionner le nombre maximum de discordances et la limite de temps, etc.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The range of practical timetabling problems is wide. Teachers and lecturers have to arrange classes or examinations; hospital managers have to devise staff rotas; transport planners have to devise bus and train schedules; distribution managers have to fix delivery routes and schedules; and so on. The problems are very difficult to be solved and are classified as NP-hard problems. Sometimes the problems are tackled by hand, it is really a painful task. Computers are the ideal tools for solving timetabling problems, but it is still difficult to find the right algorithm for the particular problem.

The difficulties of a typical timetabling problem arise from its large scale, the great number of contradictory requirements, constraints and criteria of assignments' quality. A lot of efforts have been spent on the subject, and hundreds of research papers have been published. But in spite of numerous attempts to solve them, the timetabling problems still present a challenge for Operations Research and Computer Science.

The examination scheduling problem is a special case of timetabling problems. It is becoming an important operations management problem in schools and universities. At the end of each term or year, most educational institutions must schedule a set of examinations. With the expansion of the size of the problem and with the increase of

more complicated constraints, the problem becomes even more difficult and much effort needs to be put into it.

The most usual methods to solve this kind of problem are inherited from Operations Research such as graph coloring and mathematical programming, from genetic algorithms or from local search procedures such as simulated annealing and tabu search. These well-known methods are used widely and have given good results. But still these methods have their inherent disadvantage, i.e., it is difficult to find a model that can include all the constraints.

Constraints have emerged as the basis of a representational and computational paradigm that draws from many disciplines and can be brought to bear on many problem domains. The timetabling problem can be elegantly formalized as a constraint satisfaction problem and implemented by means of constraint solving techniques.

Recently, Constraint Programming (CP) has become an interesting approach for solving timetabling problems. It takes the advantages of both of Constraint Solving and Logic Programming and makes CP programs both expressive and flexible, and in some cases, more efficient than other kinds of programs. It has already been proven that CP is successful in tackling many combinatorial optimization problems such as planning, assignment, resource allocation, scheduling, placement, and configuration. In fact, CP is

very useful for building applications where no general algorithm is available and where changes may frequently arise.

To solve its examination scheduling problem, École Polytechnique de Montréal uses a computer program developed about 15 years ago. With the expansion of the school's student enrollment and steady growth in the number of courses offered in each semester and with the increase of more complicated constraints, it may not be suitable for dealing with the problem anymore. And also its program does not allocate the rooms for the examinations.

In this project, the Constraint Programming method will be used to generate new examination timetable for École Polytechnique de Montréal, including room allocation.

1.2 Research Goals

The primary aim of this master thesis is to look for a better algorithm to solve the problem for École Polytechnique de Montréal. The algorithm should satisfy the following requirements:

- The solution must satisfy all hard constraints.
- The solution should satisfy soft constraints as much as possible.
- The algorithm should find solutions better than those from École Polytechnique.

- The algorithm should be efficient, that is, it should find better solutions within the time limit.
- The algorithm should assign rooms for each examination.
- Ad hoc constraints should be easily added to the application.

1.3 Thesis Organization

Following this chapter, chapter 2 provides an overview of timetabling problems, including two parts: definition and classification of timetabling problems in education; different kinds of methods to solve the problem. Chapter 3 describes the examination scheduling problem: the objective and the common constraints (both hard and soft constraints) of the problem. It also describes the requirements of École Polytechnique de Montréal. Chapter 4 gives a review of constraint programming. It introduces the ways to solve the constraint satisfaction problem; shows how to propagate the constraints; how to select value and variable during the search for the solution; how to solve the problem in the way of constraint programming. It also gives a brief introduction to the constraint programming language: ILOG SOLVER. In chapter 5, the model of the problem is established, and different search strategies are introduced. Combining different value and variable ordering methods together, several different implementations can be constructed. Chapter 6 shows the experimental results of the different algorithms. It also has some discussions and comparisons among the algorithms. Conclusions from this project and

future work are presented in Chapter 7. An example of the solution is shown in the Appendix.

CHAPTER 2

REVIEW OF THE TIMETABLING PROBLEM

Various scheduling problems in education including construction of lecture and exam timetables, and course and classroom scheduling are among the most difficult in educational planning. Usually the timetabling problems vary among universities depending on their specific requirements and conditions. Therefore different timetabling systems are applied to the different institutions.

2.1 TIMETABLING PROBLEMS

General timetabling problems in education can be defined as the scheduling of a set of lectures attended by a specific group of students and given by certain teachers over a preset period of time, using certain resources and satisfying a certain set of constraints.

In fact, timetabling in an educational setting cover a wide range of scheduling problems. They can be classified as follows[1]:

1. Faculty Timetabling: There exists a set of instructors and a set of courses. The problem is to match courses and the instructors under specified conditions.
2. Class-Teacher Timetabling: This problem involves a set of classes and a set of teachers. Each class has a set of students who will take the same courses together. Each single lesson required by the class must be assigned to the time period in such

a way that no teacher and no class should have two or more lessons at the same time.

3. Course timetabling: A set of students is given and for each student the set of lectures that he/she must attend is defined. The lectures must be scheduled to time periods in such a way that no student can have more than one lecture at the same time. In fact, the objective of course scheduling is to minimize the total number of conflicts (number of students who have more than one lecture at the same time). If a practical course timetabling problem requires a conflict-free schedule in uniform time periods, it is equivalent to an examination timetabling problem.
4. Examination Timetabling: A set of students and a set of examinations are given. Each examination must be assigned to a time period so that no student should attend two or more examinations simultaneously.
5. Classroom Scheduling: Each lecture or examination must take place in a classroom. On the one side, no classroom should be used by different lectures at the same time, on the other side, several examinations may be scheduled to the same classroom simultaneously, or one examination can be scheduled to several different classrooms.

These problems are not independent and usually they are considered together. For example, when scheduling examinations, one should consider classroom scheduling and faculty timetabling problems simultaneously, otherwise the examination can't really happen because rather there is no classroom or no invigilator for it.

Besides the above conditions for each timetabling problem, there also exists different general and specific requirements that must or should be satisfied. For example, the institution may preassign some lessons to specified time periods according to the requirements determined for teachers, classes, subjects, classrooms, etc. These requirements can be divided into necessary and desirable ones. They can vary from one institution to another. When the timetabling problem is formulated as a Mathematical Programming problem, the necessary conditions determine the set of feasible solutions, and desirable conditions determine the optimality criteria.

2.2 SOLVING TIMETABLING PROBLEMS

Since the fifties, many efforts have been devoted to timetabling problems[2.3], and because of its variety and its complexity[4.5], the timetabling problem continues to be studied nowadays. In this section, several approaches or methods used to solve this problem are presented. We will concentrate on those approaches relevant to examination timetabling problems.

2.2.1 OR approaches

Operations Research (OR) approach for solving hard combinatorial optimization problems was the one used for a long time. It is based on a mathematical representation of the problems that are usually modeled as integer linear programs.

2.2.1.1 Graph Coloring

It is well known that finding a conflict-free timetable without side constraints is a graph coloring problem[2]. A graph of a timetabling problem that should schedule a set of lectures over p periods can be constructed as follows:

i. Each course or examination is represented by a vertex (if a course has several lectures, then each lecture should be represented by a vertex. thus examination graphs will be generally smaller in size than course scheduling graphs).

ii. Two vertices are connected by an edge if the associated courses have at least one student or teacher in common, which means they can't be scheduled to the same period.

The question is: can the vertices of the graph be colored using a set of p colors (available periods) so that no two vertices connected to each other have the same color? A closely related problem is: what is the minimum number of colors (χ) necessary to color the vertices of the graph without conflict? It is known that the problem is NP-complete when $p \geq 3$.

However practical timetabling problems differ form graph coloring problems when side (or secondary) constraints are considered, for example: room capacity constraints, pre-assignments, exclusions (certain courses are excluded from some particular periods), etc. According to Grimmett and McDiarmid[6], if p is much greater than 2χ, then it is easy to find a conflict-free timetable, thus there is likely to be considerable flexibility in accommodating secondary constraints. But if p is close to χ, then it will be difficult to

find a conflict-free timetable, thus secondary constraints are likely violated in order to satisfy the primary objective (finding a conflict-free timetable).

Usually, most algorithms use a list processing scheme to solve the underlying graph coloring problem: first courses are sorted according to some criterion, and then they are sequentially assigned to periods without creating conflict[2,7,8,9].

2.2.1.1.1 Sequential Methods[10]

In sequential methods, courses that are chosen using some sequencing strategies are assigned to a specific period one at a time. The methods typically employ a two phase approach: a construction phase produces an initial timetable and an improvement phase makes modifications.

There exist several sequencing strategies that can be used for the construction phase[12]:

1. Largest degree (LD): largest number of conflicting courses first. i.e., courses that conflict with many other courses should be scheduled early. The rationale is that these courses are hard to schedule and thus they should be assigned first.
2. Saturation degree (SD): number of periods in conflict (smallest-domain first). The next selected course should be the one that has the fewest number of feasible available periods remaining. The rule is based on the assumption that courses with small domain are difficult to schedule. It is a dynamic selection rule and ties can be broken using some rules, i.e., largest degree.

3. Largest enrollment (LE): The next selected course should be the one that has the largest number of students. The reason is that courses with large enrollment usually create more conflicts with other courses and thus are hard to schedule.
4. Largest weighted degree (LWD): The next selected course should be the one that has the largest number of students in conflict with other courses. It can be thought of as a combination of rules 1 and 3.
5. Random ordering: select the next course in random order.
6. User defined priority groups: select high priority courses first.

Beside the different sequential strategies, there also exist different ways to choose a period for the selected course, i.e., choose the earliest feasible period for the course in order to minimize the total number of periods required, or select the best feasible period for the course according to a measure of the objectives.

In the examination timetabling system EXAMINE [11,12], the authors use sequencing strategies 1-5 as user selected options. They point out that some strategies work better on some problems. In general, the saturation degree sorting rule is the most robust.

Usually the improvement phase involves using some rules to move courses to better periods or make some interchanges among courses so that the final solution can be improved[11].

2.2.1.1.2 Cluster Methods[10]

In these methods, the examinations are first divided into different sets (cluster phase), each of which contains only compatible examinations (conflict-free group), i.e., each set can feasibly be assigned to period without conflicts. Then these sets (clusters) are scheduled into specific periods to minimize some objectives or satisfy some constraints.

There exist a variety of approaches for the cluster phase[7,13,14,15]. For example[13], select the next examination from the examination list, which is sorted either in ascending or descending order of number of conflicts with all other examinations and then the selected examination is added to a set with which it has no conflicts and has the largest number of conflicting examinations in common. In [14,16], there are a fixed number of blocks for all examinations. First the examination list is sorted by the degree times the number of student conflicts with all other examinations, then examinations are selected one by one from the list and scheduled to the first available conflict-free block, i.e., with no instructor conflict and sufficient room capacity. If there is no conflict-free block, the examination is assigned to the block with the minimum number of conflicts. Finally, employ simple interchange rules to reduce the number of conflicts. D. Johnson [7] sorts the examination list according to a linear combination of two criteria: the size of the examination and the number of conflicts with other examinations. By varying the relative weights, a wide variety of different conflict-free groups and thus different timetables can be produced. J. Fisher and D. Shier [15] use the original course timetable to construct conflict-free groups.

For the second (sequencing) phase, there also exist several methods. The simplest approach uses the objective of trying to minimize the number of students who have consecutive examinations. This problem can be modeled as a Traveling Salesman Problem(TSP) where each cluster represents a city, the distance between each pair of clusters is equal to the number of students who must attend an examination in both clusters. Thus finding a minimum distance tour that visits each cluster only once is equivalent to finding a sequence for the groups with the minimized objective. There are several ways to solve the TSP, for example, a simple pairwise interchange heuristic[15], dividing the groups into compatible pairs and solving it as a minimum weighted matching problem[16], using network model with a Lagrangian penalty[17]. Another approach is that the group sequencing problem can be modeled as a Quadratic Assignment Problem (QAP) where the objective is to minimize the number of students who have two examinations in any x consecutive periods[14].

The advantage of the cluster approach is that more sophisticated optimization techniques can be used on the period sequencing problem because examinations are first divided into low conflict group, and thus the size of the sequencing problem is reduced. However, since the search space is drastically reduced and simple heuristics are usually used to divide the examinations into compatible groups, the potential quality of final solution must be affected.

2.2.1.2 Network flow problems

As mentioned above, graph coloring problems are different from practical timetabling problem when side constraints (requirements) are considered and it is not enough to take into account more complicated requirements. To make the lessons' distribution more uniform, network flow optimization[3,18] is used. In this case, the problem is represented as a directed graph, and each arc connecting two vertices receives a lower and an upper bound that will change during the resolution of the problem. By solving max-flow problems, we can obtain some solutions that can define natural decomposition of the problem to periods.

This method is interesting because there are many well-known efficient algorithms for solving the timetabling problem. However, when solving a timetabling problem over several periods we must find a flow for each period. It is still an NP-complete problem. Also this method can't include all types of constraints, i.e., precedence requirements.

2.2.1.3 Mathematical programming

In order to take into account the most difficult requirements, timetabling is often reduced to some general integer mathematical programming problem[3,19]. In general it is an NP-complete problem. The main problem in this case is the size of the problem, i.e., the number of data and constraints. In order to reduce the problem size, it is possible to redefine the variables by grouping students, rooms or lectures and then use some relaxation methods[20].

2.2.2 Genetic Algorithms

Genetic algorithms (GA) were developed by John Holland and his students as artificial adaptive systems that simulate natural evolution[21]. One feature of GA is that it can search very large spaces effectively and efficiently, thus they are used to deal with inherently intractable problems, i.e., NP-complete problems.

GA constitutes a class of iterative optimization algorithms. The idea is to propagate a population of possible solutions such that an optimal or at least very good solution can grow in the process. GA explores the search space from many different points simultaneously. It starts with a set of initial solutions called population, and produces new generations iteratively in the following way:

- Select randomly two parent solutions from the present population.
- Apply evolutionary production operators to generate new solutions in the search space. In this way a set of offspring is produced.
- Eliminate those individuums with low fitness from the new population by a reduction operator. Thus only good solutions are kept in the new generation.

The initial populations can be produced in many ways, i.e., generating a solution randomly or using special initial algorithms. There are many different evolutionary production operators for producing children, i.e., using a vector to represent an examination timetable, the j-th entry of the vector indicates which period the

examination is assigned to. Two new children solutions can be produced by combining two different parents solutions, which are split into two parts randomly[22], or using a mutation operator where individual genes are randomly selected to be randomly moved[23], etc. A lot of fitness functions have been used to eliminate bad solutions, and it can include different criteria: the length of the timetable; the number of conflicts; the spare capacity in each of the rooms; the number of examinations in a day for a student; two consecutive examinations; spreading examinations out for students[22,23,24], etc.

GA can have good results and find efficient solutions. However, all parameters must be determined through experimentation. Furthermore it can't guarantee the convergence of solutions.

2.2.3 Simulated Annealing

Simulated annealing is analogous to the annealing of materials to produce sound low energy states. It was developed by Kirkpatrick and colleagues[25] and was used as an approach to optimization. The basic concept is that one tries to find a feasible solution fairly quickly, and then randomly and iteratively selecting a neighborhood solution. If the new solution is a better one, then it is accepted; a worse solution may be accepted with some probability. Initially the probability is high, a worse solution is more likely to be accepted, leading possibly to escape from local optima. As the process continues, the probability is slowly lowered, thus only better solutions are accepted. After finding local

minimum solutions, the probability can be increased again, and the procedure is repeated until a new local minimum is discovered. When some stopping criteria is met, the best solution is reported.

The success of simulated annealing depends on the starting probability, the initial solution, the refining heuristic and the cooling schedule (the rate of probability decrease). There are several timetabling works using simulated annealing that obtain some good solutions: Thompson & Dowsland[26,27] use the method to solve the examination timetabling problem for Swansea University, Abramson[28] applies it to solve the class-teacher timetabling problem, Davis & Ritter[29] use it to solve the student scheduling problem; etc.

2.2.4 Tabu Search

Tabu search [30,31] is an effective local search method that moves step by step from one initial solution towards an optimal or near-optimal solution. It is similar to simulated annealing in the sense that neighborhood moves are used to move out of a local optimum. The basic step is to move from the current solution to the best solution, even if the solution is worse. In order to avoid cycling, a tabu list is maintained, which contain the solutions that have been visited before, and the algorithm is forced to look elsewhere. In this manner, the algorithm looks for the best improving move without going back to a tabu move. Usually the search stops after a maximum number of iterations and the best solution visited is chosen.

Tabu search has been used to solve timetabling problems in educational institutions. It has been applied to both course and examination timetabling problems by Hertz[32]. It has been used to the solve examination timetabling problem at the University of Technology of Compiègne in France[33], which involves up to 130 examinations in 20 periods. It has also been applied to the class-teacher timetabling problem[34,35].

2.2.5 Constraint Based Approaches

In recent years a new paradigm, Constraint Programming (CP), has been successfully used for solving hard combinatorial optimization problems such as scheduling, planning, sequencing and assignment problems[36,37]. Most of the constraint programming literature models a Constraint Satisfaction Problem (CSP) as a set of variables with a finite domain and a set of constraints, each variable must be assigned to a value so that a number of constraints can be satisfied. Most of the timetabling requirements can be translated into constraints. Typically these systems are solvers that implement powerful filtering algorithm and use backtracking to find a feasible solution.

Nuijten, Kunnen, Aarts and Dignum[38] use a general constraint satisfaction technique to solve the examination timetabling problem (CSP) at the Eindoven University of Technology. The problem is fairly small with 275 examinations, 7000 students and 33 examination periods over three weeks. They use a random selection algorithm to choose an examination and assign it to the first feasible period, and so on. If no feasible solution

is found, they try backtracking. David[39] develops an incomplete algorithm based on the CSP model to generate examination timetables at the École des Mines de Nantes. The timetable of a day must be generated in the morning, and at most there are 105 examinations per day. The algorithm uses local repair techniques rather than an exhaustive search method to find feasible solutions because there is a strong constraint: the computing time must be less than 1 minute.

CP has been used to solve educational timetabling problems recently [41,42,43,44,45,46,47,48]. For example, Boizumault, Delon & Péridy[41] generate examination timetables for l'Université Catholique de l'Ouest in Angers, France. They test the model on the 308 examinations that produce 2600 constraints and the real problem is solved within one minute of CPU time. Leong[42] solves the examination timetabling problem at the National University of Singapore, the size of the problem fairly large with over 1000 examinations involved. Fahrion & Dollansky[43] solve the teacher assignment problem. Cheng, Kang, Leung & White[44] solve their timetabling problem within 25 minutes using a 33 MHz – 80486 machine. The size of the problem is large: there are 2147 course entries with 155 pre-assignments and 167 classrooms whose size ranged from 4 to 474. Comparing with the manual system they used now, the rate of seating usage is increased from about 65% to about 80% and the large amounts of work required by the manual timetabling process can be greatly reduced.

CHAPTER 3

DESCRIPTION OF PROBLEM

3.1 THE EXAMINATION SCHEDULING PROBLEM

Examination scheduling is a difficult combinatorial problem. In its simplest form, the problem is to assign a set of examinations to a fixed number of potential time periods so that no student can take more than one examination at the same time. However, additional and specific constraints must be taken into account: room capacities, consecutive and nonconsecutive examinations, pre-assignments etc. Some of the constraints are hard while others are soft, i.e., some constraints must always hold in any situation while others may be relaxed, if necessary, in order to cast the schedule. In particular, automatic building of examination timetables is difficult because of the diversity of the constraints that must be taken into account.

