UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UN ALGORITHME HEURISTIQUE POUR L'ATTRIBUTION DES COURS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

NING XIA

DÉCEMBRE 2006

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

A HEURISTIC ALGORITHM FOR COURSES ASSIGNMENT

A THESIS

SUBMITTED

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER IN COMPUTER SCIENCE

BY

NING XIA

DECEMBER 2006

ACKNOWLEDGEMENTS

Most of all I would like to thank my supervisor, Professor Guy Tremblay. His wide knowledge and experience helped me going in the right direction; his logical thinking showed me the way of doing research; his patience and encouragement gave me confidence and inspiration; his conscientiousness showed me the attitude of doing research. I am also grateful for the financial support he provided during my research (through an NSERC grant #RGPIN183776).

I also want to thank my father and mother for their constant support and encouragement.

Finally, I would like to especially thank my wife for her love and patience during this period.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

RÉSUMÉ

À chaque semestre, la direction du Département d'informatique de l'UQAM doit convenablement affecter les divers groupes-cours disponibles à ses professeurs, en se basant sur leurs préférences et certaines règles adoptées par l'assemblée départementale. Le but de notre travail est développer un algorithme qui trouve une solution de bonne qualité à ce problème, et ce dans un temps raisonnable.

Dans ce mémoire, nous appliquons deux approches différentes pour résoudre le problème d'affectation des groupes-cours. La première est basée sur un algorithme *branch-and-bound*, qui est un algorithme de recherche exhaustif et complet. La deuxième est basée sur un processus heuristique de recherche local amélioré.

Plus particulièrement, nous nous concentrons sur l'algorithme de recherche local. Des algorithmes de recherche locale divers sont présentés, entre autres, la recherche locale guidée et le recuit simulé. Ce que nous faisons n'est pas simplement d'appliquer ces algorithmes et les combiner ensemble. Nous essayons aussi de nous inspirer de ces algorithmes pour mettre au point quelques nouvelles idées appropriées pour notre propre problème.

Finalement, nous utilisons les résultats expérimentaux obtenus par l'algorithme *branch-and-bound* comme point de référence en ce qui concerne la qualité de la solution et le temps d'exécution pour évaluer les autres approches. Les résultats que nous avons obtenus de données réelles et des données générées aléatoirement montrent que notre algorithme effectue un bon travail tant en termes de qualité de solution que de temps d'exécution. Les résultats montrent aussi que notre algorithme peut trouver une bonne solution à de grands problèmes en un temps raisonnable.

ABSTRACT

Each semester, the administration of the Computer Science Department of UQAM must appropriately assign available sections to professors based on their preferences and certain rules. Our goal is to develop an algorithm that can find a solution of good quality in reasonable time.

In this thesis, we apply two different approaches to solve the assignment problem. The first one is based on a branch-and-bound algorithm, which is an exhaustive search algorithm. The second one is based on a local search heuristic process.

We focus more specifically on the local search algorithm. Various local search heuristic algorithms are presented, such as Min-conflicts, Hill-climbing, Various Neighborhood Search, Guided Local Search, and Simulated Annealing. What we do is not simply apply those algorithms and combine them together. We also try to *inspire* ourselves from these algorithms to work out some new ideas suitable for our own problem.

We use the experimental results obtained from the branch-and-bound algorithm as a reference point with respect to solution quality and execution time to evaluate the other approaches. The results we obtained from real data and from randomly-generated data show that our algorithm does a good job in terms of solution quality and execution time. The results also show that our algorithm can find a good solution to large scale problems in reasonable time.

INTRODUCTION

Each semester, the administration of the UQAM Computer Science Department must appropriately assign available sections to the professors based on their preferences and certain rules defined by the Department.

The Computer Science Department publishes all available sections for the next semester, and then professors fill in application forms ("Tâches d'enseignement", Appendix B) to make choice-lists for the sections they wish to give. In order to appropriately assign sections to professors, many factors have to be considered. Most of all, the assignment rules ("Politique relative à l'attribution des charges d'enseignement", Appendix A) must be obeyed. And the professors' preferences should also be respected to the maximum.

The job of assigning sections is currently done manually using an Excel worksheet, which is time-consuming and tedious and largely depends on the Department chair's personal experience. Furthermore, the quality of the final solution is not always guaranteed. So why not introduce an efficient algorithm and let a program do the job?

To meet the needs of the Department, we present heuristic algorithms in this thesis that can solve the assignment problem in an efficient way, yet produce good solutions.

This thesis is organized as follows.

Chapter 1 gives a rough definition of the problem to solve.

Chapter 2 analyzes the characteristics of the problem and overviews some promising approaches, some of which are further discussed in chapters 4, 5 and 6.

Chapter 3 introduces a formal way to evaluate a possible solution, i.e., a candidate assignment.

Chapter 4 describes a first attempt based on a maximum matching algorithm, which as we will see, cannot handle all the Department's rules.

Chapter 5 then presents a branch-and-bound algorithm, which can find the exact optimal solution to a small scale problem.