Usually examination scheduling problems differ from one university to another, in other words, the constraints involved in examination scheduling are different. However it is generally accepted that the following two constraints are fundamental to any examination scheduling problem[2][49]:

1. No student can take part in more than one examination at any one time.

2. A room can only hold as many students (examinations) as its capacity (the total number of available desk-seat pair).

In practice, as described by Carter and Laporte[10], there also exist several common side constraints as follows:

1. Some examinations may be required to take place at the same time. (e.g., some similar examinations for different programs may have some same questions).
2. Some examinations may be required to take place before or after some other examinations(precedence constraints).
3. Some examinations may be required to be in consecutive periods. (e.g., oral examination followed by written examination).
4. Some examinations may be required to be on the same day.
5. Only a subset of periods is suitable for an examination. (e.g., an examination can only be scheduled to an evening period because part-time students are not available during a day period).
6. An examination may be preassigned to a room.
7. There is a limit on the total number of examinations that can take place at the same time (e.g., limited by number of invigilators or rooms).
8. An examination may be required to be in a particular kind of room (e.g., it requires special resources that are only available in that kind of room).
9. At least two examinations should be scheduled in the same room (e.g., École Polytechnique de Montréal has this requirement).

10. Examinations may be split over several rooms. If only one examiner is available for the examination, the rooms should be closed to one another.
11. For special students, some special requirements can be specified. (e.g., examination spacing, length, location etc.).

The primary objective of examination scheduling is to find a conflict free timetable (satisfying all hard constraints), which is to find a feasible timetable. The problem can be solved using a variety of methods depending on its characteristics (e.g., the size of the problem, different constraints, etc.). However, educational institutions usually are not only interested in finding a feasible timetable, but also they are more interested in finding a good or best feasible timetable (that satisfies soft constraints as much as possible). That is where the secondary objective comes from. In the examination scheduling problem, there exist several common secondary objectives[10] as following:

1. Minimize the number of occurrences of x examinations in any y consecutive periods for all students.
2. For each student, examinations should be spread out as evenly as possible.
3. Examinations or some special examinations (e.g., examinations with a large number of students) should be scheduled as early as possible (this objective conflicts directly with objectives 1 and 2).
4. Some examinations may have preference requirements on where and when they should take place. They should be satisfied as much as possible.

5. All of the available rooms should be used in an efficient way. For example, no room should be full while other rooms just have few students.

It should be noticed that these constraints described above are not required in all examination scheduling problems and are specific to particular educational institutions. Therefore, dedicated algorithms must be developed to solve different problems.

3.2 THE EXAMINATION PROBLEM OF ÉCOLE POLYTECHNIQUE DE MONTREAL

To solve its examination scheduling problem, École Polytechnique de Montréal uses a computer program developed about 15 years ago. With the expansion of the school's student enrollment and steady growth in the number of courses offered in each semester, and with the increase of more complicated constraints, it may not be suitable for dealing with the problem any more. And also its program does not allocate the rooms for the examinations.

As mentioned in Chapter 2, constraint programming has become an interesting approach for solving timetabling problems recently and has already been proved to be successful in tacking many combinatorial problems. In this project, we have concentrated on generating an examination timetable for École Polytechnique de Montréal by using constraint programming.

Before describing the problem, several terms should be defined:

1. Session: It is a time slot during which the examinations can take place. For example, at end of autumn term in École Polytechnique de Montréal, there are 14 available days for the final examination, and each day has two sessions, that is morning session from 9:30 to 12:00 and afternoon session from 13:30 to 16:00. All examinations should be scheduled into these sessions.
2. Room capacity: It is the total number of available desk-seat pairs of a room. Usually the rooms have different capacities, for example, among all available rooms in École Polytechnique de Montréal, the largest room has 129 desk-seat pairs while the smallest room only has 28 desk-seat pairs. When examinations are assigned to the rooms, room capacity should be considered, e.g., the examination with a large number of students should be scheduled into big rooms etc.
3. Session capacity: It is the sum of room capacity of all available rooms for the session. More formally, for each session its capacity can be expressed as:

$$\text{sessionCapacity}[j] = \sum_i \text{roomCapacity}[i]$$

where j is the index of session, e.g., the index of the first session is 1, and the index of the nth session is n, etc. i is the index of available rooms in session j. Usually different sessions have different sets of available rooms. When assigning examinations to each session, session capacity should be considered, e.g., the session capacity should exceed the total number of students attending these examinations, otherwise there must be some students who do not have seats and desks for writing their examinations.

When we talk about conflict in the field of examination scheduling, usually we mean two kinds of conflict. The first one is referenced as time conflict. If two examinations having at least one student in common are scheduled to either the same session or to overlapping sessions, we say there is a time conflict between them. The second one is referenced as room conflict (or room capacity conflict). When we try to assign an examination to a small room (relative to the examination), there always exists this kind of conflict. We can't put a large examination in a small room as there is not enough space for all the students. In this project, when we mention conflict, we mean time or room conflict.

École Polytechnique de Montréal is not a large university. Every autumn and winter term, roughly 200 courses are offered which have final examinations at the end of the term, and about 4000 students in various programs who have selected one or several courses must attend final examinations.

For the autumn 1999 term, the problem consisted of planning 192 different examinations in 28 sessions (half-days) during two weeks. The objective was to find a conflict free examination timetable while respecting the soft constraints as much as possible. Various kinds of requirements have been proposed, we classified these requirements (constraints) into several categories according to their properties, and then divided them into two groups: hard constraints and soft constraints as following.

A set of various constraints of different types must be satisfied (hard constraint):

- C1: Examination incompatibilities: no student can attend more than one examination at the same time. It is based on the assumption that no student can finish two or more examinations during one session, otherwise either the examinations are too simple or the session is too long.
- C2: Session capacity: the total number of students who can attend examinations at the same time are limited by the session capacity, in other words, it should be smaller than the capacity of the session. In our case, there are 14 days and 28 totally available sessions, every session has the same set of available rooms, therefore they have exactly the same capacity: there are 29 different rooms whose capacities are between 28 and 129. And in total there are 1611 desk-seat pairs for each session, which means at most 1611 students can write their examinations simultaneously.
- C3: Room availability: several examinations can be assigned to the same room if they satisfy the room capacity constraint. The purpose of this constraint (requirement) is to use the rooms efficiently. For example, among the 192 final examinations there are 16 examinations whose number of students is less than 10. On the contrary, as mentioned above, even the smallest room can hold a maximum of 28 students at the same time, which means that several kinds of examination can be scheduled into it simultaneously. Rooms are a limited resource and we want to use them efficiently.

- C4: Examination can be assigned to several different rooms if the total number of students of that examination exceeds the capacity of the largest room or even if it does not exceed. But if only one examiner is available for the examination, the rooms should be close to one another. Ideally an examination should take place in one room, but in reality, this is impossible: rooms have limited capacities while the examinations may have a large number of students. For example, in École Polytechnique de Montréal, the biggest room for final examination can hold only 129 students at the same time while the largest examination has 873 attendants. Obviously we can't put such a large examination in any room.
- C5: At least two examinations should be scheduled into the same room, for some of the rooms. In our case, this constraint only applied to a subset of rooms: 2 rooms exactly. The seats for the students of different examinations should be arranged alternatively, e.g., all seats are divided into several columns, the students of one examination take those columns with odd numbers while the students of other examination take those columns with even numbers, or vice versa. With the help of this constraint, the room capacities and so the session capacity can be enlarged, e.g., suppose there is a room with several seats (say 50 seats) that are close to each other. If they are required to hold only one examination, then the room capacity will be 25 because the students of the same examination should not have seats close to each other. On the contrary, if it can take two examinations at the same time, then its capacity will be 50.

A set of soft constraints may be violated :

- C6: Avoid giving a student two examinations on the same day. Usually, after writing an examination, students need some time to rest. If a student has two examinations on the same day, that is two examinations from 9:30 to 16:00 in our case and there is only one and half hour between them, he can't even find time to rest. It is not fair to him because he may not keep his head clear during the afternoon session. Because of this reason, we take this constraint more seriously than any other soft constraints.
- C7: Avoid giving a student two examinations on consecutive days, for the same reason as described above, but this soft constraint is not as serious as C6 since students get at least one night for rest and preparation.
- C8: Examinations should be arranged in the morning sessions. Usually, we think that in the morning people have more energy and have an active brain, therefore the morning sessions are more suitable for examinations. Coincidentally, this constraint is consistent with C6, and it is one convenient way to satisfy C6 (the other possible way is to arrange the examinations into afternoon session), e.g., in the extreme situation, if all examinations are scheduled into the morning sessions without any conflict, then C6 will be completely satisfied.
- C9: Examinations with a large number of students should be scheduled as early as possible. This is a convention, and we just do not want to put such examinations at the end of examination period. But because of the limit of the session capacity and the room capacity and also because of the soft constraints C6 and C7, some of

such examinations may be scheduled to the late sessions, e.g., the sessions that are close to the end of examination period.

- C10: Examinations for each student should be spread over the examination period. In this way, students can get enough time to rest and to prepare for other examinations.
- C11: Try to schedule all examinations to early sessions. It is also a convention. Usually students want to finish their examination early. Comparing with C9, it is even softer. It should be noticed that this soft constraint conflicts directly with other soft constraints, e.g., C6, C7, C8 and C10.
- C12: Examinations in each session should use the available rooms in an efficient way, i.e., try to improve the rate of room utility, namely, no rooms will be full of students while other rooms just have few students.
- C13: The number of students of an examination distributed in several rooms should be balanced. For example, there is an examination with 100 students, and two set of rooms: one has two rooms, the size of each room is 50; the other also has two rooms, but with the size 10 and 90 respectively. In the situation, we should choose the first set as our solution.

As we can see soft constraints are not independent, on the contrary they are related to each other, or even more some of them conflict with each other. There always exists a trade-off among those soft constraints. If some of them are well satisfied, the degree of

satisfaction for other soft constraints will decrease. We will see how to handle the constraints in the section 3 of chapter 5.

CHAPTER 4

OVERVIEW OF CONSTRAINT PROGRAMMING

Recently Constraint Programming(CP) has become an interesting approach for solving timetabling problems[38-48]. The problem can be elegantly formalized as a constraint satisfaction problem and implemented by means of constraint solving techniques.

CP is based upon the integration of two declarative paradigms: Constraint Solving and Logic Programming(LP)[40]. The scheme extends conventional LP by replacing the notion of unifiability with that of constraint solvability over an underlying constraint domain. This combination helps make CP programs both expressive and flexible, and in some cases, more efficient than other kinds of programs. It has already been proven that CP is successful in tackling many combinatorial optimization problems such as planning, assignment, resource allocation, scheduling, placement, and configuration. In fact, CP is very useful for building applications where no general algorithm is available and where changes may frequently arise.

According to the characteristic of the examination scheduling problem and based on aforementioned research, in this project, we choose CP method to generate examination timetable for École Polytechnique de Montréal.

This chapter is organized as follows: the first section gives the definition of constraint satisfaction problem. The second section introduces constraint programming, including the schemes to solve CSPs, constraint propagation, search strategies and the CP way to solve the problem. The last section gives a brief introduction to ILOG SOLVER.

4.1 NOTATION

A constraint is a way of specifying that a certain relationship must hold between the values taken by certain variables. It is not a test because a constraint can be exploited before a value has been assigned to a variable.

A primitive constraint consists of a constraint relation symbol from constraint domain together with the appropriate number of arguments, i.e., $X > 3$, $X + Y - Z = 5$, etc.

A Constraint Satisfaction Problem(CSP) can be defined as: a problem composed of a finite set of variables and a set of constraints that connect these variables together. Each variable is associated with its finite domain, and the constraints restrict the values that the variables can take simultaneously[50].

Formally, a CSP can be defined in the following way. Assume the existence of a finite set I of variables $\{X_1, X_2, ..., X_n\}$, which take their values from their finite domains D_1,

D_2,..., D_n, respectively, and a set of constraints. A constraint $C(X_{i1}, X_{i2},..., X_{ik})$ between K variables from I is a subset of the Cartesian product $D_{i1} \times D_{i2} \times ..... \times D_{ik}$, which specifies which values of the variables are compatible with each other. In practice, this subset does not need to be given explicitly, but can be defined by equations, inequalities or programs whatsoever. A solution to a CSP is an assignment of values to all variables, which satisfies all the constraints[36].

4.2 CONSTRAINT PROGRAMMING

4.2.1 The schemes to solve CSPs

There exist several schemes to solve CSPs[51]. The first simplest way is to enumerate the possible solutions using Generate and Test(GT) algorithm. GT method originates from the mathematical approach to solving combinatorial problems. First, the GT algorithm generates a valuation, that is, it assigns values to variables, and then it tests whether this valuation is a solution that should satisfy all the constraints. In this paradigm, each possible valuation is systematically generated and tested and the number of valuations considered by this method is the size of the Cartesian product of all the variable domains. As we can see, the GT approach is not very efficient because it generates many wrong valuations that are rejected in the testing phase. The fact is that the generator generates the valuation without considering the conflict.

The second way is to use a simple backtracking(BT) method. The most common

algorithm for performing systematic search is backtracking. Backtracking incrementally attempts to extend a partial solution toward a complete solution by repeatedly choosing a value, consistent with the values in the current partial solution, for another variable.

Backtracking can be seen as an extension of the GT approach. In the BT method, the generate and test phase is processed alternatively. First variables are instantiated one by one (generate phase) and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked (test phase). If any of the constraints is violated, backtracking is performed to the most recently instantiated variable whose domain still has some available values. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Therefore, backtracking is strictly better than GT. The following is its pseudo-code:

```
Back_Solve (C: Constraints, D: Domain):
  IF Vars(C) is empty THEN  return Partial_Satisfiable(C)
  ELSE
    choose x in Vars(C)
    FOR each value d in D(x)
      let C1 be C with x replaced by d
      IF Partial_Satisfiable(C1) THEN
        IF Back_Solve(C1, D) = TRUE   THEN return TRUE
    return FALSE

Partial_Satisfiable(C: Constraints):
  FOR each primitive constraint c in C
    IF Vars(c) is empty THEN
      IF satisfiable(c) = FALSE   THEN   return FALSE
```

return TRUE

The BT approach has three major drawbacks: one is thrashing, i.e., repeated failure due to the same reason. It occurs because the standard BT algorithm does not identify the real reason of the conflict. Another drawback is that it has to perform redundant work. Finally, the basic BT algorithm still detects the conflict too late, only after some variables are instantiated.

A third way is using *consistency techniques* that can effectively rule out many inconsistent labelings at a very early stage, and thus cut short the search for consistent labelings. The basic idea is to find an equivalent CSP to the original one with smaller domains of variables and the key point is to examine one primitive constraint at a time. There are several well-known widely used consistency techniques. One simplest consistency technique is referenced as *Node Consistency* that is used for unary constraints. The node represents a variable V in the constraint graph. A primitive constraint c is node consistent with domain D if either $|\text{Vars}(c)| \neq 1$ or, if $\text{Vars}(c) = \{x\}$, then for each $d \in D(x)$, $\{x \rightarrow d\}$ is a solution of c[52]. We say that a CSP is node consistent if each primitive constraint in it is node consistent. If the domain of V contains a value that does not satisfy the unary constraint on V, then the instantiation of V to this value will always result in immediate failure. The node inconsistency can be eliminated by simply removing those values from the domain of each variable V that do not satisfy an unary constraint on V.

The second consistency technique is *arc consistency* that is used for binary constraints. Each arc represents a constraint between variables represented by the end points of the arc. A primitive constraint c is arc consistent with domain D if either $|Vars(c)| \neq 2$ or, if $Vars(c) = \{x,y\}$, then for each $d_x \in D(x)$, there is some $d_y \in D(y)$ such that $\{x \rightarrow d_x, y \rightarrow d_y\}$ is a solution of c and for each $d_y \in D(y)$, there is some $d_x \in D(x)$ such that $\{x \rightarrow d_x, y \rightarrow d_y\}$ is a solution of c[52]. We say that a CSP is arc consistent if each primitive constraint in it is arc consistent. Clearly, an arc (Vi,Vj) can be made consistent by simply deleting those values from the domain of Vi for which there does not exist corresponding values in the domain of D such that the binary constraint between Vi and Vj is satisfied.

Arc and node consistency work well for pruning the domains in binary CSPs. However, they do not work well if the problem contains primitive constraints that involve more than two variables. How should these kind of primitive constraints be handled? Intuitively, Hyper-arc consistency technique, which is a true generalization of node and arc consistency, should be used. Unfortunately, determining if an arbitrary primitive constraint is Hyper-arc consistent is NP-hard and so is as expensive as determining if an arbitrary CSP is satisfiable. Instead of Hyper-arc consistency, *bounds consistency* can be used.

A CSP is arithmetic if each variable in the CSP ranges over a finite domain of integers and the primitive constraints are arithmetic constraints. An arithmetic primitive constraint c is bounds consistent with domain D if for each variable $x \in \text{Vars}(c)$, there is an assignment of real numbers, say $d_1, d_2, \ldots, d_k$, to the remaining variables in c, say $x_1, x_2, \ldots x_k$, such that $\min_D(x_j) \leq d_j \leq \max_D(x_j)$ for each d_j and $\{x \rightarrow \min_D(x), x_1 \rightarrow d_1, \ldots, x_k \rightarrow d_k\}$ is a solution of c, where $\min_D(x)/\max_D(x)$ are the minimum/maximum element in $D(x)$ respectively. And also there is another assignment of real numbers, say $d_1', d_2', \ldots d_k'$, to $x_1, x_2, \ldots x_k$, such that $\min_D(x_j) \leq d_j' \leq \max_D(x_j)$ for each dj' and $\{x \rightarrow \max_D(x), x_1 \rightarrow d_1', \ldots, x_k \rightarrow d_k'\}$ is a solution of c[52]. We say that an arithmetic CSP is bounds consistent if each primitive constraint in it is bounds consistent. Since bounds consistency only depends on the upper and lower bounds of the domains of the variables, only those domains that assign ranges to each variable should be considered.

It should be noticed that consistency techniques are rarely used alone to solve CSPs completely. They are usually used together with search solver. And so, the fourth way to solve CSP is to embed a consistency algorithm inside a backtracking algorithm. Notice that consistency techniques are deterministic, as opposed to the search that is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during the search is used only when there is no more propagation.

The simple backtracking (BT) algorithm described above can be used as a skeleton. Before starting the backtracking solver and after assigning a value to the variable, some consistency techniques are applied to the constraint graph. Depending on the degree of consistency technique we get various constraint satisfaction algorithms.

The simple BT can be seen as a combination of pure GT and a fraction of consistency. The BT algorithm tests consistency among already instantiated variables. Because the domains of instantiated variables contain just one value, it is possible to check only those constraints containing the last instantiated variable. If any domain is reduced then the corresponding constraint is inconsistent and the algorithm backtracks to a new instantiation. The BT algorithm detects the inconsistency as soon as it appears and, therefore, it is far more efficient than the simple GT approach. But it has still to perform too much search.

The second algorithm is forward checking which is the easiest way to prevent future conflicts. Instead of testing consistency among the instantiated variables, it checks consistency between the current variable and those variables that are not yet instantiated (the future variables). When a value is assigned to the current variable, any value in the domain of a future variable that conflicts with this assignment is removed from the domain. If the domain of a future variable becomes empty, then the current partial solution is inconsistent and the algorithm backtracks to a new instantiation. Clearly,

forward checking is more search efficient than the simple backtracking because it allows branches of the search tree that will lead to failure to be pruned earlier, at the expense of extra work at each tree node. Note that whenever a new variable is considered, all remaining values in its domain are guaranteed to be consistent with the instantiated variables, so the consistency checking among the instantiated variables is no longer necessary.

The third algorithm is a look ahead procedure that checks consistency not only between the current variable and the future variables but also among the future variables. It can be seen as an extension of forward checking. The advantage is that look ahead prunes the search tree further than forward checking, but again like forward checking, look ahead needs to do more work when each assignment is added to the current partial solution.

It should be noticed that more consistency checking at each node will result in a search tree containing fewer nodes, but the overall cost may be higher, as the processing at each node will be more expensive.

4.2.2 Constraint propagation

In constraint programming the most important aspect is constraint propagation. In fact constraint propagation is essentially an iterative algorithm that propagates the domain reduction information throughout the constraint network.

The algorithm used for constraint propagation can be implemented by maintaining a queue of variables (the constraint propagation queue). When a constrained variable is modified, that variable is put at the end of that queue, if it is not already in the queue. As long as there are variables in that queue, the algorithm takes the first variable from the queue and processes the variable.

When a variable is processed, it is first removed from the propagation queue. Then each constraint posted on that variable is examined. For one such constraint, all the variables on which the constraint is posted are in turn examined using consistency techniques described in section 4.2.1: their domains are reduced by removing those values that are inconsistent with the constraint. If some of these variables are modified during this activity, they too are put into the queue: the reduction of a variable's domain will trigger the examination of all constraints involving this variable, which in turn may reduce other domains. The algorithm continues as long as there is a variable in the queue to process. The algorithm automatically reduces domains as necessary and halts in either of two situations: when all domains contain values consistent with the constraints, or when a domain becomes empty, in which case we know that no solution exists.

An important property of constraint propagation algorithms is that the order in which constraints are propagated is unimportant. In other words, the final state of a constraint network is independent of the order of constraint propagation (the domains of all

variables are reduced in the same way). Another property is that the problem can be handled dynamically, that is, constraints can be added dynamically during the search for a solution.

It may not appear obvious that the propagation algorithm ever terminates. The fact is that propagation only shrinks domains: values are removed from the domains but never added into it. It follows that constraint propagation must terminate simply because there are finitely many values to be removed.

The effectiveness and cost of such an algorithm can be adapted by deciding when a particular constraint is worth triggering. There are three types of propagation event: the value event, the range event, and the domain event. A value event on a variable occurs when there is only a single value left in its domain and so it has been assigned that value. A range event on a variable occurs when one of the bounds of its domain, either the minimum or maximum, is changed. A domain event on a variable occurs anytime a value in its domain is removed.

Because of its incompleteness, simple propagation of constraints to reduce the domains of variables will not usually be sufficient to solve the problem completely (i.e., to generate a solution). In other words, propagation and reduction may not be enough to find a unique solution to the problem. In such a situation, there is still a need for a search method to find a solution. Usually most CP languages provide some basic search

algorithms and advanced search facilities for this purpose, while a developer still has the choice of designing and developing a special search strategy for the problem at hand if those standard search strategies are not suitable for it. In any case constraint propagation will continue to be useful in guiding that search.

4.2.3 Value and variable ordering

As described above, a search algorithm tries to construct a solution to a CSP by sequentially instantiating the variables of the problem. The order in which the variables are instantiated and the values are assigned to the variables on backtracking is known to have a potentially profound effect on its efficiency. Choosing the right order of variables and values can noticeably improve the efficiency of constraint satisfaction.

Variable ordering is by far the most important because it can drastically alter the shape of the search tree. The ordering may be either a static ordering, in which the order of the variables is specified before the search begins and is not changed thereafter, or a dynamic ordering in which the choice of next variable to be considered at any point depends on the current state of the search. The advantage of static ordering is that it is very fast because the order is fixed before the search. On the other hand, it does not use any information of the current state during the search. While with dynamic ordering, it is possible to base the choice of the next variable on the information of the current set of instantiations, therefore the variable order can vary from branch to branch in the search tree. In reality, dynamic variable ordering is generally recognized to be more effective

than static variable ordering.

Various heuristics have been investigated for selecting variable ordering. The most common is based on the first-fail principle, which can be explained as: To succeed, try first where you are most likely to fail, which means that one should provoke dead-ends as early as possible to reduce the search effort. The variable with the smallest domain and involved in the greatest number of constraints is favored for instantiation first. Thus the order of variable instantiations is, in general, different from branch to branch, and is determined dynamically. This method is based on the assumption that any value is equally likely to participate in a solution, so that the more values there are, the more likely it is that one of them will be a successful one.

Besides the first-fail principle, there also exist many other heuristics[51, 53] i.e., the least regret principle, the least minimal bound first principle, etc. Depending on the problem, the user can either use those well-developed heuristics or develops a heuristic himself.

Value ordering can have substantial impact on the time to find the first solution and also it can affect the topology of the search tree if the best solution should be found. However, if all solutions are required or there are no solutions, then the value ordering is indifferent, it only affects which parts of the search tree are first explored and therefore

the order in which solutions are found. A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch that leads to a solution is searched earlier than branches that lead to death ends. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking. So in opposition to the first-fail principle, the most common one is based on succeed first principle: try those values first that are most likely leading to a solution. Also there are many other heuristics for value ordering[51, 53], i.e., favoring those values that maximize the number of options available, favoring the value that leads to an easier to solve CSP, etc.

4.2.4 The features of CP

With regard to the flow of control, the constraint programming paradigm is inherently different from imperative programming languages. In conventional imperative languages, no matter if the program is declarative or not, there always exists a fixed scheme that implements a particular algorithm for the program. CP, on the other hand, only states constraints and then asks the solver to find a solution that satisfies all of them. The search is rather data-driven than program-driven.

A major advantage of CP is that it enables us to use the statement of a problem directly to develop a model for that problem. When we design a constraint-based model for a problem, we simply articulate the constraints themselves and then choose variables with values that represent the solution of the problem. Since the model of the problem derives

so directly from the problem representation, there is much less chance of error or incongruence between the semantics of the problem representation and the semantics of the problem solution.

Besides this, it also has some other advantages:

1. CP has some search techniques imbedded in it, the user does not need to concern himself with the management of the constraints or with handling their propagation, so he/she can concentrate on the problem.
2. The constraints of the problem are stated in a natural way, they can be understood easily. Also CP provides a set of predefined constraints that will make the program more efficient and easier to be constructed.
3. Programs can be easily modified and extended. One fundamental design principle in CP is that the problem statement and problem resolution are separated from each other, therefore the user can develop different solving methods for the same model or vice versa, which will make the application easy to use and maintain.

4.2.5 How to solve the problem by CP

The conventional method in CP involves the following steps:

1. Describe the problem, usually in natural language, which is the first step to understand the problem. Try to describe the details of the problem as much as possible. Only after you really understand the problem, you can probably design a

good and correct model for it.

2. Design a model to represent the problem. Usually, there is more than one way to model the problem, you may find several models at the same time. Depending on the features of the particular CP language and the size of the problem, one model may be more efficient than the others.

3. Implement the representation using the facilities provided by the particular CP language. Usually it includes the following parts:

   •Declare the unknowns in the representation as variables and associate finite domains with them, respectively.

   •Create the constraints imposed on those variables. Sometimes redundant constraints should be created in order to improve the performance of the program. In most CP(FD) systems, each constraint is considered separately, the only constraint that is used in combination with others is the variable domain itself. It may often happen that some information is not encoded in the constraints[54].

   • Tell the constraint solver about those constraints. Constraints may be added or relaxed dynamically during the search for a solution.

4. Let the constraint solver search for the solution. The constraint solver can use either standard search algorithms and facilities imbedded in it or search strategies developed by the user to find solutions.

Following the method described above, we can divide a CP program into two main logical parts:

1. The first part is used to establish the model to represent the problem. In the model, we specify all variables, each of which is associated with its finite domain, and their interpretation. And also we construct constraints that are used to express the properties of the solution in the system.
2. The second part is used to define and customize the labeling strategy if the standard search algorithms are not suitable. The labeling strategy (search strategy) is used to search for the solution. This includes both selection of variables to label and value ordering in the selected variable domains.

4.3 ILOG SOLVER[55]

In this project, we use the CP language ILOG SOLVER as our tool to solve the examination scheduling problem. We will give a brief introduction to ILOG SOLVER in this section.

ILOG SOLVER is a commercial package. It is a constraint based object oriented tool. It provides the object technology to model the relationships and constraints of the problem accurately. It allows the user to model one objective function. As the branch and bound

search progresses, the program will locate feasible solutions. The system can then add a constraint that only allows solutions with a lower objective function value.

ILOG SOLVER is a C++ library for solving combinatorial problems and providing optimal solutions. Based on constraint programming, it can also serve as an open framework for integration of other optimization technologies, including linear programming, local search and user-defined application specific heuristics and genetic algorithms. Like C++, ILOG SOLVER is evolving and scalable.

Just like most CP languages, ILOG SOLVER provides some basic functions: for example, it allows cooperation between linear programming and constraint programming; it separates problem modeling and problem solving; it provides domain reduction technology and constraint-programming search algorithms for quickly finding solution, it supports user-defined search strategies that incorporate domain expertise for finding better, solutions faster, etc.

ILOG SOLVER features a unique architecture consisting of three layers: powerful modeling, intelligent control, and fast algorithms.

The modeling layer provides an intuitive way to model problems. This can be achieved by using the following facilities:

1. There are various kinds of decision variables, including real, integer, set, logical and choice variables. Users can easily find the most suitable variable for the problem, i.e., to solve crew-allocation problem that assigns several crew members to a group, the set variable should be used because it can avoid symmetries.
2. ILOG SOLVER provides an extensive set of predefined constraints, including linear, non-linear and logical constraints, and some global constraints such as IlcAllDiff, IlcDistribute, IlcSequence, etc. These predefined constraints can simplify the model and make it clearer, and also they help to construct the model easily.
3. It supports user-defined constraints that can be added simply by extending the provided classes, therefore both predefined and user-defined constraints have the same format, and can be used in the same manner.
4. Metaconstraints allow the user to apply constraints to constraints.
5. It provides an intuitive way to model the constraints and relationships: constraints can be weighted and ordered. Constraints can also be added or relaxed dynamically during the search for a solution.
6. It also provides an easy way to express objective functions, either minimization or maximization of a cost function.

The control layer allows the user to define its own search strategy. First of all the user can choose one or several suitable methods from a wide range of search control

algorithms. One is Depth First Search (DFS) that is the standard search procedure. During the search, the algorithm expands the most recently generated node first. The second algorithm is Best First Search (BFS). It maintains an open list containing the frontier nodes of the tree that have been generated but not yet expanded, and a closed list containing the expanded nodes. Every node has an associated cost value. At each cycle, an open node of minimum cost is expanded, generating all of its children. The children are evaluated by the cost function, inserted into the open list and the parent node is placed on the closed list. The third algorithm is Limited Discrepancy Search (LDS). A discrepancy can be seen as any branch point in a search tree where we go against the heuristic. LDS explores the leaf nodes in increasing order of the number of discrepancies taken to reach them. There always exists a trade-off between the number of discrepancies and the overhead of expanding branches from these discrepancies. The less discrepancies it has, the more efficient the algorithm is. Therefore it will work well if a heuristic only makes a few mistakes during the search for a solution. The fourth algorithm is Depth Bounded Discrepancy Search (DDS) that is a variation of LDS. It makes the assumption that mistakes are made more likely near the top of the search tree than further down. For this reason, it does not count the number of discrepancies but the depth of the last one. It is more efficient if the search heuristic is very good, that is, if it makes mistakes only in the top of the search tree. The last algorithm is Interleaved Depth First Search (IDFS). Like DDS, it is used to reduce the cost of heuristic mistakes at the top of the search tree. IDFS searches in parallel several subtrees at certain tree levels. It traverses depth-first the current subtree until it finds a leaf. If it is a solution, search

terminates. Otherwise, the state of the current subtree is recorded so that it can be resumed later, and IDFS switches to the earliest parallel level, where it selects another subtree and repeats the process.

Secondly the user has the choice to use one of parameter-based basic predefined strategies, e.g., ***IlcGenerate(const IlcIntVarArray, IlcChooseIntIndex, IlcIntSelect)***; And if those predefined strategies are not suitable, the user can still either customize or define a new one. The users can do much more customization in the solution design and define their constraints as long as they have more knowledge of the problem. In fact, with the help of those facilities mentioned above, it is easy to develop complex combinations of algorithms.

Besides those predefined search strategies, the user can choose the potential uses, e.g., compute one, all or the best solution from scratch; update an existing solution to include new information; check a given solution against the constraints, etc.

The powerful algorithmic layer uses cutting-edge solver and tree search engines to deliver fast, reliable solutions. It applies the most efficient algorithms available for information propagation between constraints. And for each predefined constraint it uses state-of-the-art algorithms, for example, flow algorithm for ***IlcAllDiff*** and ***IlcDistribute***, etc.

It is well known that real-world operating problems are highly complex and difficult to solve. These problems can result in enormous search spaces and be impractical to solve with conventional methods. ILOG SOLVER is powerful in solving combinatorial problems. It provides cutting-edge optimization technology for powering scheduling, sequencing, timetabling, configuration, dispatching and resource allocation applications with logical constraints. According to the report from ILOG company, it is particularly useful in the following applications: Telecommunications, Transportation and Manufacturing. The interested reader can find more information in reference [55] and on their website: http://www.ilog.com.

CHAPTER 5

IMPLEMENTATION

Constraint programming languages over finite domain such as ILOG SOLVER or CHIP use constraint solving and consistency techniques inherited from CSP. As mentioned in chapter 4, CP is very useful for building applications where no general algorithm is available and where changes may frequently arise.

The problems in our project fit very well with the constraint programming over finite domains paradigm. In fact, every examination period can be represented by an integer domain variable ranging over the available periods. The rooms for each examination can be identified by a set domain variable. Moreover, we can naturally express some of the constraints of our problems using constraints over finite domains of ILOG SOLVER.

This chapter is organized as follows: section 1 gives a brief introduction of our problems. Section 2 presents the model of the problem. Section 3 shows the representation of the constraints. Several different search strategies will be presented in section 4. Different search space reduction methods are introduced in section 5. Finally different search control algorithms (DFS, DDS and LDS) will be introduced.

5.1 A Brief Overview Of Our Problems

The École Polytechnique de Montréal is not a large university. Every autumn and winter term, roughly 200 courses are offered that have final examinations at the end of the term, and about 4000 students in various programs who have selected one or several courses must attend final examinations.

For autumn term 1999, the problem consisted of planning 192 different examinations in 28 sessions (half-days) during two weeks (two sessions per day). Each session has the same set of available rooms, therefore they have exactly the same capacity: there are 29 different rooms whose capacities are between 28 and 129. And in total there are 1611 desk-seat pairs for each session. The first objective is to find a conflict free examination timetable satisfying all the hard constraints. The second objective is to find a better timetable that satisfies soft constraints as much as possible. As mentioned in chapter 3, the soft constraints are related to each other, some of them even conflict with each other. Thus there always exists a trade-off among them. Among all of the soft constraints of our problem, constraints C6 and C7 (avoid giving a student two examinations on the same day or on consecutive days) are the most important ones. We will construct an optimization function to satisfy C6 and C7 as much as possible, while all other soft constraints will be considered in search strategies. In sections 3 and 4, we will see how to represent and satisfy these constraints.

5.2 MODELING THE DATA

We introduce '***numExams***' integer domain variables to represent the 192 examinations:

IlcIntVarArray exams(manager, numExams, 1, 2*(numPeriods – 1))

where ***IlcIntVarArray*** is a variable data type that is an integer type variable in this case. It is just like an array containing several variables of the same type; ***manager*** is used to handle input and output, memory allocation, and other general services for constrained variables, constraints, and goals; and ***numPeriods*** is the number of available periods, in our case it is 28; the third and fourth parameters represent the initial domain range of each integer variable ***exams[i]***, in our case it is between 1 to 54. Furthermore, the domain of each integer variable ***exams[i]*** will be determined by its release and due date. The aim is to find, for each domain integer variable, a period that satisfies all of the hard constraints, and in the meantime satisfies the soft constraints as much as possible.

One may ask why, since there are only 28 available periods, do you define an integer variable array ***exams*** with initial domain size 54? The reason is that it is an easy way to find out whether or not two conflicting examinations are on the same day or on consecutive days. If we only use 28 continuous integers, say 1 to 28, to represent the domain, it will be difficult to decide whether the two examinations are on the same day or on consecutive days, i.e., there are three examinations in session 1, 2 and 3, respectively. We can't decide if they belong to the same day only by the distance between each pair of them.

After declaring the integer variable array ***exams***, we remove some values from each variable's domain:

exams[i].removeValue(4*n+3)

and

exams[i].removeValue(4*n+4)

where n ∈ [0,13]. Then the size of each variable's domain will be 28, and the values remaining in the domain are 1, 2, 5, 6,..., 4n+1, 4n+2,..., 53, 54 corresponding to each available period, where 4n+1 represents the morning session and 4n+2 represents the afternoon session on the same day and n ∈ [0,13]. Thus if the distance between two examinations is 0, they will be on the same period; if it is 1, they will be on the same day; if it is 3, 4 or 5, they will be on consecutive days. According to this notation, we will define ***costWeight[distance]*** in following parts.

In order to obtain a better timetable, we declare an optimization function variable:

IlcIntVar cost(manager, 0, maxCost)

Where ***cost*** is an integer variable whose domain range is between 0 and ***maxCost*** (defined ourselves). And the optimization function is defined as:

cost=0.5*$\sum_i \sum_j$ examConflict[i][j] * costWeight[distance between examinations i and j]

where ***examConflict*** is a two-dimensional integer array (192×192) and each element of it ***examConflict[i][j]*** represents the number of students in common, namely conflicts, between examinations i and j, and it can be derived from the data given by the university; ***costWeight*** is a one dimensional integer array containing 55 elements (2

(numPeriods-1)+1) and each element of it ***costWeight[distance between examinations i and j]*** represents the cost between examinations i and j if they have one student in common. The value for each element of ***costWeight*** is defined experimentally, i.e., costWeight[0] = ∞, costWeight[1] = 32, costWeight[3] = 2, costWeight[4] = 2, costWeight[5] = 2, and all other costWeight[i] = 0, which means: if the distance of two conflict examinations is 0, the cost is infinite; if the distance is 1, which is the case that a student has two examinations on the same day, the cost will be 32; if the distance is 3, 4,or 5, which is the case that a student has two examinations on consecutive days, the cost will be 2; and the cost will be 0 for all other cases.

Ideally, most students will have their examinations spaced out as much as possible in the final timetable. This objective can be achieved simply in our case, that is, finding a timetable with minimized ***cost***.

The third type of variable we define is: 192 set domain variables, which represent the rooms allocated to each examination.

IlcIntSetVarArray roomsForExam(manager, numExams, room_array)

Where ***room_array*** is the initial domain range of each variable, that is, the available rooms for each examination. As described in chapter 3, the size of some examinations is larger than the size of the biggest room so it is impossible to assign the examinations to a single room. Thus we should use set variables to represent it. The aim is to find, for each domain integer set variable ***roomsForExam[i]***, a set of rooms that satisfies all of

the hard constraints, i.e., satisfying room capacity constraint; and in the meantime satisfies the soft constraints as much as possible, i.e., improving the room utilization.

Besides these variables, we also use several reversible integers ***IlcRevInt***, which are used to remember the old value of an old state, i.e., we define a reversible integer:

IlcRevInt ** sessionSize;

Where ****sessionSize*** represents the capacity of a session. When an examination is scheduled in a session, the ****sessionSize*** should be adjusted: more accurately, it should be reduced. During the search for a solution, the ****sessionSize*** will be changed any time an examination is assigned to the session. Different from an ordinary integer, the ****sessionSize*** will remember all its old values in those old states. When a backtrack is performed, the ****sessionSize*** will restore to the corresponding old state, thus retrieving the old value. As one can expect that reverse integer will make the programming easy and effective. All others reversible integers will be introduced in following sections when they appear in the program.