Chapter 6 presents a local search algorithm and then improves it step by step to obtain a final heuristic local search algorithm.

Chapter 7 presents some experimental results that compare one branch-and-bound algorithm and two local search algorithms we presented.

Finally, in the last chapter, we conclude that the algorithm we proposed does a good job in terms of solution quality and execution time. We also present some possible future work.

CHAPTER I

DEFINITION OF THE PROBLEM

In this chapter, we describe in detail the problem we are going to solve.

1.1 Some important notions

Before we introduce the problem, we need to explain some important notions:

- **Professor**: Professors are consumers. A professor indicates his need for the number of courses he wants to teach, makes his choice-list of courses (in decreasing order of preference), and will be assigned sections according to his needs, choice-list, and the department's assignment rules.

- **Section**: Sections are resources. A section is an activity that happens during a given period of time and involves a given course and a given group of students. During the assignment procedure, sections are assigned to professors according to the professor's need, choice-list, and the department's assignment rules.

- **Professor's Need**: A professor's need is a number that indicates how many sections the professor wants to teach.

- **Professor's Choice-list**: Any professor who wants sections has to fill in his choice-list. A choice-list contains a certain number of choices, which are ordered in their importance to the professor. Each choice represents a section and a section assigned to a professor must be in the professor's choice-list (see appendix B for the exact form filled out by professors to indicate their choice-list). In what follows, we will

assume that the choices in a choice-list are arranged in the order of their priority from high to low.

- **Rule**: The Computer Science Department defined the assignment rules ("Politique relative à l'attribution des charges d'enseignement", Appendix A). In this thesis, a rule is a constraint though it may have different meaning in different context. A rule defines a complex relationship among professors, sections, choice-lists, previous assignments, and other related data, which restricts some assignments of sections under certain circumstances. In a final solution, as many rules as possible should not be violated.

- **Assignment**: An assignment is a solution to our problem. An assignment defines connections between professors and sections, i.e., what sections will be taught by each professor.

1.2 Brief description of the problem

We describe our problem and goal briefly by using the notions mentioned above.

Every semester, the Computer Science Department publishes all available *sections* for the next semester, and then *professors* indicate their *needs*, make their *choice-lists*. An *assignment* is then worked out by respecting the *rules* and the professors' preferences (*needs* and *choice-lists*) to the maximum.

Our goal is to find an *assignment* of good quality in a reasonable time, within a few minutes.

1.3 Functional Model: the context of our problem

The Unified Modeling Language (UML) is a general-purpose modeling language, which is designed to specify, visualize, and construct software systems (Larman, 2004).

A Use Case is used to describe the system's functional requirements from the user's point of view. Each use case contains one or more scenarios. A scenario uses the interactions between actors to describe how a specific function is achieved

The following use case describes the functional requirements associated with our problem at the summary level (for more details, please see Appendix C), which will help understand the *context* in which our solution will fit.

Range: University

Level: Summary

Actors: Professor, Department Chair, Program Director, Administrative Agent

Preconditions: The information about the professors is available and the assignment rules are defined by the Department.

Scenario:

(Process the information about previous semesters, depending on the rules)

1. The Administrative Agent indicates the courses assigned to professors in previous semesters.
2. The Program Director indicates the warnings for poor evaluation in previous semesters.

(Define the information for current semester)

3. The Program Directors indicate the involvement level of professors for the current semester.
4. The Administrative Agent defines the sections of the current semester.
5. The Administrative Agent indicates the courses' coordinators of the current semester.

(Make the demand)

6. The Professors indicate their choices of sections.

7. The Department Chair verifies the professors' choices.

(Do the assignment)

8. The Department Chair pre-assigns courses to some professors in consultation with the Program Directors.

9. The Department Chair executes the section assignment procedure.

Step 4 defines the sections to assign; steps 6 and 7 gather the information about choice-lists; and steps 1, 2, 3, 5, 8 collect all information required by the various rules.

This thesis concentrates on step 9, which is the assignment procedure. We assume that all required data is available at that point. So, our goal is to develop an algorithm that can find a good assignment within a reasonable period of time.

1.4 Object Model

In UML, a class diagram is used to describe a system's conceptual structure by showing classes and relationships among them. More precisely, the goal is to give an abstract description of the data associated with the problem.

The following class diagram gives us a static view of the data involved in our problem, and helps us better understand the relationships among those classes.

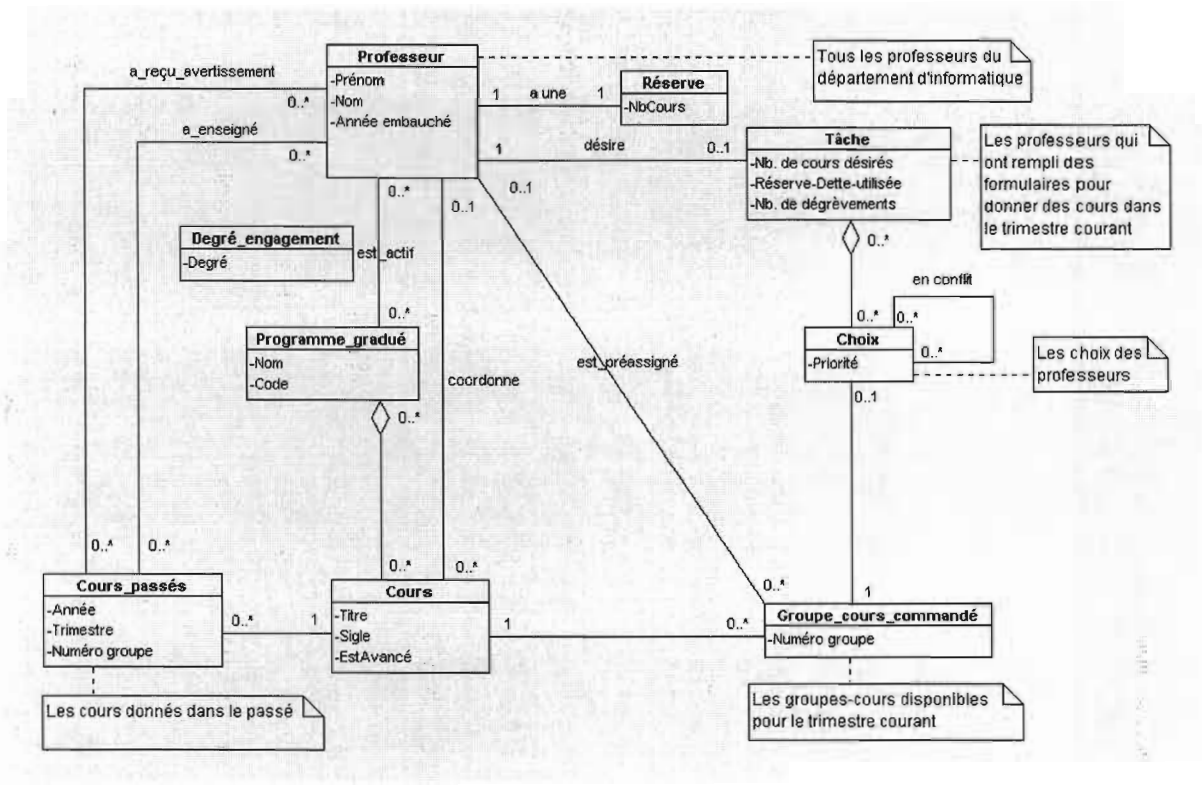


Figure 1.1 Class diagram

1.5 Rules

In the original document from the Computer Science Department, seven rules are stipulated to guide the assignment of sections ("Politique relative à l'attribution des charges d'enseignement", Appendix A). Furthermore, those rules have to be applied in a specific order.

- Rule #1 stipulates the number of choices a professor may have.
- Rule #2 stipulates that a professor who has received two warnings on a given course over the last two or three successive semesters will not be assigned again the same course for the next six semesters.

- Rule #3 stipulates that new professors and professors who are currently engaged in some graduate programs have priority in being assigned a graduate course.

- Rule #4 stipulates that a course coordinator has priority in being assigned a section that he coordinates as long as the section is indicated in his first two choices.

- Rule #5 stipulates that each professor has to be assigned at least one section in his first two choices.

- Rule #6 stipulates that a professor who gave a course in the past has priority in being assigned the same course again. However, this rule can only be applied at most twice on the same course.

- Rule #7 stipulates that a professor who has been assigned a graduate course does not have priority in being assigned another graduate course.

1.6 Definition of the problem

So far (sections 1.1 - 1.5), we have briefly explained the problem in natural language. In order to precisely define the problem, we will try to define the problem in a slightly more formal manner (Tremblay, 2004).

First of all, we define some types. All those types are based on three basic collection types, which are homogeneous collections (all elements are of the same type):

- Set: A set is an unordered collection of elements, where multiplicity is ignored.

- Map: A map is a collection containing pairs of key and value, so that each defined key is bound to a single definition.

- Sequence: A sequence is an ordered collection of elements.

We assume that Section, Prof, Rule, and Integer are four basic primitive types.

For defining our problem, we define some types as follows:

```
TYPE Sections = set{Section}
TYPE Profs = set{Prof}
TYPE ChoiceLists = map{Prof, sequence{Section}}
TYPE Needs = map{Prof, Integer}
TYPE Rules = sequence{Rule}
TYPE Assignment = map{Section, Prof}
```

These types come from the notions we mentioned in section 1.1. They can be regarded as formal representations of those notions. A Sections represents a group of elements of type Section. A Profs represents a group of elements of type Prof. A ChoiceLists represents the mapping relationships between Profs and their ChoiceLists. A Needs represents the mapping relationships between Profs and their Needs. A Rules represents a group of ordered elements of type Rule. An Assignment represents the mapping relationships between Sections and Profs.

Using these types, we then define the input data for our problem as follows:

```
sections: Sections
profs: Profs
choiceLists: ChoiceLists
needs: Needs
preAssigned: Assignment
rules: Rules
```

Finally, we define the output result:

```
assignment: Assignment
```

At this point, we defined the form of a solution. Furthermore, we need to have a clear image of the solution we desire. As we mentioned before, our goal is to find a good solution. But what is a good solution?