École Polytechnique de Montréal gives us several data files. The data includes: all of the courses offered in autumn 1999 and their name, 192 of them have final examination; 28 available periods with the same length: 2 and half hours; 29 available rooms with different capacity from 28 to 129, their name and location; the number of students for each examination; the number of conflict examinations for each examination; and the

number of students in conflict between each pair of examinations. And also they give us the results obtained from their program.

5.3 REPRESENTING THE CONSTRAINTS

As described in chapter 3 we have divided the constraints into two groups: hard constraints and soft constraints. The following parts will show how to deal with the constraints in our model.

For constraint C1 (Examination incompatibilities): we have two ways to represent it. If only two examinations have at least one student in common, then there is a simple way as following (not equal constraint):

manager.add(exams[i] ≠ exams[j])

Namely, the two examinations can't be scheduled in the same period. If several examinations (more than 2) conflict with each other, we can use the global constraint ***IlcAllDiff()***, which is built-in in ILOG SOLVER, to express it as follow:

manager.add(IlcAllDiff(tempExams))

where ***tempExams*** is an array containing the examinations conflicting with each other. The function posts a constraint: the examinations in the array ***tempExams*** should be scheduled to different periods. Logically, the single global constraint ***IlcAllDiff()*** on n variables is equivalent to $(n-1)n/2$ instances of the 'not equal' constraint, but in terms of performance, that single constraint is much more efficient if n is large enough[55].

For constraint C2 (Session capacity): when assigning an examination i to a session j, it must satisfy the following condition:

numStudForExam[i] <= *sessionSize[j]

where ***numStudForExam*** is an integer array containing ***numExams*** elements, each of which represents the number of students for each examination. The constraint says if the number of students for examination i is bigger than the capacity of session j, then examination i can't be scheduled to session j.

For constraints C3 and C4 (Room constraints): in order to assign room(s) to an examination or schedule several examinations to a room, we use two reverse integers:

IlcRevInt *roomCapacity[period][roomIndex]

IlcRevInt *numStudLeft[examIndex]

****roomCapacity*** is a two dimensional array, each element of it representing the current capacity of the room (the ****roomCapacity*** will be adjusted any time an examination or part of an examination is assigned to the room). ****numStudLeft*** is a one dimensional array, each element of it representing the current number of (unassigned) students of an examination, i.e, when part of an examination is scheduled to a room, the current number of students will be reduced.

With the help of the reverse integers, we can construct C3 and C4 easily. If ****numStudLeft[examIndex]*** <= ****roomCapacity[period][roomIndex]***, it is possible to schedule the examination '***examIndex***' to the room '***roomIndex***' which belongs to session '***period***'. If ****numStudLeft[examIndex]*** is larger than any element of

roomCapacity[period], or it is the prerequisite from the university that the examination '***examIndex***' must be scheduled to several rooms, then we need several rooms to hold the examination. Anyway in each situation, they should satisfy the following condition:

$$*numStudLeft[examIndex] <= \sum_{roomIndex} *roomCapacity[period][roomIndex]$$

that is, when the total capacity of several rooms is larger than the number of students of an examination, then it is possible to schedule the examination to the several rooms that belong to session '***period***'.

For constraint C5, if a room is required to hold at least two different examinations, we divide the room into two parts, and simply regard them as two different rooms. Then post a constraint for each pair of these rooms: they can't be in the same set.

```
manager.add(IlcCard(IlcIntersection(roomsForExam[k],tempRooms))<=1);
```

where ***roomsForExam[k]*** is an integer set variable as described is section 2: ***tempRooms*** is another integer set variable containing two elements: two rooms which originally come from the same room. The function ***IlcIntersection()*** returns the intersection of two sets, while ***IlcCard()*** returns the number of elements in the set. Thus the constraint says that for any examination, it is impossible to have both of the rooms coming from the same one as part of its solution simultaneously.

For constraint C6, C7 and C10 (examinations for each student should be spread over the examination period): as described in section 2, we define an optimization function to achieve the objectives. In the function, we combine C6 and C7 together, and give them

different weights, which show how important each item is. For example, in our case, the contribution to the total cost related to C6 and C7 are 32 (for examinations on the same day) and 2 (for examinations on consecutive days), respectively. As long as we can find solutions with minimized cost, C6 and C7 will be satisfied as much as possible. And hence C10 will be satisfied partly.

As we know there always exists a trade-off among the soft constraints. In our case, C6 and C7 are the most important ones, we will satisfy them first, and then all other soft constraints will be satisfied if they don't affect C6 and C7.

For constraint C8 (morning session constraint): we can satisfy it in the search strategy, i.e., when an examination needs to be fixed, we always select the morning session if both morning session and afternoon session have the same cost.

For constraint C9 (large size examination constraint): it can also be satisfied in the search strategy, i.e., selecting the examination with a large number of students first will cause large examinations to be scheduled first, and most likely they will be assigned to early sessions as long as the assignment leads to a minimized cost. We use C8 and C11 to break a tie.

For constraint C11 (all examinations should be scheduled to early sessions): it conflicts directly with C8 and we satisfy C8 first. When two morning sessions or two afternoon sessions have the same cost, we always select the early session as the solution.

For constraints C12 and C13 (room utility and uniform distribution of student constraints): we can use a room scheduling algorithm to satisfy them, i.e., using the best-fit algorithm, which tries to find the most suitable rooms, regarding the room utility and the request of uniform distribution of students, for an examination. We will give more details in the following section.

5.4 Search Strategies

Good search strategies for selecting variables and values should be efficient to find solutions, but there is no single rule of thumb for discovering what is a good strategy. You should try out different ways of solving a given problem to find the one most advantageous to your situation. However, there are two main kinds of strategy: the one used for constraint satisfaction problems, and the one used for optimization problems.

In either kind of strategy described above, it is best to select variables dynamically. Dynamic variable ordering is generally recognized as a key factor of success: rather than fixing an ordering a priori, the most promising variable is chosen according to the information available at that moment. Within this scheme, the first-fail-principle[56]

seems to emerge as a good strategy. It states that one should provoke dead-ends as early as possible to reduce the search effort. One example[55] is that when we consider satisfaction problems, a strategy that is often successful for selecting variables is to choose the most constrained one--the one with the smallest domain and involved in the greatest number of constraints (selected dynamically). In effect, those variables are where propagation should be the most significant and, as a consequence, where entire portions of the search space can be pruned as quickly as possible.

When considering optimization problems, strategies based on generation often first generate those variables and their values that are statistically the most likely to lead to low-cost (best) solution.

In this section we address the order in which variables are selected and their values tried. As described in chapter 4, variable ordering is by far the most important because it can drastically alter the shape of the search tree. Value ordering does not affect the topology of the tree generally, but rather which parts of the tree are first explored and, ultimately, how soon a very good or even optimal solution will be found.

Our examination scheduling problem is an optimization problem. We have developed several strategies following the guidelines described above, but specializing them for our particular problem, and we will present them one by one to show how to select the next variable (examination) and how to instantiate it. But first of all, let us see how to

decompose the examination scheduling problem into small problems and how to solve them.

The examination scheduling problem involves two sub-problems: assign a session to each examination without any time conflict; find a room or several rooms for each examination. There are two way to solve the problem: one is to combine the two sub-problems and solve them together. The advantage is that it is possible to find an examination timetable with a better room allocation, i.e., the rate of room utilization is high. The disadvantage of the method is that the size of the problem will be enlarged and also it will waste some time on finding rooms for examinations if backtrack is needed, thus it will spend more time to find a solution, i.e., when an examination is fixed to a session, we should assign a room or several rooms for it right away. And when backtracking over this search node and fixing the examination to another session, we should find another set of rooms for it again. One may argue that assigning rooms to an examination early will help us to prune the search space as early as possible, i.e., when an examination is fixed to a session, we know the partial solution satisfies all constraints so far, but if there are not enough rooms in the session for the examination, then we know the partial solution will not lead to a final solution, and a backtrack will be executed. It is true, but it is only one way to prune the search space. As we will discuss in second method, we can use '***sessionSize***' to achieve the same objective without allocating rooms for any examinations.

The second method is: solve the two sub-problems in two phases. In the first phase, we find a session for each examination. In the second phase, we assign a set of rooms for each examination. Instead of '***roomSize***', we use '***sessionSize***', which is the total capacity of all rooms in that session and will change during search for a solution, to limit the number of examinations that can be scheduled to the session, i.e., the constraint C2. In this way, we can guarantee that if we find a solution in the first phase, we can also find a room timetable for examinations in the second phase. One advantage is that the size of the problem is limited because the second sub-problem will be solved only after the first sub-problem is solved. The other advantage is that different variable selection rules can be applied to improve the results of room scheduling, i.e., we can choose a rule different from that of the first phase.

5.4.1 Search strategies for the first phase: examination-session timetabling problem

5.4.1.1 Choose variables: examination

Basically, there are two kinds of method for choosing the next variable: the static method and the dynamic method. As their names suggest, the static method selects the next variable according to an ordered variable list while the dynamic method selects the next variable according to the current state, i.e., one does not know which variable will be chosen before the current variable is fixed.

Static method

We sort the examination list according to the number of students of each examination in descending order. and select the next examination in order, from the first one to the last one. It is based on the assumption that a large examination usually has a large number of conflicts (common students) with other examinations, and it is difficult to find a right period for it, so it should be scheduled as early as possible. The method is easy, and runs very fast because it only sorts the examination list once and does not spent much time on selecting the next examination.

Dynamic method

Small-domain first: select the next examination that has the smallest domain, in other words select the one that has the smallest number of available periods. The number of students are used to break ties, i.e., when the size of two examination domains is the same, we select the one with a large number of students because these two examinations are likely to have more students in common with other examinations, thus difficult to be scheduled later. The assumption is that the examination with a small domain has less opportunity to succeed, thus it should be scheduled first.

Random selection: select the next examination randomly among unfixed examinations. The aim of the method is to change the shape of the search tree so that one can search solutions in a different part of the search space. For sure each time you run the program,

you will not get the same results because the shape of the search tree is different every time. Also it can be used to compare with other methods to see how well they perform.

Large number of conflicts first: select the next examination that has the largest number of conflicts (the total number of students in common) with the fixed examinations. For the first examination, we select the one that has the largest number of conflicts with all other examinations.

More formally, Let **A={Ei}** be set of the fixed examinations, while **B={Ej}** be set of the unfixed examinations. For each unfixed examination, we define a cost function that counts how many students are involved in conflict with the fixed examinations, then we select the examination that has the largest cost:

$$Cost_j = \sum_i examConflict[i][j]$$

where ***i*** is from the set **A** while ***j*** is from the set **B**, ***examConflict[i][j]*** represents the number of students in common between examinations ***i*** and ***j***.

The total number of examinations in conflict are used to break ties, i.e., if two unfixed examinations have the same cost, we choose the one that has the largest number of examinations in conflict with the fixed examinations because its domain size is smaller than that of the other and thus it has less chance to succeed.

Following the definition given above, for each unfixed examination, we define a function to count how many fixed examinations conflict with it, then we select the one with largest count in case of ties:

$$Count_j = \sum_i M_i$$

$$M_i = \begin{cases} 0 & when\ examConflict[i][j] = 0 \\ 1 & when\ examConflict[i][j] \neq 0 \end{cases}$$

where i is from the set **A** and j is from the set **B**.

For an optimization problem, the basic rule is: generate those variables that are the most likely to lead to a low-cost solution first. The point here is that assigning a period to the examination with the largest number of conflicts later will likely increase the total cost of the final schedule dramatically, so it should be scheduled as early as possible because there are more periods available at an early stage.

Least regret strategy: we select the next examination that has the largest cost difference between its two best available periods. Indeed, if we wait until the best-cost period is no longer available, then the difference in cost for the schedule will be large.

5.4.1.2 Choose Values: periods

Usually when choosing a period for an examination, the impact of both fixed examinations and unfixed examinations should be considered in some way. The following figure shows the relationship among the fixed examinations, the unfixed examinations and the selected examination just before it is fixed.

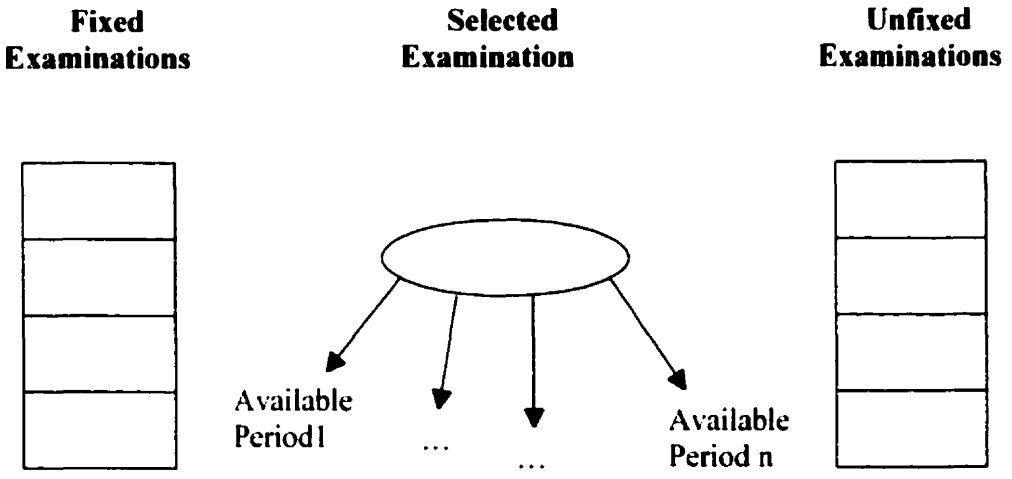

Figure 5.1 The relationship among the fixed, unfixed and selected examination(s)

Depending on different threshold functions, we adopt three methods for choosing a period for the selected examination as follows:

The first method concentrates on reducing the cost of the partial solution; it only concerns the effect of the fixed exams. It is based on the assumption that an optimized partial solution will lead to a better final solution (of course it is not necessarily the best solution). We always select the period for the selected examination that has the smallest

cost with the fixed examinations. For the first examination, we try every possible period in turn to find the best period.

Let ***k*** be the index of the selected examination from set **B**. For each available period ***l*** of the selected examination, we define a cost function that calculates the cost of the partial solution, then we select the period for the examination ***k*** that has the smallest cost:

$$cost_l = \sum_i examConflict[i][k] * costWeight[distance\ between\ i\ and\ k]$$

where ***i*** is from the set **A**, ***examConflict[i][k]*** represents the number of students in common between examinations ***i*** and ***k***, and ***costWeight[distance between i and k]*** represents the cost between examinations ***i*** and ***k*** if there is one student in common, in our case it can be 0, 2 or 32 depending on the distance between the two examinations.

In case of ties, we choose a morning period or an early period depending on the situation, i.e., if a morning period and an afternoon period have the same cost, we will choose the morning period first, while if two morning periods or two afternoon periods have the same cost, we will choose the earlier one first. This way we can satisfy the soft constraints C8 and C11. Usually this only changes the search order without changing the shape of the search tree. But if the small-domain first strategy is adopted for selecting the next examination, it may also change the shape of the search tree because the domain size of the unfixed examinations depends on the selected examination.

The second method mainly considers the effect of the unfixed examinations. For each available period ***l*** of the selected examination, calculate how many unfixed examinations conflicting with the selected examination can be assigned to the period. Then select the period with the smallest number of examinations. The point is that this leaves more opportunities for other unfixed examinations and it is therefore easy to find feasible solutions. This strategy is very similar to small-domain first strategy except that it is used to choose a value instead of a variable.

For each available period ***l*** of the selected examination, we define a function to calculate how many conflict unfixed examinations can fit in the period. Then we select the period for the examination ***k*** having the smallest count:

$$Count_l = \sum_j M_j * N_{lj}$$

$$M_j = \begin{cases} 0 & when\ examConflict[k][j] = 0 \\ 1 & when\ examConflict[k][j] \neq 0 \end{cases}$$

$$N_{lj} = \begin{cases} 0 & when\ period\ l\ is\ not\ in\ the\ domain\ of\ examination\ j \\ 1 & when\ period\ l\ is\ in\ the\ domain\ of\ examination\ j \end{cases}$$

The first method is used to break ties, i.e., when two periods have the same count, we choose the one that incurs the least cost with the fixed examinations because it will optimize the partial solution.

The third strategy is a method that chooses a period at random: the impact of neither the fixed examinations nor the unfixed examinations is considered. It is used as a comparison basis with other methods to see how important the threshold function is.

As mentioned above, usually value ordering does not affect the topology of the tree, but rather which parts of the tree are first explored and, ultimately, how soon a very good or even optimal solution will be found. However, if all solutions are required or there are no solutions, then the value ordering is indifferent.

5.4.2 Search strategies for the second phase: examination-room timetabling problem

As described above, we decompose the examination scheduling problem into two sub-problems. First solve the examination-session timetabling sub-problem, and if there is a solution, namely every examination has a fixed period, then we continue to solve the examination-room timetabling sub-problem.

5.4.2.1 Choose variable: examination

This is a relatively easy process compared with that of the first phase. And depending on whether the selected examination is fixed at one time or through several phases, there are two different ways to choose the next examination.

The usually static method, select the next examination according to their size (the number of students), is good enough. First the examination list is sorted in decreasing order. When the next examination is needed, the first examination is removed from the list because the size of the examination is large and it needs more rooms to hold it, thus it should be scheduled first.

The other way is very similar to the static method except the selected examination is not removed from the list. Instead it is put back in the examination list in the right place according to its size, namely after the examination gets its partial solution (several rooms are assigned to it) and becomes a small size examination. The point is that the large rooms should be retained for large size examinations so that they can be scheduled into fewer rooms. We will describe it in detail in next section: choose value.

5.4.2.2 Choose value: rooms

It seems that for the examination-room scheduling problem, it is more important to find a good search strategy for choosing values in order to improve the quality of the final schedule, i.e., the rate of room utility, distributing students evenly, etc..

Three methods are investigated: ***the largest-first method, the first-fit method and the best-fit method***. Besides ****roomCapacity[period][roomIndex]*** and ****numStudLeft[examIndex],*** we also define and use another reverse integer array:

roomOrder[period][i] to keep track of the room order list for each period so that we can choose rooms easily. In our case, there are 28 room order lists, in other words, each session (period) has its room order list.

At the beginning of the search (or before the search), the examination list and the room order lists (***roomOrder***) should be sorted in decreasing order according to their size. And during the search ***roomCapacity***, ***numStudLeft*** and the examination list will be adjusted accordingly. Depending on the methods, the room order list (***roomOrder***) may also be sorted during the search for solutions. We will describe these methods one by one as follow.

The largest-first method

After an examination is selected, a largest room or several largest rooms big enough to hold the examination should be assigned to it right away. The point is that using large rooms first will reduce the number of rooms for the examination, and finally it may lead to use fewer rooms for all examinations. In outline form, the algorithm looks like the following:

- While there exists unfixed examination(s) (its size is not zero) in the list
  - Select an examination from the head of the examination list, and according to its period, select the corresponding room order list (***roomOrder[period][index]***).
    - Select the first available room from the room list, and if it is in the domain of the selected examination, then assign it to the selected examination. Adjust

the size of the room (***roomCapacity[period][roomIndex]***) and the size of the examination (***numStudLeft[index]***), also sort the room order list.

- If the size of the selected examination becomes zero (***numStudLeft[index]*** = 0), then the assignment for the examination has succeeded. Remove the examination from the examination list (in fact it is put at the end of the list). Otherwise repeat the above step (select the first available room) until ***numStudLeft[index]*** = 0.