A good solution is a solution that is very close to an optimal solution. Then what is an optimal solution?

Satisfying all rules (constraints) does not necessarily make a solution optimal. Reciprocally, a solution with violation of rules might still be an optimal solution because

sometimes a solution that satisfies all constraints simply does not exist. Suppose that there exist more than one solution that satisfies all rules. In that case, we should find out the best among them by measuring other factors. So whether a solution satisfies all rules is not the only thing we should consider.

So an optimal solution is a solution that meets the criteria in the following order:

1. The assignment breaks as few rules as possible — we will describe it more precisely in chapter 3.

2. The assignment assigns as many sections as possible.

3. The assignment assigns as many choices at the beginning of the professors' choice-lists as possible, i.e., it tries to obey the preferences indicated by the professors.

CHAPTER II

CHARACTERISTICS OF THE PROBLEM

In this chapter, we explain how we started to tackle our problem. First, we analyzed the problem to find out what category it belonged to. Then, we studied some of the well-known approaches used for solving similar problems. Those approaches were regarded as potential solutions to our problem. Last, we tried to apply what we learned on our problem. Eventually, as we will show, the problem was solved by applying adaptations, modifications, and inspirations from those potential approaches.

As we stated in the previous chapter, an optimal solution must meet the following criteria, which are ordered in priority from high to low:

1. The assignment breaks as few rules as possible.
2. The assignment assigns as many sections as possible.
3. The assignment assigns as many choices at the beginning of the choice-lists as possible.

As we will explain, this problem is a combination of two different types of problems: Constraint Satisfaction Problem (CSP) and Maximum Matching Problem.

- If we only consider criterion 2 or 3, then it is a Maximum Matching Problem.
- If we only consider criterion 1, then it is a Constraint Satisfaction Problem (or a Partial CSP).

Our assignment problem has the characteristics of both matching problem and constraint satisfaction problem. Thus, our goal is to find a maximum matching on the base that the constraints are satisfied to the maximum.

2.1 Maximum Matching Problem

Hopcroft and Karp (Hopcroft and Karp, 1973) proposed an algorithm that computes maximum matchings in bipartite graphs in time $O(\sqrt{n}m)$. Another algorithm proposed by Harold Kuhn and revised by James Munkres, known as Munkres Assignment Algorithm, solves maximum weighted matching problem in time $O(n^3)$ (Munkres, 1957). Edmonds' algorithm also can solve unweighted and weighted matching problem in time $O(n^3)$ (Edmonds, 1965). We will discuss Munkres algorithm in chapter 4.

Max-flow algorithms also can be used for computing maximum matchings in bipartite graphs by using a simple transformation. By adding a super source that connects to all other sources and a super sink that connects to all other sinks, a maximum matchings in a bipartite graph is transformed into a network flow problem (Johnson and McGeoch, 1993).

Branch-and-bound also can solve Maximum Matching Problem though the execution time is exponential. We will discuss branch-and-bound further in chapter 5.

2.2 Constraint Satisfaction Problem (CSP)

Generally speaking, a CSP consists of a set of variables and a set of constraints. Each variable is associated with a domain of possible values. And each constraint limits the value the variables can simultaneously take. The task is to assign each variable a value without violating any rules. An assignment that does not break any constraints is called a *consistent assignment.*

2.2.1 Partial Constraint Satisfaction Problem

Sometimes, a solution to a CSP simply does not exist or it takes too long to obtain. The notion of Partial CSP was thus introduced to handle such a situation. By relaxing some

constraints, A Partial CSP makes it possible to find a near-solution to an originally hard problem in a reasonable time.

Weakening a Constraint Satisfaction Problem

In real life, we sometimes need to weaken a CSP without dramatically changing its characteristics in order to find a near-solution. Usually, we do some changes to make the original CSP be a subclass of a Partial CSP. For example: a constraint "a nurse works from Monday to Friday" is changed to "a nurse works from Monday to Friday and she may also work on Saturday if needed".

A CSP can be weakened in the following ways:

- Removing a variable
- Enlarging a variable's domain
- Removing a constraint
- Enlarging a constraint's domain

The most fundamental way of weakening a problem is to relax a constraint by enlarging its domain.

Constraint Hierarchies

In the context of Partial CSP, constraints can be divided into two categories:

- Hard constraints: Constraints that must absolutely be satisfied.
- Soft constraints: Constraints that may be violated and the violations are subjected to a penalty in some appropriate objective function.

Furthermore, soft constraints can be divided into different levels. The violations of soft constraints are then minimized level by level (Wilson and Borning, 1993).

In a hierarchical approach, a comparator is defined to help select a solution via minimizing violations of hierarchic constraints. There are various possible comparators, such as weighted-sum comparator, and worst-case comparator. A lexicographic comparator is designed to compare two solutions lexicographically (Rudova, 1998).

2.2.2 Various approaches proposed for solving CSPs

Depending on the type of search, algorithms can be classified in two types: exact algorithms and heuristic algorithms. Exact algorithms do a *systematic* search to find the optimal solution. Heuristic algorithms are designed to find good solutions in a reasonable time. Put simply, exact algorithms are designed to find *the best* solution and heuristic algorithms are designed to find a *good* solution.

Exact algorithms:

- Backtracking. A backtracking algorithm tries all possible combinations in order to get a solution. Many partial solutions are avoided during the searching process. A recursive depth-first search backtracking algorithm is presented for solving CSPs in (Russell and Norvig, 2002).

- Branch-and-bound. This approach divides the search space into sub-regions and uses a bounding function to safely prune many of those sub-regions. Palacios presented a branch-and-bound algorithm combined with heuristic search for solving CSP-based planning tasks (Palacios and Geffner, 2002).

- Constraint propagation. This approach solves problems by removing values that cannot be part of any solution. Arc Consistency #3 proposed by Alan Mackworth is a well known algorithm used for solving CSP (Mackworth, 1977).

Heuristic algorithms:

- Min-Conflicts algorithm. This approach randomly chooses a conflicting variable, and then picks a value that minimizes or at least does not increase the number of violations (Minton, 1992).

- Hill-Climbing. This heuristic approach explores the local neighborhood of a solution and moves to a neighbor with a better evaluation value (Russell and Norvig, 2002).

- Simulated Annealing. This approach simulates the controlled cooling process in metallurgy that increases the size of the crystal (Kirkpatrick, 1983). It allows "bad" moves in order to escape from a local optimum and the probability of taking "bad" moves decreases while the temperature drops.

- Genetic Algorithm. This approach simulates the life evolution process and belongs to population-based algorithms (Goldberg, 1989). A GA starts with a randomly generated population and evolution happens in generations. Individuals in the current population are evaluated, then selected according to their fitness values, and then mutated and/or recombined to form a new population. This evolution process repeats until certain criteria are met. GAs have been growing popular over the last decade for solving many kinds of problems. They have also been used to find near-optimal solutions to CSPs (Tsang, 1993).

2.3 Summary

We briefly introduced some promising approaches to our problem in this chapter. As we will see, a branch-and-bound algorithm will be proposed in chapter 5 as a reference algorithm, meanwhile, Min-Conflicts, Hill-Climbing and Simulated Annealing will be used to develop our own local search algorithm.

CHAPTER III

EVALUATION OF A SOLUTION

As we stated in chapter 1, an optimal solution must meet the following criteria, which are ordered in priority from high to low:

1. The assignment breaks as few rules as possible.
2. The assignment assigns as many sections as possible.
3. The assignment assigns as many choices at the beginning of the choice-lists as possible.

These criteria are hierarchical, which means that the criteria at higher level are always more important than those at lower level. For example, an assignment with 1 violation and 30 assigned sections is better than an assignment with 2 violations and 50 assigned sections.

In order to compare two solutions in terms of quality, we present the lexicographic ordering.

3.1 Lexicographic Ordering

Suppose we have two vectors X, Y of n components defined as $X = [X_0, X_1, \ldots, X_{n-1}]$ and $Y = [Y_0, Y_1, \ldots, Y_{n-1}]$.

For vectors of two components (n=2), we define the lexicographic ordering as follows:

$[X_0, X_1] =_{lex} [Y_0, Y_1] \Leftrightarrow X_0 = Y_0$ and $X_1 = Y_1$

$[X_0, X_1] >_{lex} [Y_0, Y_1] \Leftrightarrow X_0 > Y_0$, or $X_0 = Y_0$ and $X_1 > Y_1$

$[X_0, X_1] <_{lex} [Y_0, Y_1] \Leftrightarrow X_0 < Y_0$, or $X_0 = Y_0$ and $X_1 < Y_1$

For vectors of more than two components, we define the lexicographic ordering recursively as follows:

$X =_{lex} Y \Leftrightarrow X_0 = Y_0$ and $[X_1, \ldots, X_{n-1}] =_{lex} [Y_1, \ldots, Y_{n-1}]$

$X >_{lex} Y \Leftrightarrow X_0 > Y_0$, or $X_0 = Y_0$ and $[X_1, \ldots, X_{n-1}] >_{lex} [Y_1, \ldots, Y_{n-1}]$

$X <_{lex} Y \Leftrightarrow X_0 < Y_0$, or $X_0 = Y_0$ and $[X_1, \ldots, X_{n-1}] <_{lex} [Y_1, \ldots, Y_{n-1}]$

3.2 Solution Value

In our case, a solution can be valued as a vector of length three, $V = [V_{Viol}, V_{Sect}, V_{Pref}]$.

- V_{Viol} indicates the number of violations of rules in an assignment multiplied by -1, which keeps the consistency with other components: the greater the vector component is, the better the solution is (see below).
- V_{Sect} indicates the number of assigned sections in an assignment.
- V_{Pref} indicates the sum of the weights of every assigned section. The choices at the beginning of a professor's choice-list weigh more than those at the end.