The first-fit method

Like ***the largest-first method***, the next selected examination is always the one with largest size. The difference is that ***the first-fit method*** does not fix the selected examination at one phase, instead it only solves the problem partially, namely assign room(s) to the examination and then put the selected examination back into the examination list. Later when it becomes the largest examination in the list, it will be selected again. It should be mentioned that during the search, the room order list remains the same and does not change (it is sorted before the search), and also if there are several rooms, each of which is big enough to hold the examination, we always choose the first available one in the list, and assign it to the examination. There are two reasons for using the method. First, the large rooms should be reserved for the large size examinations so that each examination will use fewer rooms. Second, the examinations will use large rooms first, and leave those small rooms untouched (if the total number of students is

much less than the capacity of the rooms), thus it uses rooms as little as possible, also it can improve the rate of room utility. The actual algorithm works as follows:

- While there exists unfixed examination(s) (its size is not zero) in the list
  - Select an examination from the head of the examination list, and according to its period, select the corresponding room order list (****roomOrder[period][i]***).
    - Go through the room order list, select the first available room, which is in the domain of the examination and is big enough to hold the examination, then assign it to the examination. Adjust the size of the room and the size of the examination, and also sort the examination list.
    - If the largest room can't hold the examination, then divide the examination into two parts: the first part can be held by the largest room (it is assigned to the examination). Adjust the size of the room and the size of the examination.
    - For the second part, if there exists a room that can hold it, then we put the examination back into the list (it will be scheduled late because it becomes a small examination now). Sort the examination list.
    - If there is no room that is large enough to hold the second part of the examination, divide the examination again and repeat the above two steps.

The best-fit method

Like the two methods mentioned above, we always select the largest size examination as the next one. It is different from ***the first-fit method*** in three aspects: first it fixes the selected examination in one phase, second the room order list is sorted during the search,

third the best fit rooms will be selected for the selected examination. Of course there must be a way to define what is the best fit. The aim of the method is: try to improve the rate of room utilization as much as possible while spreading the students from the same examination as evenly as possible. As we can expect, the method will likely use small rooms and may leave some large or middle size rooms untouched. The algorithm is now outlined:

- While there exists unfixed examination(s) (its size is not zero) in the list
  - Select an examination from the head of the examination list, and according to its period, select the corresponding the room order list (***roomOrder[period][i]**).
    - If the size of the examination is smaller than the size of the first available room that is in the domain of the examination, then try to find a best room for it (the smallest room that is big enough to hold it).
    - If the examination is equal to the first room, then the room is assigned to it.
    - If the size of the examination is bigger than the size of the first room, get the second available room. If the size of the examination is smaller than the size of two rooms, then try to find two best rooms for it.
    - If the size of the examination is equal to or bigger than the size of two rooms, then the first room is assigned to the examination.
    - Repeat the above four steps until the selected examination is fixed.

During the search we should adjust the following lists (***roomOrder[period][index]**, ***roomCapacity[period][index]** and ***numStudLeft[examIndex]** any time when a room

is assigned to the selected examination. Also we should sort the examination list after the selected examination is fixed.

Considering the trade-off between room occupancy rate and uniform distribution of students of an examination, we construct a selection function, which is used to select suitable (best-fit) room(s) for an examination. Suppose ***n1*** is the number of students of an examination, ***n2*** and ***n3*** are the capacity of two rooms, respectively, ***n2*** >= ***n3*** and ***n1*** <= ***n2*** + ***n3***, then the function is defined as following:

$$F = n2 + n3 - n1 + balanceFactor * n2 / n3;$$

we always select two rooms that will minimize ***F***. For example there are two sets of rooms: ***n2*** = 60, ***n3*** = 55; ***n2*** = 90, ***n3*** = 20 and an examination ***n1*** = 100. Which room set should be chosen for the examination?

According to the function: suppose balanceFactor = 2.0

$$F1 = 60 + 55 - 100 + 2 * 60 / 55 = 15 + 2.2 = 17.2$$

$$F2 = 90 + 20 - 100 + 2 * 90 / 20 = 10 + 9 = 19$$

so we will select the rooms: n2 = 60 and n3 = 55

5.5 Search Space Reduction

The question is how to reduce the search space? To answer the question, let's see how to assign a set of periods to the examinations first. Generally speaking, we fix examinations one by one, i.e., first select an examination from the examination list, then choose a

period and assign it to the examination. Repeat the process until all examinations are fixed without conflict. Figure 5.3 illustrates the process.

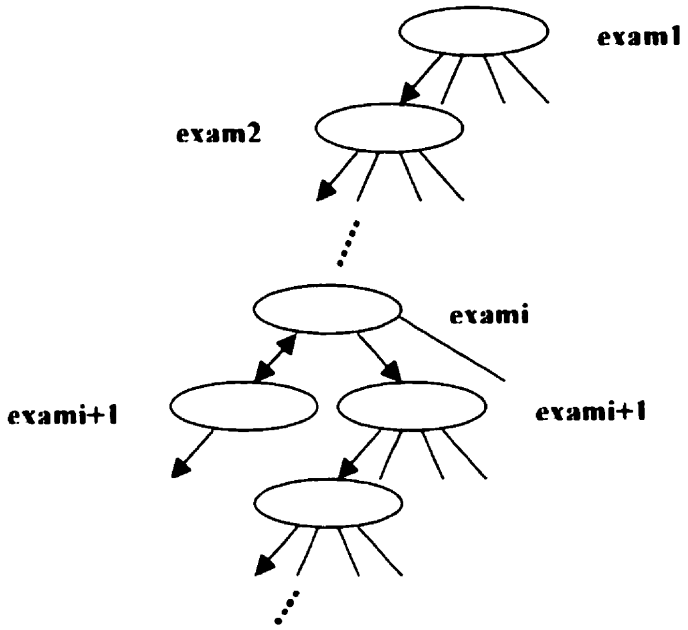

Figure 5.2 The process of fixing examination during the search for solution

Where each node (an ellipse and several lines) represents an examination and its domain (the available periods), a line with down-arrow means the period is assigned to the examination while a line with up-arrow means backtracking to up node has happened when there is no solution in the following part of sub-space.

In fact, what we do is to apply a search strategy, and let ILOG SOLVER to handle the search process. It will go through the search space to find solutions. During the search, when necessary, ILOG SOLVER will backtrack automatically and follow another path. This happens in two situations: the domain of an examination is empty, that means there

is no solution, or at some point the cost of partial solution is larger than that of solution found before, and so we do not need to go further.

Clearly, if we can cut some branches (available periods) at each node, then the search space will be reduced dramatically. Based on this idea, we define several score functions, which are used to sort the available periods at each node. And according to a threshold, some available periods are removed from the domain of the examination. Three different score functions are described as following:

The first one is exactly the same as the first method described in section ***5.4.1.2 Choose Values: Periods***. It only considers the effect of the fixed examinations. The available periods of the examination are sorted in ascending order according to the cost with the fixed examinations:

$$cost_l = \sum_i examConflict[i][k] * costWeight[distance\ between\ i\ and\ k]$$

The second score function is the same as the second method described in section ***5.4.1.2 Choose Values: Periods***. It mainly considers the effect of the unfixed examinations.

The third score function is more complicated compared with the first two. It considers the impact of both the fixed examinations and the unfixed examinations. For each available period of the selected examination, assign it to the examination, then try to find a feasible solution (fix the unfixed examinations) according to the adopted search

strategy described in section ***5.4.1 Search strategies for first phase: examination-session timetabling problem***. Usually different feasible solutions will have different costs, if there is no solution for an available period of the selected examination, then the cost will be infinite (∞). Finally sort the available periods of the examination in ascending order according to the cost of their feasible solutions.

5.6 PUTTING IT TOGETHER

In the above sections (***5.4 Search Strategies*** and ***5.5 Search Space Reduction***), we present different search strategies, which will guide ILOG SOLVER to search solutions, and search space reduction methods, which will limit the search within some sub-search spaces. Now let us see how they can be put together and be used to solve our optimization problem.

In fact, no matter what search strategy we adopt and what search space reduction method we use, the basic algorithm is the same.

- While there are some unexplored search spaces, search and find the next better solution.
  - Select the next examination according to the adopted search strategy.
  - Sort all available periods of the selected examinations according to the score function, and remove some periods from its domain according to a certain threshold.

- Select a period (depending on which search control algorithm you use) from the sorted list and assign it to the selected examination.
- Repeat the above three steps until a better solution is found.

There are several search control algorithms, which control how to explore the search space (tree): which parts of the tree are first explored, i.e., Depth First Search (DFS), Limited Discrepancy Search (LDS), Depth bounded Discrepancy Search (DDS), etc.

DFS is the default search control algorithms. It expands the most recently generated node first, and always follows the best path first according to the search strategy.

A discrepancy is a right move in the path from the root of the search tree to the current node. The intuition is that a corresponding to heuristic choice is better than a right move. Thus, by limiting the number of discrepancies, we try to stick close to the search heuristics. LDS divides the search tree into strips. Strip k correspond to the open nodes of the search tree with a number of discrepancies between **k*step** and **k*(step + 1) - 1**. The search should explore strip 0, then strip 1, then strip 2, and so on.

DDS is a variation of LDS. It makes the assumption that mistakes are made more likely near the top of the search tree than further down. For this reason, it does not count the number of discrepancies but the depth of the last one. It is more efficient if the search heuristic is very good, that is, if it makes mistakes only in the top of the search tree. DDS

divides the search tree into steps. In the first pass, it explores nodes with all discrepancies appears with a depth less than step. In the second pass, it does the same with a depth less than 2*step, and so on.

It should be observed that these search control algorithms will change the ordering of exploring sub-search trees, and thus have an impact on how soon a good solution will be found. But if we need to find the best solution, then there is no difference among them, since they should explore the whole search tree before they can confirm it.

CHAPTER 6

EXPERIMENTAL RESULTS

As described in Chapter 5, there are several ways to select the next variable and to choose the suitable value for the selected variable. Combining variable and value selection methods together, we can get several different search strategies, that is several different implementations.

Experiments are carried out on a SUN workstation with 1 G memory and the CPU speed is 333 MHz.

6.1 THE RESULTS OF EXAMINATION-SESSION TIMETABLING PROBLEM

6.1.1 Comparison among different variable selection methods

Compared to value ordering, variable ordering is the most important because it can drastically alter the shape of the search tree, and leads to a total different search process. thus it will have a huge impact on how soon a very good or even optimal solution will be found.

In this section, we construct five implementations. All implementations use the same value ordering rule to select the period for the chosen examination: selecting the period

that has the smallest cost with the fixed examinations (see section 5.4.1.2 Choose Values: periods). Variable selection methods described in section 5.4.1.1 are used in different implementations. The first implementation adopts the large-number-of-conflicts-first rule to order the examinations. The second implementation uses the small-domain-first rule to select the next examination. The third implementation applies least-regret. The fourth implementation uses a static method. The fifth implementation orders the examination list in a random way. Every implementation can solve the problem, but with totally different results, e.g., the solutions of some implementations are much better than that of others, the time needed to find better solutions are different from one implementation to another implementation. The solutions of these implementations will be given next, followed by comparison and discussion.

As mentioned in Chapter 5, the total cost of the final solution is dependent on the number of students who have two examinations on the same day or on consecutive days. By changing the value of the 'costWeight[distance]', the solutions will be different even if the same implementation is used. The value of CostWeight[1] represents the cost contributing to the total cost if a student has two examination on the same day, while the value of CostWeight[3, 4 or 5] represents the cost if there is a student having two examinations on consecutive days. The ratio between CostWeight[1] and CostWeight[3, 4 or 5] is more important: the larger the ratio is, the less the number of students having two examinations on the same day are in the final results. In our implementation,

CostWeight[3,4 or 5] is fixed to 2, while CostWeight[1] can be changed so that different solutions can be obtained.

The purpose of the section is to make a comparison among variable selection rules, it is enough to find the first better solution.

Table 6.1 summarizes the results of the first implementation. There are four columns: the first column is CostWeight[1], which is an input parameter, the second column shows the number of students having two examinations on the same day while the third column shows the number of students having two examinations on consecutive days, the last column shows the total cost of a solution, which can be obtained as following:

totalCost = 2 * column three + column one * column two

Note that with different input parameter (costWeight[1]), the results is different, with the increasing of costWeight[1], the number of students having two examinations on the same day decrease while the number of students having two examinations on consecutive days increase, in other words, there always exists a balance between column two and column three. To find the first better solution, the program takes about 56 seconds and there is no failure.

It should be mentioned that for each costWeight[1], there are 28 first better solutions corresponding to 28 periods available for the first selected examination. The table 6.1 only shows some better solutions.

Cost Parameter	nb Same Day	nb Consecutive Day	Total Cost
16	94	1793	5090
	103	1646	4940
	107	1582	4876
32	83	1987	6630
	85	1832	6384
	86	1783	6318
64	71	2353	9250
	72	2332	9272
	78	2191	9374
128	67	2290	13156
	71	2219	13526

Table 6.1 The results of the largest-number-of-conflicts first implementation

École Polytechnique gives us their final result as follows:

The number of students who have two examinations on the same day: 100

The number of students who have two examinations on consecutive days: 2000

From our initial solution (Table 6.1), we can see that our results are slightly better than theirs. For example, there are 86 students having two examinations on the same day, 1783 students having two examinations on consecutive days.

For the small-domain-first implementation, the program runs faster than the first implementation: it needs about 44 seconds to find the first better solution, the reason being that it takes less time to choose the next examination and has no failure during search for the first better solution. Table 6.2 summarizes the results of the second

implementation. It illustrates the similar feature as that of table 6.1 except that the results are not as good as that of the first implementation. Comparing with the results of École Polytechnique, it is still good or at least they are at the same level. Anyway it is only the initial solution, not the final solutions.

Cost Parameter	nb Same Day	nb Consecutive Day	Total Cost
16	96	2066	5668
	107	1903	5518
	121	1772	5480
32	81	2275	7142
	86	2175	7102
	91	1976	6864
64	74	2514	9764
	81	2166	9516
	89	2091	9878
128	73	2380	14104
	75	2313	14226

Table 6.2 The results of the small-domain-first implementation

As we know, the small-domain strategy it may not be suitable for finding optimal solution. In our case, there are 192 examinations. 28 available periods within 14 days and the capacity of each period is 1611. Let us first check if it is possible to find a feasible solution with no students who have two examinations on the same day. Put differently, the question is whether there exists a set in which the examinations are in conflict with each other (each pair of examinations have common students and can't be scheduled to the same period) and whose size is bigger than 14? After calculation, we find that the size of the largest such kind of set is 19, it is larger than 14, that means we

can't avoid situations where students have two examinations on the same day. Furthermore we can deduce that there does not exist a feasible solution if the total number of available periods for examinations is less than 19.

Now we ask how many available periods are sufficient to solve the problem. As we know, the number of feasible solutions will decrease if the total number of available periods is reduced, thus it is more difficult to find a feasible solution. In our case, if there are only 19 available periods, then no solution can be found because of the limitation of either session capacity or running time. If the total number of available periods is bigger than 19, we can find feasible solutions within the time limit. The question is how soon it can be found. For example, suppose there are 20 available periods, if the small-domain-first strategy is used to select the next examination, then we can find the first feasible solution within 56 seconds. During the search the number of choice points is 182901 and the number of fails is 100463. While if the first implementation is used, the first feasible solution can be found within 225 seconds: the number of choice points is 548650 and the number of fails is 440129.

The reason why the first implementation is not suitable in this more constrained situation is that scheduling those examinations that may have more choices (available periods, large domain) first may make the domain of those small domain variables even smaller, thus leave them less chance of success. That means before finding a solution, the program will make a lot of wrong choices, it will backtrack many times and waste much

time. Although scheduling the examination with the smallest-domain-first still reduces other variable's domain size, the domain of those variables is relatively large and they have more choices and thus it is relatively easy to find a feasible solution.

In our problem, there are 28 available periods that is much larger than 19, thus each examination has a large initial domain. The domain reduction based on constraint propagation will have less chance to cause backtracking. In other words, even small domains may have many choices and it will not be the bottleneck in finding solutions any more. In this situation, it is not necessary to use small-domain-first strategy, especially when dealing with optimization problems because small-domain-first strategy focuses on finding feasible solutions and does not concern itself with how to select the next variable for finding a better solution. Since there are many choices, you can focus on optimizing the solutions and choose the most suitable strategy for your problem. This explains why the first implementation works well.

Table 6.3 summarizes the results of the third implementation that uses the least-regret rule to select the next examination. Comparing with the first two implementations, it takes much time to find the first better solution, about 480 seconds, the reason is that the program has to spend a lot of time on finding the best two periods for each unfixed examination before it can make a decision which examination should be selected. From the table we can see that the results are as good as that of the first implementation and

are better than that of the second implementation. This is because some rules are used to reduce the total cost during the search process.

Cost Parameter	nb Same Day	nb Consecutive Day	Total Cost
16	103	1670	4988
	108	1661	5050
	112	1602	4996
32	79	2020	6568
	80	1926	6412
	84	1790	6268
64	73	2171	9014
	77	2149	9226
	83	2075	9462
128	69	2260	13352
	74	2311	14094

Table 6.3 The results of the least regret implementation

Although the results of the first and third implementations are at the same level, they use a different approach to search the solutions. Both of them try to first generate those variables that are the most likely to lead to low-cost solution but in a different way. However the two methods are related to each other somewhat, i.e., in third implementation, we use the large number of conflicts to break the ties when more than one examinations have the same cost difference, which means that at least the first seven selected examinations are the same in both implementations because the cost difference for them are all equal to zero (there are 14 available days). From the results we can't tell which approach is better in our case, but the run time performance of the first

implementation is much better than that of the third implementation because it needs to do some extra calculations to select the next examination.

The results of the fourth implementation are summarized in table 6.4. The program takes about 50 seconds to find the first better solution with 1 failure during the search, it is slower than the second implementation, and is faster than the rest of the implementations. The results are not as good as that of the first and third implementations, but are at the same level as that of the second implementation. As we know, the examination with a large number of students usually has a large number of conflicts with the fixed examinations, but this is not always true, thus selecting the next examination according to their size sometime may lead to such a situation: the examination that has the largest number of conflicts with the fixed examinations is left to later, which in turn increases the total cost of final solution quite a bit. This is also true for the second implementation. The examination with the smallest domain is not necessary to have large number of conflicts with the fixed examinations, it only means that more examinations conflicting with it have been scheduled to different periods and thus it may not make a big difference to the total cost whether it is scheduled early or later. For example, there are 8 periods in total, each period has already had some examinations. Suppose now there are two candidates, the first one has two students, it has two available periods: periods 3 and 6, the other is a big examination and has the largest number of conflicts with the fixed examinations, it has three available periods: period 2, 3 and 6, also two candidates conflict with each other. Coincidentally period 3 is

the best period for both examinations, the cost difference between period 3 and 6 for the first candidate for sure is not big, at most 34 in our case, while there may be a great cost difference if the second candidate is scheduled into period 2 or 6. Just like the description of the third implementation, it is better to select the second candidate first.

Cost Parameter	nb Same Day	nb Consecutive Day	Total Cost
16	108	1984	5696
	112	1786	5364
	113	1770	5348
32	84	2102	6892
	91	2099	7110
	95	1884	6808
64	80	2454	10028
	84	2337	10050
	85	2183	9806
128	73	3052	15448
	80	2323	14886

Table 6.4 The results of the largest number students first implementation

Unlike the other implementations described above, in which the impact of the selected examination on the total cost is considered explicitly or implicitly, the fifth implementation does not consider the cost at all, it only randomly selects the next examination and does not consider whether or not this selection can lead to a feasible solution easily. As expected, the results and run time performance of this implementation are not good, i.e., it takes about 700 seconds to find the first better solution with around 1,000,000 failures, and there are around 200 or 2300 students

having two examinations on the same day or on consecutive days, respectively. It illustrates from the other side how important it is to have a good search strategy.

Putting the fifth implementation aside, we still have four implementations to choose from. In fact, they are somewhat related to each other, each one making a trade-off between reducing cost and easily finding a solution, and emphasis on one aspect. For example, a big examination usually has a lot of conflicts with others examinations, its domain size is likely to be reduced quickly. Thus small-domain-first rule selects the big examination implicitly, and the large-number-of-conflicts-first rule will select the examination with the smallest domain implicitly. However the statement is not always true, and it makes a difference among these four implementations. Comparing the four implementations according to the results and run time performance, the first implementation is by far the best one in our case, it will be used in the following experiments.