In the lexicographic ordering we defined above, the greater (relative to $>_{lex}$) the vector V is, the better the assignment is. Thus, the optimal solution to our problem has a value that equals $\max([V_{Viol}, V_{Sect}, V_{Pref}])$ among the solution value vectors associated with all possible solutions.

After having introduced the lexicographic ordering, we are able to evaluate an assignment precisely. Because we have ordered rules, the violations of those rules are also ordered. V_{Viol} was defined as a number for the sake of simplicity. We are going to improve and redefine V_{Viol} as a vector with the same lexicographic ordering we introduced in section 3.1.

As we have introduced in section 1.5, there are seven ordered rules: Rule #1, Rule #2, Rule #3, Rule #4, Rule #5, Rule #6 and Rule #7. Accordingly, we have $V_{Viol} = [V_{R1}, V_{R2}, V_{R3}, V_{R4}, V_{R5}, V_{R6}, V_{R7}]$. V_{R1} represents the number of the violations of Rule #1, and so on.

By putting $V = [V_{Viol}, V_{Sect}, V_{Pref}]$ and $V_{Viol} = [V_{R1}, V_{R2}, V_{R3}, V_{R4}, V_{R5}, V_{R6}, V_{R7}]$ together, we define our solution value as $V_{Solu} = [V_{R1}, V_{R2}, V_{R3}, V_{R4}, V_{R5}, V_{R6}, V_{R7}, V_{Sect}, V_{Pref}]$. The fewer violations there are, the better the solution is. Meanwhile, the more sections are assigned, the better the solution is. As we mentioned above, in order to keep the consistency (the greater the V is, the better the assignment is), we turn the values pertaining to violations into non-positive value by multiplying them by -1. By doing so, we have a complete maximization problem. That is to say, for any variable, the higher it gets, the better the solution is. By using Solution Value, we know exactly how to compare two assignments in terms of quality. For example, an empty assignment is not the worst solution because it is better than those that have violations. One more example: suppose we have a Solution Value V = [0, -1, 0, -2, 0, 0, 0, 50, 123], then we know that there is one violation of Rule #2, two violations of Rule #4, 50 assigned sections, and the total weight of the preferences of the professors is 123 in that assignment.

3.3 Violation of a rule

We have introduced the solution value we adopt for evaluating possible solutions to our problem. What is left to explain is "what exactly a violation of a rule is"? A violation of a rule is an offence against the rule, which prevents assigning certain sections to some professors under certain circumstances. The rules from the Computer Science Department specify which professors have higher priorities in getting a section under certain circumstances (see appendix A).

3.3.1 Determining a violation

Suppose that Professor A needs two sections and he gives five choices. In an assignment, the sections indicated by his second and fourth choices are assigned to Professor A. Professor A will check the assignment and wonder why we did not assign the sections indicated by his first and third choices to him. We have to give him a reason for that, which is

either the sections indicated by his first and third choices are assigned to professors with higher priorities (the rules specify so) or they are assigned to professors with equal priorities (bad luck or global optimization). When his first or third choice is assigned to a professor with lower priority, a violation occurs. Professor A will never ask why we did not assign the section indicated by his fifth choice. Thus we do not check the section indicated by his fifth choice, because it does not cause a violation no matter whatever it is.

Let us introduce some notions, which are used to determine whether a violation exists. Those notions are defined relative to the type and variables defined at the end of chapter 1.

Need-met Professor: A professor whose Need is met, which means he has been assigned enough sections relative to what he needs.

```
Need-met(p: Professor) ⇔
  NUMBER(s: Section SUCH THAT assignment[s] = p :: s) = needs[p]
```

Need-not-met Professor: A professor whose Need is not met, which means he has not been assigned enough sections that he needs.

```
Need-not-met(p: Professor) ⇔
  NUMBER(s: Section SUCH THAT assignment[s] = p :: s) < needs[p]
```

Need-met Point: A special choice in the Choice-list. We suppose the choices are considered from the first to the last. Before the Need-met point, the professor has not been assigned all the sections he needs; after this point inclusively, the professor has been assigned all the sections he needs. Obviously, for Need-not-met Professors, Need-met Point does not exist.

```
Need-met(c: Choice) ⇔
  NUMBER(s: Section SUCH THAT choiceLists[p].indexOf(s) < c AND assignment[s] = p :: s)
    < needs[p]
  AND
  NUMBER(s: Section SUCH THAT choiceLists[p].indexOf(s) ≤ c AND assignment[s] = p :: s)
    = needs[p]
```

Ineffective Choice: For a Need-met Professor, Ineffective Choices are the choices after the Need-met Point. A Need-not-met Professor does not have Ineffective Choices. Ineffective Choices have no effect on determining whether a violation exists.

Effective Choice: For a Need-not-met Professor, Effective Choices are all his choices. For a Need-met Professor, Effective Choices are the choices before and at the Need-met Point. We call those choices Effective Choices because those choices have effect on determining whether a violation exists.

- **Accepted Choice**: An Effective Choice such that the section indicated by this choice is assigned to the professor.

- **Rejected Choice**: An Effective Choice such that the section indicated by this choice is not assigned to the professor.

We create the following table to illustrate these notions by using the example we mentioned at the beginning of this section, where professor A needs two sections and he gives us five choices:

Professor A (Professor's Need = 2)	**Result**	**The number of assigned sections from the first to the n^{th} choice**	**Is it a Need-met Point?**	**What kind of choice is it?**
1^{st} choice	not assigned	0	no	Effective Choice (Rejected Choice)
2^{nd} choice	Assigned	1	no	Effective Choice (Accepted Choice)
3^{rd} choice	not assigned	1	no	Effective Choice (Rejected Choice)
4^{th} choice	Assigned	**2**	**yes**	Effective Choice (Accepted Choice)
5^{th} choice	not assigned	2	no	**Ineffective Choice**

Table 3.1 An example of Need-met Point, Effective Choice, and Ineffective Choice

As we can see, whether a choice is a Rejected Choice, Accepted Choice, or Ineffective Choice depends on the specific assignment. When we determine violations in an assignment, we check all Effective Choices and we ignore all Ineffective Choices. Since a section may be requested by several professors, we ignore the professors whose choice indicating the section is an Ineffective Choice.

Section's Competing Professor: A professor who has an Effective Choice indicating the section and has not been assigned the section.

Section's Winner Professor: A professor who has an Effective Choice indicating the section and has been assigned the section.

A section's Winner Professor should have higher or equal priority than any of its Competing Professors in getting the section. If not, a violation on the section exists. In order to determine if a violation exists on a section, we first find the Competing Professor with the highest priority, and then we compare it with the Winner Professor. If the Winner Professor has higher or equal priority, then the rules are respected and no violation occurs. Otherwise, the rules are broken and a violation occurs.

A violation of a rule (except Rule #5: see below) is related to one section. Thus, the total number of violations signifies how many sections are wrongly assigned.

Exception to the violation of Rule #5

Rule #5 is different from other rules. Rule #5 (see Appendix A) stipulates that each professor must be assigned at least one section indicated by one of his first two choices. A violation of one of the other rules is related to one section. A violation of Rule #5 is related to an assignment. The possible value of V_{R5} can only be 0 or 1. In practice, Rule #5 cannot always be respected. When Rule #5 is not respected, the more professors comply with Rule #5, the better an assignment is. In order to deal with situations where Rule #5 cannot be respected, we change the meaning of V_{R5}. Instead of indicating whether a violation of Rule #5 exists in an assignment, V_{R5} now indicates the number of the professors who are assigned at least one section indicated by one of their first two choices. This new meaning of V_{R5} works better when comparing two assignments.

Time conflict

Handling time conflicts means not assigning two sections in the same time slot to a professor. In this thesis, we do not deal with time conflict. The document from the Département d'informatique does not describe time conflict in detail. In fact, time slots or schedule are not necessarily involved in our case. A professor may thus indicate any combinations of choices. Furthermore, according to the data we collected from the department, only one or two professors actually wrote a note indicating time conflicting sections each semester.

3.3.2 Level of a violation

If a violation occurs, we also need to know which rule is broken, which signifies the level of the violation.

The rules from the Computer Science Department are ordered. A rule with a higher priority always overrules a rule with a lower priority. In this thesis, Rules are defined as an ordered collection of rules.

```
TYPE Rules = sequence{Rule}
```

A violation is related to a section, a rule, the Winner Professor, and a Competing Professor. A rule describes a relationship among two professors and a section. It stipulates which professor deserves the section. When comparing two professors, rules are applied one by one in the priority order until a violation is detected or all rules have been applied. When a violation is detected, we then know which rule is violated. Because of the ordering of the rules, there is no need to check the rest of the rules.

3.3.3 Simplified Solution Value

In section 3.2, we defined our solution value as $V_{Solu} = [V_{R1}, V_{R2}, V_{R3}, V_{R4}, V_{R5}, V_{R6}, V_{R7}, V_{Sect}, V_{Pref}]$. And in subsection 3.3.1, we separated Rule #5 from the other rules and V_{R5} now indicates the number of the professors who are assigned at least one section indicated by one of their first two choices. Suppose we have an assignment that does not break any rules. Its Solution Value should then be a vector like $[0, 0, 0, 0, V_{R5}, 0, 0, V_{Sect}, V_{Pref}]$. For convenience's sake, we define a Simplified Solution Value as $V_{Simp} = [V_{R5}, V_{Sect}, V_{Pref}]$ for assignments that do not contain any violation:

- V_{R5} indicates the number of professors who have been assigned at least one section from one of their first two choices.

- V_{Sect} indicates the total number of assigned sections

- V_{Pref} indicates the total weight of every assigned section relative to the professors' preferences

We will use this *simplified* ordering in chapter 7, when we present some experimental results.

3.4 Summary

In this chapter, we introduced the Solution Value, which is based on the Lexicographic Ordering, to evaluate an assignment. Then we explained what a violation is by using the terms of Need-met Point, Effective Choice, Ineffective Choice, Competing Professor, and Winner Professor. At the end of this chapter, we also explained the level of a violation.