6.1.2 Comparison among different value selection methods

Variable ordering determines the shape of the search tree, while value ordering specifies which parts of the search tree are first explored. If the search space is not too big and we can search the space within the time limit, then no matter which value selection rule we use, the final results will be the same because we will have tried every possibility. The only difference is how soon the best solution will be found. In practice, usually the real problem is of large scale, its search space is huge, so it is impossible to search the whole

space within the time limit. Thus it is important to explore the right branches of the search tree first (the branches where the best solution is likely to be), that means value ordering will play a very important role in searching for the optimized solution.

In this section, we construct three implementations. All implementations use the same variable ordering rule to select the next examination: selecting the examination that has the largest number of conflicts with the fixed examinations. Value selection methods described in section 5.4.1.2 are used in different implementations. The first implementation adopts the first method of section 5.4.1.2, which only considers the effect of the fixed examinations and concentrates on reducing the cost of the partial solution by choosing the best period for the selected examination. The second implementation uses the second method of section 5.4.1.2, which mainly considers the effect of the unfixed examinations, and chooses the period that has less possibility to be chosen by the unfixed examinations. The third implementation orders the value list in a random way. Every implementation can solve the problem, but with totally different results, e.g., the solutions of some implementations are much better than that of others. The solutions of these implementations will be given in next part, followed by comparison and discussion.

The purpose of the section is to make a comparison among value selection rules, so it is enough to just find the first solution of different implementations.

Table 6.5 summarizes the results of the first implementation. The program runs very fast, it can find the first solution within 1 second, more accurately, around 0.3 seconds.

Cost Parameter	nb Same Day	nb Consecutive Day	Total Cost
16	114	1894	5612
	125	1776	5552
	132	1758	5628
32	96	2086	7244
	99	2056	7280
	105	1939	7238
64	94	2357	10730
	95	2269	10618
	101	2108	10680

Table 6.5 The results of the first implementation: select period according to the fixed examinations.

From the table we can see that the results are at the same level as those of École Polytechnique. The implementation and the first implementation described in section 6.1 adopt the same search strategy: the same variable and value selection rule. The difference is that in this implementation, we try to find the first feasible solution, while in the former implementation, we try to find the first better solution. Before the first better solution is found, a lot of feasible solutions are found, including the first one.

The results of the second implementation are summarized in table 6.6. The first feasible solution is found around 0.35 second. It is a little longer than the first one because it does more work to decide which period should be chosen.

Cost Parameter	nb Same Day	nb Consecutive Day	Total Cost
16	283	2195	8918
	295	2047	8814
32	236	2694	12940
	300	2310	14220
	322	2245	14794
64	247	2587	20982
	303	2345	24082
	316	2301	24826

Table 6.6 The results of the second implementation: select period according to the unfixed examinations.

As we can see, the results are not good at all, the reason being that when choosing a period for the selected examination, we do not put much effort on reducing the total cost. on the contrary we try to keep the domain of the unfixed examinations as large as possible so that it is easy to find a feasible solution. It is very similar to the small-domain first rule used for variable ordering. An available period that has less impact on the domain size of the unfixed examinations usually is not the best period for the selected examination, thus the results are not good.

The third implementation chooses a period randomly, it does not consider the total cost at all. The results are even worse than that of the second implementation, i.e., there are 577 or 4593 students who have two examinations on the same day or on consecutive days. Choosing a period randomly means that the worst period may be assigned to the selected examination, or the best period has little opportunity to be chosen. The results

prove that the value ordering rule is important indeed, if not choosing it carefully, it may lead you to a wrong subspace. Additionally the program runs very fast, it takes around 0.3 second to find the first solution with no failure. This illustrates again that in our case the domain size of each variable is big enough, and even using random selection rule, it is still easy to find a feasible solution with a few or no failure. This implies that there are a large number of feasible solutions in the search space and we do not need to worry about finding a feasible solution, instead, we should concentrate on how to find better or best solutions.

Among the three implementations, it is obvious that the first one is the best. We will use it in following experiments.

6.1.3 Reduction of search space

To find the best solution, we need to explore the whole search space, but it is impossible because the search space is huge. We need to reduce it to some subspace, namely cutting some branches from each node, so that it is easy to find the better solutions.

In this section, three implementations are constructed corresponding to the three score functions that are used to sort the periods, described in section 5.5 *Search Space Reduction*. Recalling them briefly, the first score function sorts the periods according to the fixed examinations and does not consider the effect of the unfixed examinations. The second score function sorts the periods according to the unfixed examinations, and does

not consider the cost at all. The third score function considers the impact of both the fixed and unfixed examinations, it sorts the periods according to the final cost of the first feasible solution.

We already have the first solution of these implementations: the results of the first and second implementations in section 6.1.2 and the result of the first implementation in section 6.1.1. As we can see that the third implementation is the best one, followed by the first and second implementations. We also use some kind of minimizing functions to search for the best solution within the time limit. The results still show that the third implementation is the best one.

According to the definition, the second score function does not consider the cost, thus its solutions are not good for sure. The first score function is the simplest one among these three methods, as it sorts the periods according to the cost with the fixed examinations. Its solutions are better than the second method, but because it only considers the impact of the fixed examinations, the solutions it finds will not be the best one. The third score function is expensive and the most complicated method: it sorts the periods according to the corresponding feasible solutions, that means that for each available period, we find a feasible solution first, then sort them. This way the impact of both the fixed and unfixed examinations are taken into account, thus its solutions are much better than that of the other methods.

In following parts, we will use the third score function to search for the best solution in the reduced search space.

6.1.4 Comparison among different search control algorithms

As mentioned in chapter 5, search control algorithms determine which node should be explored first, lead to how soon the better solution will be found. There is no difference among the search control algorithms if the best solution must be found. But if we can't go through the whole reduced search space within the time limit, that means we are not sure whether the solution found is the best one or not, then they will make a difference according to your problem and search strategy, i.e., some search control algorithms will give better results than others.

In this section, three implementations are constructed. They adopt the same search strategy: the first implementation in section 6.1.1 to select the next examination and assign a period to it, and the same score function: the third one in section 6.1.3 to reduce the search space. The first implementation uses the Depth First Search (DFS) control algorithm to search for optimized solution, the second one uses the Limited Discrepancy Search (LDS) control algorithm and the third implementation applies the Depth bounded Discrepancy Search (DDS).

Table 6.7, 6.8 and 6.9 summarize the results. There are five columns: the first column is the ID of a solution, the second column shows how many students have two

examinations on the same day while the third column shows the number of students who have two examinations on consecutive days, the fourth column shows the total cost of a solution, and the last column shows how much time it takes to find the solution.

Solution ID	nb Same Day	nb Consecutive Day	Total Cost	CPU time
1	86	1783	6318	40.89
2	82	1740	6104	722.45
3	83	1697	6050	22295.42

Table 6.7 The results of DFS

Solution ID	nb Same Day	nb Consecutive Day	Total Cost	CPU time
1	86	1783	6318	39.88
2	83	1814	6284	334.28
3	83	1789	6234	535.47
4	85	1701	6122	907.24
5	83	1697	6050	3811.92

Table 6.8 The results of LDS

Solution ID	nb Same Day	nb Consecutive Day	Total Cost	CPU time
1	86	1783	6318	40.23
2	82	1740	6104	708.42
3	83	1697	6050	21832.29

Table 6.9 The results of DDS

From the results, we can see that each implementation can find the best solution but with a different time. Also we notice that they find different solutions because they follow different paths to explore the search space. Among the three implementations, the second one (LDS) is the best one because it takes less time to find the best solution.

The results shown in table 6.7-6.9 are only an example (the first examination is fixed to period 26): we also try other possibilities, and the results show the same trend: the LDS search control algorithm is the best one in our case.

6.1.5 Stability of the search strategy

Besides solving the examination scheduling problem for autumn term 1999, we also use the application to solve the problem for winter term 2000 and autumn term 2000. The application can find better solutions within the time limit and the results are better than that of École Polytechnique.

6.2 The Results Of The Examination-Room Timetabling Problem

The examination-session problem is only one part of the examination scheduling problem, the examination-room timetabling problem is also an important part of the problem. As we know that usually an examination (course) has one or several sections, thus there are two different views of the examination. From the high level point of view, the examination is treated as the basic unit, while from the low level point of view, the examination is composed of several sections, and the section is the basic unit. Thus there exist two possible ways to schedule rooms to an examination. One way is: ignore the sections, regard the examination as a whole, and try to assign room/rooms to it. The

other way is: try to assign room/rooms to sections one by one, finally fixing the examination, that is: rooms are scheduled for it since each section of it is fixed.

In the following sections, we will investigate these two approaches.

6.2.1 Comparison among different value selection methods

The first approach is adopted to assign room/rooms to the selected examination that is the examination is considered as the basic unit. In this approach, the variable selection is very simple: always select the biggest examination from the unfixed examination list, so that the large examination will take fewer rooms, which in turn results to improve the room utilization.

Three implementations are constructed, all of them use the same variable selection rule as described above, and depending on the value selection rule, the selected examination can be fixed completely at once and removed from the examination list or it can be fixed partially and put back into the examination list in the right place according to its remaining size. Value selection methods described in section 5.4.2.2 are used in different implementations. The largest-first value selection rule, which always selects the largest available room/rooms for the selected examination, is used in the first implementation. The first-fit method, which selects the first suitable room for the selected examination, is adopted in the second implementation and the best-fit method, which selects the best set of rooms for the examination, is applied in the third

implementation. Each implementation can solve the problem, but the results are different, e.g., the solutions of some implementations use fewer rooms, while other solutions have a high rate of room utilization. The solutions of these implementations will be given in the next part, followed by comparison and discussion.

Table 6.10, 6.11, 6.12 summarize the results of the first, second and third implementation, respectively. There are six columns in each table: the first column is the ID of sessions, the second column shows the number of students in a session, the third column shows how many rooms are used by the session, the fourth column shows room usage for the session. Each room contains a certain number of students: the room that is assigned to one or several examinations and contains the least number of students in the session is shown in the fifth column (the number of students versus the room size). Each exam is scheduled into one or several rooms: the examination that is scheduled into more than one room and has the least number of students in one of the rooms (comparing with other examinations in the session and the size difference between the room and the examination in the room is at least five) is shown in the last column (the number of students in the room versus the size of the examination). As described in Chapter 3, the total capacity of each session is the same: 1611, and the total number of available rooms of each session is 31.

SessionID	nb Students	nb Rooms	Room Utility	Smallest Stud	Balanced Stud
1	1320	30	0.834	4 / 32	1 / 43
2	339	4	0.819	50 / 102	5 / 105
3	602	12	0.693	15 / 48	73 / 202
4	220	4	0.531	44 / 83	
5	1043	20	0.849	9 / 39	5 / 722
6	203	3	0.613	37 / 100	37 / 166
7	539	10	0.699	1 / 54	12 / 343
8	136	2	0.589	47 / 102	
9	1399	27	0.943	9 / 36	4 / 873
10	246	5	0.515	7 / 64	46 / 175
11	923	20	0.752	11 / 42	1 / 49
12	86	2	0.372	30 / 102	
13	754	11	0.918	27 / 50	48 / 646
14	263	3	0.795	32 / 100	15 / 246
15	623	13	0.646	1 / 48	4 / 418
16	170	2	0.736	77 / 102	
17	929	19	0.781	2 / 42	6 / 604
18	136	2	0.589	60 / 102	
19	410	7	0.686	7 / 60	40 / 271
20	404	5	0.845	41 / 83	41 / 141
21	1142	22	0.876	8 / 37	15 / 60
22	162	3	0.489	42 / 100	
23	662	10	0.859	22 / 60	10 / 93
24	283	4	0.684	46 / 83	51 / 180
25	851	18	0.742	11 / 45	39 / 122
26	245	4	0.592	18 / 83	1 / 61
27	831	14	0.861	13 / 48	
28	347	5	0.726	20 / 64	22 / 151

Table 6.10 The results of the first implementation: the largest-first algorithm

SessionID	nb Students	nb Rooms	Room Utility	Smallest Stud	Balanced Stud
1	1320	23	0.985	31 / 39	5 / 53
2	339	4	0.819	50 / 83	3 / 105
3	602	8	0.915	33 / 60	69 / 171
4	220	2	0.952	101 / 102	
5	1043	17	0.946	9 / 45	5 /722
6	203	2	0.879	74 / 102	37 / 166
7	539	7	0.901	12 / 60	12 / 343
8	136	2	0.589	47 / 102	
9	1399	25	0.991	27 / 36	4 / 873
10	246	3	0.743	21 / 100	46 / 175
11	923	14	0.956	15 / 48	1 / 61
12	86	1	0.667	86 / 129	
13	754	11	0.918	27 / 50	48 / 646
14	263	3	0.795	32 / 100	15 / 246
15	623	9	0.869	30 / 59	4 / 418
16	170	2	0.736	77 / 102	
17	929	14	0.963	12 / 48	3 / 53
18	136	2	0.589	60 / 102	
19	410	5	0.858	7 / 64	40 / 271
20	404	5	0.845	39 / 64	39 / 141
21	1142	19	0.960	22 / 42	8 / 231
22	162	2	0.701	42 / 102	
23	662	9	0.923	42 / 59	6 / 70
24	283	3	0.855	57 / 102	51 / 180
25	851	13	0.928	24 / 48	6 / 301
26	245	3	0.740	60 / 100	
27	831	12	0.956	43 / 48	34 / 365
28	347	4	0.838	42 / 83	22 / 151

Table 6.11 The results of the second implementation: the first-fit algorithm

SessionID	nb Students	nb Rooms	Room Utility	Smallest Stud	Balanced Stud
1	1320	26	0.960	12 / 45	
2	339	5	0.985	41 / 42	
3	602	10	0.941	16 / 32	73 / 202
4	220	4	0.965	44 / 45	
5	1043	19	0.963	15 / 36	
6	203	3	1.000	37 / 37	
7	539	9	0.969	24 / 32	
8	136	2	0.925	47 / 47	
9	1399	27	0.989	26 / 36	8 / 49
10	246	5	0.918	14 / 32	
11	923	17	0.970	15 / 36	94 / 325
12	86	2	0.945	30 / 32	
13	754	11	0.991	27 / 28	
14	263	4	0.956	17 / 28	
15	623	10	0.986	27 / 28	
16	170	2	0.929	77 / 83	
17	929	17	0.989	27 / 28	
18	136	2	0.951	60 / 60	
19	410	7	0.962	16 / 32	
20	404	5	0.978	41 / 42	76 / 205
21	1142	21	0.970	22 / 36	
22	162	3	0.982	42 / 42	
23	662	9	0.947	12 / 28	6 / 70
24	283	4	0.979	46 / 47	
25	851	16	0.938	19 / 36	72 / 301
26	245	4	0.904	18 / 28	
27	831	13	0.971	27 / 28	
28	347	6	0.935	20 / 32	

Table 6.12 The results of the third implementation: the best-fit algorithm

From the tables, we can see that in terms of room utilization the best-fit algorithm (in most of the cases) is the best one, followed by the first-fit and the largest-first algorithms. The average rate of room utilization is 96.6%, 90.4% and 76.6%,

respectively. In terms of using few rooms, the first-fit algorithms is the best one, it always uses few rooms compared with the other two algorithms. In most of the cases the best-fit algorithm uses fewer rooms than the largest-first algorithm does, except when there are few students in the session, i.e., less than 400 students, the largest-first may uses few rooms. The reason is that when few students are scheduled into a largest room, the room may still be the largest one, so that it can be used for the next examination. While in the best-fit algorithm, usually the next examination is scheduled into another room. In terms of uniform distribution of students, the best-fit algorithm offers the best solution, i.e., there are only two examinations whose small part of students (6 and 8 respectively) is scheduled into different rooms. While the other two algorithms give similar results, they are not good. i.e., more than 10 examinations have small number of students in other rooms. The reason is that the two algorithms only consider the room size and always select the largest rooms for the examination, i.e., suppose the size of the examination is 101, and the largest room's capacity is 100, then 100 students will be scheduled into this room, the remaining one student will be scheduled into the second largest room (the largest-first algorithm) or the first suitable room in the room list (the first-fit algorithm). While in the best-fit algorithm, it will find the best two rooms for the examination.

As we can see the results of the largest-first algorithm are the worst one among the three of them: the basic reason is that it always uses the largest room first even if the size of

the examination is very small, which results in the low rate of room utilization and using more rooms.

The first-fit algorithm uses few rooms, the reason being that it only sorts the room list once (just before scheduling any examination into rooms) and uses the first suitable rooms for the examination, thus it will use large rooms before small rooms, which means it needs few rooms to hold the examinations of the session. As described above, on average the rate of room utilization of the first-fit algorithm is much better than that of the largest-first algorithm and is not as good as that of the best-fit algorithm. But there are two exceptions: on one hand, when the session has a lot of students, say more than 1300 students in our case (big session), then the room utilization of the first-fit algorithm may be better than that of the best-fit algorithm, i.e., for the session 1 and 9, there are 1320 and 1399 students respectively, the rate of room utilization is 98.5% / 96.0% (the first-fit/best-fit algorithm) and 99.1% / 98.9% respectively. The reason is that the best-fit algorithm has a tendency to use small rooms first because it will find the best set of rooms for the selected examination. When the session is a big one, it will run out of small rooms, and has to use a large room at some point even if there are only few students. This will decrease the room utilization. While for the first-fit algorithm, it will use large rooms before small rooms. When there are a large number of students in a session, it will use up the large rooms, and begin to use small rooms, the room utilization will not decrease as much as that of the best-fit algorithm. On the other hand, when there are a small number of students in a session, then the rate of room utilization of the first-

fit approach could be as worse as that of the largest-first approach, i.e., in session 2 and 8, there are 339 and 136, and the rate of room utilization is 81.9% and 58.9% respectively. The reason is that both approaches try to use large rooms first, the only difference is that the first-fit approach sorts the examination list once while the largest-first approach sorts the examination list whenever a room is assigned to an examination. Thus if the session is a small one, it is possible that both approaches use the same set of rooms for the examinations. For example, suppose there are only two examinations in a session, they have 80 and 60 students respectively. In our case, both approaches will use the first and the second largest rooms. In the largest-first case, after 80 students are scheduled into the largest room, the second largest room becomes the largest one, the second examination is scheduled into it. While in the first-fit approach, the second examination is still scheduled into the second largest room because the largest room can't hold the two examinations and the second largest room is the first room that can hold the second examination.

From the results we can see that in term of room utilization the best-fit approach works well most of the time. Since it has a tendency to use small rooms, it will use more rooms compared with the first-fit approach. Although in terms of the uniform distribution of students the best-fit approach is the best, it can still be improved by introducing a variable 'balanceFactor' and increasing its value (described in Chapter 5), i.e., we can obtain the result in which no examinations whose small part of students (less than 12) are scheduled into different rooms. But there exists a balance between the room

utilization and the uniform distribution of students, i.e., the average rate of room utilization decreases to 95.6%. In fact if the uniform distribution of students is more important than the room utilization, there exists another way to schedule rooms to examinations. We will describe it in the next section.

6.2.2 Room scheduling based on section

As described above, from the low level point of view, the examination is no longer the basic unit, the room scheduling problem can be solved according to the size of sections. This method is used in such a situation that the uniform distribution of students should be satisfied.

The basic step of the method is, first select an examination from the list, then try to assign a section per room, if it isn't possible, try to assign half of the section to a room and the other half in another room, and so on. When each section of the examination is fixed, the examination is fixed finally.

There are two possible rules for selecting an examination. The first one is, select the largest examination first just as the one used in the above section. The second one is, the next selected examination is always the one whose biggest section's size is the largest. The point is that room scheduling is based on section, thus the examination that has the largest section should be scheduled first so that it will use fewer rooms.

Two implementations will be constructed based on the two variable selection rules in this section. According to the above section, the best-fit approach is the best one in term of room utilization in most situations. It will be used as the value selection rule in the following implementations.

SessionID	nb Students	nb Rooms	Room Utility
1	1277	26	0.919
2	321	5	0.819
3	856	18	0.818
4	116	2	0.885
5	1336	28	0.906
6	88	1	0.880
7	389	6	0.888
8	128	2	0.970
9	977	20	0.862
10	336	4	0.955
11	789	15	0.855
12	200	3	0.939
13	579	11	0.822
14	1060	21	0.858
15	495	8	0.811
16	263	3	0.795
17	1022	19	0.928
18	154	2	0.969
19	547	9	0.809
20	336	5	0.939
21	854	16	0.834
22	141	3	0.904
23	1518	31	0.942
24	220	4	0.940
25	683	11	0.924
26	241	3	0.938
27	1087	18	0.979
28	178	3	0.844

Table 6.13 The results of the first implementation

SessionID	nb Students	nb Rooms	Room Utility
1	1277	24	0.942
2	321	5	0.819
3	856	16	0.882
4	116	2	0.885
5	1336	29	0.869
6	88	1	0.880
7	389	6	0.888
8	128	2	0.970
9	977	19	0.862
10	336	4	0.955
11	789	16	0.811
12	200	3	0.939
13	579	11	0.826
14	1060	21	0.849
15	495	8	0.811
16	263	3	0.795
17	1022	19	0.879
18	154	2	0.969
19	547	9	0.809
20	336	5	0.939
21	854	17	0.798
22	141	3	0.904
23	1518	31	0.942
24	220	4	0.940
25	683	10	0.970
26	241	3	0.938
27	1087	19	0.943
28	178	3	0.844

Table 6.14 The results of the second implementation

Table 6.13 and Table 6.14 show the results of the two implementations. Each table has four columns that are the same as that of Table 6.10.

From the tables we can see that in terms of the room utilization, they are very close, the first implementation is a little bit better, i.e., the average rate of room utilization is 88.8% and 88.1%, respectively. But if we count how many sections are separated into different rooms, the two implementations give different answers. For the first implementation, there are 55 sections, each of which is scheduled into two rooms, and 3 sections, each of which is scheduled into three rooms. While for the second implementation, there are 49 sections, each of which is scheduled into two rooms, and 1 section that is scheduled into three rooms. It is clear that the second implementation is better than the first one. The reason is that for the first implementation, it always selects the largest examination from the remaining list, and the section's size of the selected one is not necessary the largest, which means some small sections will be scheduled first and cause the large sections to be separated into more rooms because the size of the remaining rooms is not big enough to hold it. For example, there are three examinations, each of them only has one section whose size is 104, 121 and 123, respectively. They are all large sections, but they are not the largest examination in their sessions. Thus in the first implementation they are selected relatively late, the results are, each of them is scheduled into three rooms. While in the second implementation, they are selected first, and each of them is scheduled into one room.

6.3 SUMMARY

The examination scheduling problem is solved through solving its two subproblems: examination-session problem and examination-room problem. To solve the first

subproblem we try several variable and value selection rules. Each of them can solve the problem, but the results are different, some of them can find a better solution quickly while the others need a long time. Depending on the request, the second subproblem can be solved based on the size of either examination or section, and no matter which one is adopted, the best-fit value selection rule offers better solution in most cases.

CHAPTER 7

CONCLUSION

The examination scheduling problem is very difficult to solve and is classified as NP-hard. The difficulties arise from its large scale, the large number of contradictory requirements, constraints and quality criteria.

This thesis has reviewed the methods for solving the problem, and adopted constraint programming to generate examination timetables. We have investigated several algorithms, and tried to look for the best one in our case.

7.1 Contributions

We have divided the problem into two subproblems, and solved them in two phases: examination-session timetabling and examination-room timetabling. For each sub-problem we developed several search strategies, including static and dynamic rules for variable and value selection. Combining variable and value ordering rules together, different algorithms were constructed.

In order to improve the run-time performance, we have used different techniques to reduce the search space so that only those sub-spaces in which the better solutions are likely to be are explored during the search. Additionally, we have investigated different

search control algorithms to improve the performance and see where the algorithms are likely to make a wrong choice: at the top of the search tree, in the middle of the search tree or at the bottom of the search tree.

All hard constraints were satisfied in each algorithm. Soft constraints were classified according to priorities; the soft constraints with high priority were satisfied before the ones with low priority.

Though several of our algorithms could find better solutions within the time limit, one of them had a better run-time performance. Summarizing it here, the algorithm had several rules: large number of conflicts first was used as the rule for variable selection; the value that led to the best partial solution was selected first; for the rule of the search space reduction, we sorted the available periods according to the cost of their feasible solutions, then cut some of them; limited discrepancy search was used as the search control algorithm. The algorithm is very stable, we tested it by using three sets of data from École Polytechnique de Montréal. The results were very good.

Each implementation can allocate rooms for each examination automatically. Several value selection rules and two variable selection rules were investigated. The examination-session timetabling problem can be solved based on the size of either examination or section, and no matter which one is adopted, the best-fit value selection rule offers the better solution in most cases.

The constraints, such as precedence constraints, which states that some examinations may be required to take place before or after some other examinations, etc., can be easily added to the application at run time by reading them from a text file.

The results of our study prove that constraint programming is very powerful in dealing with examination scheduling problems. ILOG SOLVER was used in the project. It provided some basic functions and facilities, and it was easy to model the relationships and constraints of the problem.

7.2 Discussion and Future Work

Variable ordering plays an important role in searching for a better solution. In this project several rules were investigated, all of them only considered the impact of the fixed examinations. The ideal way to select the next examination is to consider the impact of both the fixed and unfixed examinations.

Our method for sorting the list of available periods is expensive, if a simple and an efficient method can be found, then we can reduce the search space quickly and thus improve the run-time performance.

The room allocation strategies used in this project is relatively simple, we only try to find the first solution. It is possible to define a cost function associated with each solution, and find the optimal solution.

Besides the three aspects described above, we can also add some visualization options to the application, i.e., we can provide a GUI to let users add or delete constraints, select the max number of discrepancy and the time limit, etc.

REFERENCES

[1] BARDADYM, V. A. (1996). "Computer-Aided School and University Timetabling: The New Wave", *Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Edinburgh, U.K. August/September 1995. Springer-Verlag Lecture Notes in Computer Science.* **1153**, pp. 22-45.

[2] CARTER M. W. (1986). "A survey of practical applications of examination timetabling algorithms", *European Journal of Operational Research*, **34**, no. 2 pp. 193-202.

[3] de WERRA D. (1985). "An introduction to timetabling", *European Journal of Operational Research*, **19**, pp. 151-162.

[4] WHITE G. M. and KANG L. (1992). "A logic approach to the resolution of constraints in timetabling", *European Journal of Operational Research*, **61**, no. 3, pp. 306-317.

[5] YOSHIKAWA M., KANEKO K., YOMURA Y. and WATANABE M. (1994). "A constraint-based approach to high-school timetabling problems: A case study", *Proceedings of theTwelfth National Conference on Artificial Intellegence (AAAI'94), Seattle, WA, July 31 – August 4, 1994.*

[6] GRIMMETT G. R. and McDIARMID C. J. H. (1975). "On colouring random graphs", *Mathematical Proceedings of Cambridge Philosophical Society*, **77**, pp. 313-324.

[7] JOHNSON D. (1990). "Timetabling university examinations", *Journal of the Operational Research Society*, **41**, no. 1, pp. 39-47.

[8] BARHAM A. M. and WESTWOOD J. B. (1978). "A simple heuristic to facilitate course timetabling", *Journal of the Operational Research Society*, **29**, no. 11, pp. 1055-1060.

[9] METHA N. K. (1981). "The application of a graph colouring method to an examination scheduling problem", *Interfaces*, **11**, no. 5, pp. 57-46.

[10] CARTER M. W. and LAPORTE G. (1996). "Recent developments in practical examination timetabling", *Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Edinburgh, U.K. August/September 1995.* Springer-Verlag *Lecture Notes in Computer Science.* **1153**, pp. 1-21.

[11] CARTER M. W., LAPORTE G. and CHINNECK J. W. (1994). "A general examination scheduling system", *Interfaces*, **24**, no. 3, pp. 109-120.

[12] CARTER M. W., LAPORTE G. and LEE S. Y. (1996). "Examination timetabling: algorithmic strategies and applications", *Journal of the Operational Research Society*, **47**, no. 3, pp. 373-383.

[13] WHITE G. M. and CHAN P. W. (1979). "Towards the construction of optimal examination timetables", *INFOR*, **17**, no. 3, pp. 219-229.

[14] LOTFI V. and CERVENY R. (1991). "A final-exam-scheduling package", *Journal of the Operational Research Society*, **42**, no. 3, pp. 205-216.

[15] FISHER J. G. and SHIER D. R. (1983). "A heuristic procedure for large-scale examination scheduling problems", *Tech. Report 417, Dept. of Mathematical Sciences, Clemson University.*

[16] ARANI T. and LOTFI V. (1989). "A three phase approach to final exam scheduling", *IIE Transactions*, **21**, no. 1, pp. 86-96.

[17] BALAKRISHNAN N., LUCENA A. and WONG R. T. (1992). "Scheduling examinations to reduce second-order conflicts", *Computers & Operations Research*, **19**, no. 5, pp. 353-361.

[18] CHAHAL N. and de WERRA D. (1989). "An interactive system for constructing timetabling on a PC", *European Journal of Operational Research*, **40**, no. 1, pp. 32-37.

[19] GOSSELIN K. and TRUCHON M. (1986). "Allocation of classrooms by linear programming", *Journal of the Operational Research Society*, **37**, no. 6, pp. 561-569.

[20] TRIPATHY A. (1980). "A lagrangean relaxation approach to course timetabling", *Journal of the Operational Research Society*, **31**, no. 7, pp. 599-603.

[21] HOLLAND J. H. (1975). "Adaptation in natural and artificial systems", *The University of Michigan Press, Ann Arbor, Michigan.*

[22] CORNE D., FANG H. L. and MELLISH C. (1993). "Solving the modular exam scheduling problem with genetic algorithms", *Proceedings of the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, P.W.H. Chung, G. Lovegrove & M. Ali (eds.), Edinburgh, Scotland, pp. 370-373.

[23] BURKE E., ELLIMAN D. and WEARE R. (1994). "A genetic algorithm for university timetabling", *AISB Workshop on Evolutionary Computing, University of Leeds, UK.*

[24] ERGÜL A. (1996). "GA-Based examination scheduling experience at Middle East Technical University", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp.212-216

[25] KIRKPATRICK S., GELATT C. D. and VECCHI M. P. (1983). "Optimization by simulated annealing", *Science*, **220**, no. 4598, pp. 671-680.

[26] THOMPSON J. M. and DOWSLAND K. A. (1995). "TISSUE wipes away exam time tears: a computerized system helps Swansea University improve examination timetabling ", *O.R. Insight*, **8**, no. 4, pp. 28-32.

[27] THOMPSON J. M. and DOWSLAND K. A. (1996). "General cooling schedules for a simulated annealing based timetabling system", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp. 345-363.

[28] ABRAMSON D. (1991). "Constructing school timetables using simulated annealing: sequential and parallel algorithms", *Management Science*, **37**, no. 1, pp. 98-113.

[29] DAVIS L. and RITTER L. (1987). "Schedule optimization with probabilistic search", *Proceedings of the 3rd IEEE Conference on Artificial Intelligence Applications, Orlando, Florida, USA*, pp. 231-236.

[30] GLOVER F. (1989). "Tabu search – Part 1", *ORSA Journal on Computing*, **1**, pp. 190-206.

[31] GLOVER F. (1990). "Tabu search – Part 2", *ORSA Journal on Computing*, **2**, pp. 4-32.

[32] HERTZ A. (1991). "Tabu search for large scale timetabling problems", *European Journal of Operational Research*, **54**, no. 1, pp. 39-47.

[33] BOUFFLET J. P. and NEGRE S. (1996). "Three methods used to solve an examination timetabling problem", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp. 327-344.

[34] COSTA D. (1994). "A tuba search algorithm for computing an operational timetable", *European Journal of Operational Research,* **76**, no. 1, pp. 98-110.

[35] SCHAERF A. (1996). "Tabu search techniques for large high-school timetabling problems", *Comp. Science Dept. of Interactive Systems, CS-R9611, Centrum voor Wiskunde en Informatica, SMC, Netherlands Organization for Scientific Research.*

[36] Van HENTENRYCK P. (1989). "Constraint Satisfaction in Logic Programming", *MIT Press, Cambridge, MA.*

[37] DINCBAS M. and Van HENTENRYCK P. (1990). "Solving large combinatorial problems in logic programming", *Journal of Logic Programming,* **8.** no. 1&2 pp. 75-93.

[38] NUIJTEN W. P. M., KUNNEN G. M., AARTS E. H. L. and DIGNUM F. P. M. (1994). "Examination timetabling: a case study for constraint satisfaction", *Proceedings of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications (M. J. Wooldridge & N. R. Jennings eds.),* Amsterrdam, Pays-Bas, pp. 11-19.

[39] DAVID P. (1997). "A constraint-based approach for examination timetabling using local repair techniques", *Proceedings of the 2nd International Conference on the Practice and Theory of Automated Timetabling (E. Burke & M. Carter eds.), Toronto, Canada,* pp. 132-145.

[40] JAFFAR J. and MAHER M. J. (1994). "Constraint Logic Programming: a survey", *Journal of Logic Programming,* **19/20**, pp. 503-581.

[41] BOIZUMAULT P., DELON Y. and PERIDY L. (1995). "Constraint logic programming for examination timetabling", *Journal of Logic Programming,* **26**. no. 2, pp. 217-233.

[42] LEONG Y. L. (1995). "Right on schedule: exam timetabling at the National University of Singapore gets and added boost with new tools", *ComputerWorld,* May 12-18 1995, pp. 22-23.

[43] FAHRION R. and DOLLANSKY G. (1992). "Construction of university faculty timetables using logic programming", *Discrete Applied Mathematics*, **35**, no. 2, pp. 221-236.

[44] CHENG C., KANG L., LEUNG N. and WHITE G. M. (1995). "Investigations of a constraint logic programming approach to university timetabling", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp. 112-129.

[45] GUERET C., JUSSIEN N., BOIZUMAULT P. and PRINS C. (1995). "Building university timetables using constraint logic programming", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp. 130-145.

[46] LAJOS A. (1995). "Complete university modular timetabling using constraint logic programming", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp. 146-161.

[47] HENZ M. and WURTZ J. (1995). "Using Oz for college timetabling", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153**, pp. 162-177.

[48] GOLTZ H. J. and MATZKE D. (1999). "Constraint programming – university timetabling using constraint logic programming", *Lecture Notes in Computer Science.* **1551**, pp. 320-334.

[49] BURKE E. K., NEWALL J. P. and WEARE R. F. (1996) "A memetic algorithm for university exam timetabling", *The Practice and Theory of Automated Timetabling, First International Conference (E.K. Burke and P. Ross eds.), Springer-Verlag, Lecture Notes in Computer Science.* **1153** pp. 241-250

[50] TSANG E. (1993). "Foundation of Constraint Satisfaction", *Academic Press.*

[51] BARTAK R., "Online tutorial: Constraint programming", *http://kti.ms.mff.cuni.cz/~bartak/constraints/*

[52] MARRIOTT K. and STUCKEY P. J. (1998). "Programming with constraints: an introduction", *Cambridge, Mass.: MIT Press.*

[53] PESANT G. (2000). "Course notes for INF6101: Programmation par constraintes" *http://www.cours.polymtl.ca/inf6700/*

[54] MEIER M. (1995) "Debugging constraint programs", *Principles and Practice of Constraint Programming, First International Conference, CP'95 (U. Montanari & F. Rossi eds.), Cassis, France, Lecture Notes in Computer Science*, **976**, pp. 204

[55] ILOG SOLVER User Manual and reference Manual Version 4.4

[56] HARALICK R. M. and ELLIOTT G. L. (1980). "Increasing tree search efficiency for constraint satisfaction problem", *Artificial Intelligence* **14**, pp. 263-313.

APPENDIX

AN EXAMPLE OF THE SOLUTION

This is the 5th solution, it is found within 5153.100000 second
The total cost is: 6050
There are 83 students who have two exams on the same day
There are 1697 students who have two exams on connective days
There are 0 students who have two exams in the same time
Max number of discrepancies = 10
The smallest number of students in a room is 15 and it is in session: 7
The average occupancy rate is: 0.898

The session 0:

examID	Size	sectID	room
1.427	19	1 :	A-532(19/129);
1.562	1	1 :	B-512(1/100);
2.531	11	1 :	B-618(11/ 64);
2.574	20	1 :	A-616(20/ 60);
3.412	49	1 :	B-600.5(49/ 50);
4.320	59	1 :	B-600.6(59/ 59);
5.270	12	1 :	B-618(12/ 64);
6.555	30	1 :	B-429(30/ 32);
7.433	6	1 :	A-416_1(6/ 48);
7.440	15	1 :	B-418(15/102);
8.405	28	1 :	A-522(28/ 28);
9.350	56	1 :	B-620.2(56/ 60);
AE434	34	1 :	B-505(34/ 35);
AE471	7	1 :	B-512(7/100);
ELE3300	46	1 :	B-512(46/100);
ELE3300	46	2 :	B-512(46/100);
EN050	1	1 :	A-532(1/129);
IF215	60	1 :	B-600.3(60/ 60);
IF570	31	1 :	A-604(31/ 32);
ING1015	41	1 :	B-418(41/102);
ING1015	41	2 :	B-418(41/102);
ING1015	41	3 :	B-415(41/ 83);
ING1015	41	4 :	B-415(41/ 83);
ING1015	41	5 :	B-618(41/ 64);
ING1015	40	6 :	A-616(40/ 60);
ING1015	40	7 :	A-416_1(40/ 48);
ING1015	40	8 :	B-314(40/ 48);
ING1015	40	9 :	B-405(40/ 42);
MEC3310	55	1 :	A-532(55/129);
MEC3310	54	2 :	A-532(54/129);
PHS0101	53	1 :	B-315(53/ 54);

```
  TS410    6   1 :     B-314(  6/ 48);

The session 1:
  examID  Size sectID     room
   3.302   60   1 :   B-600.3( 60/ 60);
   3.415   48   1 :     B-512( 48/100);
   3.415   47   2 :     B-512( 47/100);
 ING1010   42   1 :     A-532( 42/129);
 ING1010   42   2 :     A-532( 42/129);
 ING1010   42   3 :     A-532( 42/129);
 ING1010   42   4 :     B-418( 42/102);
 ING1010   42   5 :     B-418( 42/102);
 ING1010   42   6 :     B-415( 42/ 83);
 ING1010   42   7 :     B-618( 42/ 64);
 ING1010   42   8 :     A-616( 42/ 60);
 ING1010   42   9 :   B-620.2( 42/ 60);
 ING1010   42  10 :   B-600.6( 42/ 59);
 ING1010   42  11 :     B-315( 42/ 54);
 ING1010   42  12 :   B-600.5( 42/ 50);
 ING1010   42  13 :   A-416_1( 42/ 48);
 ING1010   42  14 :     B-314( 42/ 48);
 ING1010   42  15 :     B-637( 42/ 47);
 ING1010   42  16 :   B-600.4( 42/ 45);
 ING1010   41  17 :     A-622( 41/ 45);
 ING1010   41  18 :     B-415( 41/ 83);

The session 2:
   examID Size sectID     room
   2.515   27   1 :     A-522( 27/ 28);
   3.507    8   1 :     B-512(  8/100);
   4.220   43   1 : B-316.1_1( 43/ 45);
   5.478   12   1 :     B-512( 12/100);
   8.231   41   1 :     B-418( 41/102);
   8.231   41   2 :     B-418( 41/102);
   8.231   40   3 :     B-512( 40/100);
   8.231   40   4 :     B-512( 40/100);
   8.231   40   5 :     B-405( 40/ 42);
  GLQ2201  19   1 :     B-418( 19/102);
    IN505  23   1 :     A-532( 23/129);
  IND2101  46   1 :     B-637( 46/ 47);
  MEC2400  51   1 :     A-532( 51/129);
  MEC2400  50   2 :     A-532( 50/129);
  MEC2400  50   3 :   B-600.5( 50/ 50);

The session 3:
   examID Size sectID     room
   1.320   57   1 :   B-600.6( 57/ 59);
```

```
   IF335   62   1 :     B-618( 62/ 64);

The session 4:
   examID Size sectID      room
   3.251    3   1 :     B-315(  3/ 54);
     340   77   1 :     B-415( 77/ 83);
   5.217   11   1 :     B-418( 11/102);
   6.502   27   1 :     A-522( 27/ 28);
   7.534   14   1 :     B-512( 14/100);
   9.370   56   1 :   B-600.6( 56/ 59);
 CHE0501   51   1 :     B-315( 51/ 54);
 ING1030   41   1 :     A-532( 41/129);
 ING1030   41   2 :     A-532( 41/129);
 ING1030   41   3 :     A-532( 41/129);
 ING1030   41   4 :     B-418( 41/102);
 ING1030   41   5 :     B-418( 41/102);
 ING1030   41   6 :     B-512( 41/100);
 ING1030   40   7 :     B-512( 40/100);
 ING1030   40   8 :     B-618( 40/ 64);
 ING1030   40   9 :   B-600.3( 40/ 60);
 ING1030   40  10 :     A-616( 40/ 60);
 ING1030   40  11 :   B-620.2( 40/ 60);
 ING1030   40  12 :   B-600.5( 40/ 50);
 ING1030   40  13 :   A-416_1( 40/ 48);
 ING1030   40  14 :     B-314( 40/ 48);
 ING1030   40  15 :     B-637( 40/ 47);
 ING1030   40  16 :     B-405( 40/ 42);

The session 5:
   examID Size sectID      room
   9.583   41   1 :     A-532( 41/129);
   9.583   41   2 :     A-532( 41/129);
   9.583   41   3 :     A-532( 41/129);
   9.583   41   4 :     B-418( 41/102);
   9.583   41   5 :     B-418( 41/102);
   9.583   41   6 :     B-405( 41/ 42);

The session 6:
   examID Size sectID      room
   1.564   15   1 :     B-512( 15/100);
   2.533   32   1 :     B-505( 32/ 35);
     215   42   1 :     B-418( 42/102);
     215   42   2 :     B-418( 42/102);
     215   42   3 :     B-512( 42/100);
     215   42   4 :     B-512( 42/100);
     215   42   5 :     B-415( 42/ 83);
     215   42   6 :     B-618( 42/ 64);
```

```
    215   42   7 :   B-600.3( 42/ 60);
    215   42   8 :     A-616( 42/ 60);
    215   41   9 :   B-620.2( 41/ 60);
    215   41  10 :     B-415( 41/ 83);
  3.539   27   1 :     A-522( 27/ 28);
  4.504   32   1 :     A-604( 32/ 32);
  9.473   26   1 :     B-304( 26/ 36);
  AE464   32   1 :     B-429( 32/ 32);
  E-302    2   1 :     B-505(  2/ 35);
  IF314   53   1 :     A-532( 53/129);
  IF314   52   2 :     A-532( 52/129);

The session 7:
   examID Size sectID      room
   2.561   15   1 :     A-522( 15/ 28);
   3.413   57   1 :   B-600.6( 57/ 59);

The session 8:
   examID Size sectID      room
   1.540   14   1 :     B-315( 14/ 51);
   3.430   59   1 :   B-600.6( 59/ 59);
   3.528   64   1 :     B-618( 64/ 64);
   5.476   17   1 :     B-512( 17/100);
   7.510    1   1 :     B-405(  1/ 42);
 IND2802   30   1 :     A-604( 30/ 32);
 ING1035   41   1 :     A-532( 41/129);
 ING1035   41   2 :     A-532( 41/129);
 ING1035   40   3 :     A-532( 40/129);
 ING1035   40   4 :     B-418( 40/102);
 ING1035   40   5 :     B-418( 40/102);
 ING1035   40   6 :     B-512( 40/100);
 ING1035   40   7 :     B-512( 40/100);
 ING1035   40   8 :   B-600.3( 40/ 60);
 ING1035   40   9 :     A-616( 40/ 60);
 ING1035   40  10 :   B-620.2( 40/ 60);
 ING1035   40  11 :     B-315( 40/ 54);
 ING1035   40  12 :   B-600.5( 40/ 50);
 ING1035   40  13 :   A-416_1( 40/ 48);
 ING1035   40  14 :     B-314( 40/ 48);
 ING1035   40  15 :     B-637( 40/ 47);
 ING1035   40  16 :   B-600.4( 40/ 45);
 ING1035   40  17 :     A-622( 40/ 45);
 ING1035   40  18 :     B-405( 40/ 42);
   IT500   37   1 :     B-543( 37/ 37);
 MEC3320   71   1 :     B-415( 71/ 83);

The session 9:
```

```
   examID Size sectID      room
   IF430   76    1 :     B-415( 76  83);
   SH469   23    1 :     A-522( 23  28);
   SH515   60    1 :   B-600.3( 60  60);

The session 10:
   examID Size sectID      room
   1.420   24    1 :     A-604( 24  32);
   2.313   57    1 :   B-620.2( 57  60);
   3.516   59    1 :   B-600.6( 59  59);
   5.210   35    1 :     B-505( 35  35);
   6.291   25    1 :     A-522( 25  28);
   7.415    4    1 :     A-604(  4  32);
 ING1003   43    1 :     A-532( 43 129);
 ING1003   43    2 :     A-532( 43 129);
 ING1003   43    3 :     A-532( 43 129);
 ING1003   43    4 :     B-418( 43 102);
 ING1003   43    5 :     B-418( 43 102);
 ING1003   43    6 :     B-512( 43 100);
 ING1003   43    7 :     B-512( 43 100);
 ING1003   42    8 :     B-405( 42  42);
 MTH0102   51    1 :     B-315( 51  54);
   SH485   16    1 :     B-418( 16 102);

The session 11:
   examID Size sectID      room
   3.165   45    1 :     B-512( 45 100);
   3.165   44    2 :     B-512( 44 100);
   SH415   47    1 :     B-637( 47  17);

The session 12:
   examID Size sectID      room
   1.512    3    1 :     B-415(  3  83);
   2.541   31    1 :     A-604( 31  32);
   2.579   21    1 :     A-522( 21  28);
   3.522   17    1 :     B-512( 17 100);
   3.583   73    1 :     B-415( 73  83);
     424   62    1 :     B-618( 62  64);
     426   44    1 :     A-532( 44 129);
     426   44    2 :     A-532( 44 129);
     426   44    3 :     B-418( 44 102);
     426   43    4 :     B-418( 43 102);
   5.364   12    1 :     B-512( 12 100);
   8.331   39    1 :     B-411( 39  39);
   9.495   45    1 :   B-600.4( 45  45);
   AE450    6    1 :     A-522(  6  28);
 MEC3500   70    1 :     B-512( 70 100);
```

The session 13:

examID	Size	sectID	room		
ING1001	42	1 :	B-415(42/ 83);		
ING1001	42	2 :	B-618(42/ 64);		
ING1001	42	3 :	B-600.3(42/ 60);		
ING1001	42	4 :	A-616(42/ 60);		
ING1001	42	5 :	B-620.2(42/ 60);		
ING1001	42	6 :	B-600.6(42/ 59);		
ING1001	42	7 :	B-315(42/ 54);		
ING1001	42	8 :	B-600.5(42/ 50);		
ING1001	42	9 :	A-416_1(42/ 48);		
ING1001	42	10 :	B-314(42/ 48);		
ING1001	42	11 :	B-637(42/ 47);		
ING1001	42	12 :	B-600.4(42/ 45);		
ING1001	41	13 :	A-622(41/ 45);		
ING1001	41	14 :	B-316.1_1(41/ 45);		
ING1001	41	15 :	B-405(41/ 42);		
ING1001	41	16 :	B-415(41/ 83);		
ING1001	41	17 :	B-411(21/ 39);	B-543(20/ 37);	
ING1001	41	18 :	A-621(21/ 37);	A-609(20/ 36);	
ING1001	41	19 :	A-608(21/ 36);	B-506(20/ 36);	
ING1001	41	20 :	B-304(21/ 36);	B-505(20/ 35);	
ING1001	41	21 :	A-522(21/ 28);	B-618(20/ 64);	
MTH2301	43	1 :	A-532(43/129);		
MTH2301	43	2 :	A-532(43/129);		
MTH2301	43	3 :	A-532(43/129);		
MTH2301	42	4 :	B-418(42/102);		
MTH2301	42	5 :	B-418(42/102);		
MTH2301	42	6 :	B-512(42/100);		
MTH2301	42	7 :	B-512(42/100);		

The session 14:

examID	Size	sectID	room
1.430	17	1 :	B-304(17/ 36);
2.330	44	1 :	B-512(44/100);
2.330	43	2 :	B-512(43/100);
3.213	44	1 :	B-316.1_1(44/ 45);
3.565	22	1 :	A-604(22/ 32);
4.370	48	1 :	A-416_1(48/ 48);
8.543	20	1 :	B-505(20/ 35);
9.266	32	1 :	B-429(32/ 32);
CIV2105	17	1 :	B-304(17/ 36);
IF505	27	1 :	A-522(27/ 28);
IF540	20	1 :	B-418(20/102);
ING1002	41	1 :	A-532(41/129);
ING1002	41	2 :	A-532(41/129);

```
ING1002   41   3 :     A-532( 41/129);
ING1002   41   4 :     B-418( 41/102);
ING1002   41   5 :     B-418( 41/102);
ING1002   40   6 :     B-415( 40/ 83);
ING1002   40   7 :     B-415( 40/ 83);
ING1002   40   8 :     A-622( 40/ 45);
MEC2200   42   1 :     B-405( 42/ 42);

The session 15:
   examID Size sectID     room
   3.352   56   1 :   B-600.6( 56/ 59);

The session 16:
   examID Size sectID     room
   1.416    9   1 :     B-512(  9/100);
     421   45   1 :   B-600.4( 45/ 45);
   5.523    7   1 :     B-418(  7/102);
   GP501   30   1 :     A-604( 30/ 32);
 PHS0102   44   1 : B-316.1_1( 44/ 45);
   SH550   50   1 :   B-600.5( 50/ 50).
 SSH5401   46   1 :     A-532( 46/129);
 SSH5401   45   2 :     A-532( 45/129);
 SSH5401   45   3 :     B-418( 45/102);
 SSH5401   45   4 :     B-418( 45/102);
 SSH5401   45   5 :     B-512( 45/100);
 SSH5401   45   6 :     B-512( 45/100);

The session 17:
  examID  Size sectID     room
 ELE3200   61   1 :     B-618( 61/ 64);
 INF2600   45   1 :     B-418( 45/102);
 INF2600   45   2 :     B-418( 45/102);
 INF2600   45   3 :     B-512( 45/100);
 INF2600   44   4 :     B-512( 44/100);
 MEC2210   47   1 :     A-532( 47/129);
 MEC2210   47   2 :     A-532( 47/129);
 MEC2210   47   3 :     B-637( 47/ 47);
 MEC3210   45   1 :   B-600.4( 45/ 45);

The session 18:
   examID Size sectID     room
  1.430A   18   1 :     B-415( 18/ 83);
   2.465   40   1 : B-316.1_1( 40/ 45);
   3.540   65   1 :     B-415( 65/ 83);
   4.506   60   1 :   B-600.3( 60/ 60);
   5.231    8   1 :     B-314(  8/ 48);
     580   16   1 :   B-600.6( 16/ 59);
```

```
  6.326   27   1 :    A-522( 27/ 28);
  7.519   14   1 :    B-315( 14/ 54);
ING1020   41   1 :    A-532( 41/129);
ING1020   41   2 :    A-532( 41/129);
ING1020   41   3 :    A-532( 41/129);
ING1020   41   4 :    B-418( 41/102);
ING1020   40   5 :    B-418( 40/102);
ING1020   40   6 :    B-512( 40/100);
ING1020   40   7 :    B-512( 40/100);
ING1020   40   8 :    A-616( 40/ 60);
ING1020   40   9 :  B-620.2( 40/ 60);
ING1020   40  10 :  B-600.6( 40/ 59);
ING1020   40  11 :    B-315( 40/ 54);
ING1020   40  12 :  B-600.5( 40/ 50);
ING1020   40  13 :  A-416_1( 40/ 48);
ING1020   40  14 :    B-314( 40/ 48);
ING1020   40  15 :    B-405( 40/ 42);
MEC2410   62   1 :    B-618( 62/ 64);

The session 19:
  examID  Size sectID     room
MTH2401   47   1 :    B-512( 47 100);
MTH2401   46   2 :    B-512( 46 100);
SSH5103   42   1 :    A-532( 42 129);
SSH5103   42   2 :    A-532( 42 129);
SSH5103   41   3 :    A-532( 41 129);
SSH5103   41   4 :    B-405( 41  42);

The session 20:
   examID Size sectID     room
   3.523    8   1 :    B-418(  8 102);
   3.553   47   1 :    B-418( 47 102);
   3.553   46   2 :    B-418( 46 102);
   5.372   13   1 :    B-429( 13  32);
   8.412   28   1 :    A-522( 28  28);
   9.481   49   1 :  B-600.5( 49  50);
 CIV2103   30   1 :    A-604( 30  32);
 INF2500   41   1 :    A-532( 41 129);
 INF2500   41   2 :    A-532( 41 129);
 INF2500   41   3 :    A-532( 41 129);
 INF2500   41   4 :    B-415( 41/ 83);
 INF2500   41   5 :    B-415( 41/ 83);
 ING1000   47   1 :    B-512( 47 100);
 ING1000   47   2 :    B-512( 47 100);
 MEC3200   55   1 :  B-600.6( 55/ 59);
 MIN1101   12   1 :    B-429( 12/ 32);
 SSH5101   46   1 :    B-637( 46/ 47);
```

The session 21:

examID	Size	sectID	room
3.313	57	1 :	B-600.6(57/ 59);
IND2301	45	1 :	A-532(45/129);
IND2301	45	2 :	A-532(45/129);
IND2301	45	3 :	B-512(45/100);
IND2301	45	4 :	B-512(45/100);
SH502	53	1 :	B-315(53/ 54);

The session 22:

examID	Size	sectID	room
1.322	22	1 :	A-522(22/ 28);
3.414	63	1 :	B-618(63/ 64);
7.420	38	1 :	B-316.1_2(38/ 38);
8.351	35	1 :	B-505(35/ 35);
9.380	52	1 :	B-315(52/ 54);
IF302	43	1 :	B-512(43/100);
IF302	43	2 :	B-512(43/100);
INF1101	47	1 :	A-532(47/129);
INF1101	46	2 :	A-532(46/129);
INF1101	46	3 :	B-418(46/102);
INF1101	46	4 :	B-418(46/102);
INF1101	46	5 :	B-637(46/ 47);

The session 23:

examID	Size	sectID	room
2.525	57	1 :	B-600.6(57/ 59);
3.520	41	1 :	B-316.1_1(41/ 45);
ING1025	42	1 :	A-532(42/129);
ING1025	41	2 :	A-532(41/129);
ING1025	41	3 :	A-532(41/129);
ING1025	41	4 :	B-418(41/102);
ING1025	41	5 :	B-418(41/102);
ING1025	41	6 :	B-512(41/100);
ING1025	41	7 :	B-512(41/100);
ING1025	41	8 :	B-415(41/ 83);
ING1025	41	9 :	B-415(41/ 83);
ING1025	41	10 :	B-618(41/ 64);
ING1025	41	11 :	B-600.3(41/ 60);
ING1025	41	12 :	A-616(41/ 60);
ING1025	41	13 :	B-620.2(41/ 60);
ING1025	41	14 :	B-315(41/ 54);
ING1025	41	15 :	B-405(41/ 42);
Z-610	17	1 :	B-512(17/100);

The session 24:

```
   examID Size sectID      room
   2.573   20    1 :     B-418( 20/102);
   3.310   25    1 :     A-522( 25/ 28);
   3.416   58    1 :   B-600.6( 58/ 59);
   3.521   15    1 :     A-604( 15/ 32);
   4.517   62    1 :     B-618( 62/ 64);
   5.310   11    1 :     A-604( 11/ 32);
   9.455   16    1 :     B-512( 16/100);
   GP502   37    1 :     B-543( 37/ 37);
 GPM5260   32    1 :     B-429( 32/ 32);
 IND2801   53    1 :     B-315( 53/ 54);
 INF2300   41    1 :     B-418( 41/102);
 INF2300   41    2 :     B-418( 41/102);
 INF2300   41    3 :     B-512( 41/100);
 INF2300   41    4 :     B-512( 41/100);
 INF2300   41    5 : B-316.1_1( 41/ 45);
 MEC2500   43    1 :     A-532( 43/129);
 MEC2500   43    2 :     A-532( 43/129);
 MEC2500   43    3 :     A-532( 43/129);
 MEC2500   42    4 :     B-405( 42/ 42);
 MEC3300   50    1 :   B-600.5( 50/ 50);

The session 25:
   examID Size sectID      room
   IF518   30    1 :     A-604( 30/ 32);
   IT430   49    1 :   B-600.5( 49/ 50);
   Z-620   21    1 :     A-522( 21/ 28);

The session 26:
   examID Size sectID      room
   1.509   17    1 :   B-600.3( 17/ 60);
   2.401   23    1 :     B-429( 23/ 32);
   4.340   24    1 :     A-604( 24/ 32);
   5.558   18    1 :     A-616( 18/ 60);
   7.331   16    1 :   B-620.2( 16/ 60);
   7.504    9    1 :     B-429(  9/ 32);
   7.532   15    1 :     B-304( 15/ 36);
   8.310   45    1 :   B-600.4( 45/ 45);
   9.351   61    1 :     B-618( 61/ 64);
   AE430   35    1 :     B-505( 35/ 35);
 ELE1400   43    1 :     A-532( 43/129);
 ELE1400   43    2 :     A-532( 43/129);
 ELE1400   43    3 :     A-532( 43/129);
 ELE1400   43    4 :     B-512( 43/100);
 ELE1400   43    5 :     B-512( 43/100);
 ELE1400   43    6 :     B-415( 43/ 83);
 ELE1400   43    7 :     A-622( 43/ 45);
```

ELE2600	41	1 :	A-616(41/ 60);
ELE2600	41	2 :	B-620.2(41/ 60);
ELE2600	40	3 :	B-415(40/ 83);
ELE3700	51	1 :	B-418(51/102);
ELE3700	50	2 :	B-418(50/102);
GLQ2101	21	1 :	B-304(21/ 36);
IF304	43	1 :	B-600.3(43/ 60);
IF304	43	2 :	B-637(43/ 47);
IF510	28	1 :	A-522(28/ 28);
MEC2110	44	1 :	B-316.1_1(44/ 45);
MTH0103	42	1 :	B-405(42/ 42);
Z-010	40	1 :	B-314(40/ 48);

The session 27:

examID	Size	sectID		room
3.511	77	1 :	B-415(77/ 83);
ELE2301	45	1 :	B-512(45/100);
ELE2301	45	2 :	B-512(45/100);
GLQ1101	42	1 :	B-405(42/ 42);