By introducing these notions, we will be able to evaluate an assignment. In our case, an optimal solution will be an assignment that has the highest Solution Value.

CHAPTER IV

MUNKRES ASSIGNMENT ALGORITHM

4.1 Related work

Suppose that we have m workers and n jobs, and suppose also that each job has a different profit for each worker and each worker can only perform one job. Here is an example of profit matrix:

	Job 1	**Job 2**	**Job 3**	**Job4**
Worker 1	1	2	3	4
Worker 2	12	13	14	5
Worker 3	11	16	15	6
Worker 4	10	9	8	7

Table 4.2 An example of a profit matrix for an assignment problem

We want an assignment that maximizes the total profit, which is called *Maximum Matching* problem. Munkres Algorithm is a well-known algorithm that finds the optimal solution in time of $O(n^3)$ (Munkres, 1957).

4.2 Similarities between job assignment problem and our problem

Our problem has something in common with the maximum matching problem. In our case, we hope to maximize the number of professors who get one of their first two choices assigned and to maximize the total number of assigned sections. Also, each professor's choice on the application form has a different preference weight depending on its position on the list. We also hope to maximize the total preference value of a solution.

We tried to utilize an approach for solving weighted maximum matching problem, which is Munkres' assignment algorithm, to tackle our own problem. Munkres's algorithm, also called the Hungarian algorithm, is well known for solving profit-maximization assignment problem.

4.3 Adapting to our problem

In order to use Munkres Algorithm, we have to customize the weight for each section and handle the situation where a professor needs multiple sections.

Weight

We apply different weight on each matching of a professor and a section, where the weight is position-related. The choices at the beginning of a professor's choices list weigh more than those at the end. Weight is also rule-related. A section may have different weights for different professors due to the rules. For example: according to Rule #3 (see Appendix A), a graduate course has more weight for a new professor than an old professor.

Multiple jobs

In the original assignment problem, one worker gets only one unique job. In our case, a professor may need to get 1, 2, or 3 sections. We adapt our assignment problem to Munkres' algorithm by duplicating professors who want more than one section. For example, if Prof01 needs two sections, then Prof01 will be duplicated into Prof01a and Prof01b. Each duplicated professor might then get one different section. We then merge them back after the algorithm finishes.

4.4 Difficulties

When attempting to use Munkres Algorithm to solve our problem, we encountered the following difficulties:

- We cannot transform all the rules into static weights. Some weights may vary during the assigning process. For example, Rule OneGraduate (Rule #7, see Appendix A) dynamically causes some changes in weight whenever a professor is assigned his first graduate course, which Munkres Algorithm cannot handle.

- It is difficult to use vector values instead of numeric values in Munkres Algorithm.

Due to these difficulties, we believe that Munkres Algorithm and the likes of it are not appropriate approaches to our problem.

CHAPTER V

BRANCH-AND-BOUND

5.1 Related work

Branch-and-bound is a traditional approach to find the optimal solution to many kinds of optimization problems (Neapolitan, 1997). The algorithm usually takes exponential time to find the optimal solution, even with the help of a good bounding function.

A branch-and-bound algorithm consists of two parts:

The first part is branching. The algorithm tries to explore all regions of the entire search space by exploring each sub-region recursively.

The second part is bounding. Utilizing a bounding function, we can safely estimate that the best solution that we can find by exploring a given sub-region will not be better than the best solution found so far. Therefore, we can avoid exploring those sub-regions without missing the optimal solution. This procedure is usually called pruning.

5.2 Purpose of implementing this algorithm

Branch-and-bound is not an effective approach for solving large Constraint Satisfaction Problems. Variables that seem locally consistent are not always globally consistent. Checking consistency in a global scope is often costly. This prevents us from effectively trimming branches that lead to infeasible assignments.

However, we know that a branch-and-bound algorithm, for small problems, can find an exact optimal solution. The goal of applying this approach is thus to get a reference point (a

benchmark) with respect to solution quality and execution time. Thus, this algorithm is implemented for the purpose of comparison with the other approaches we will propose.

5.3 Ways to improve performance

Many techniques can be applied to improve the performance of a branch-and-bound algorithm.

Exploration order

We can sort sections in the order of their popularity. We can then start exploring sections that fewer professors want to give. In other words, variables with smaller value domains can be explored earlier. The reason is that pruning usually happens in the later stage of exploration. So, we should minimize the number of branching before reaching pruning points. This is why we explore variables with a smaller branching factor first.

Bounding function

A better bounding function makes us explore less nodes, thus we are likely to get the optimal solution more quickly. On the other hand, a better bounding function takes more time at each node to compute, thus the total exploration time may increase. So, in practice, there is always a compromise between the quality of the bounding function and the time it takes to compute.

In our case, a tight bounding function takes much more time than a simple bound function does. Considering the huge number of nodes to explore, we adopt a simple bound function which takes time $O(1)$ to compute.

Early good solution

Traditionally, people try to get an early good solution by exploring the best promising node at each level in a breadth-first branch-and-bound algorithm. We also can use a good solution obtained from another algorithm. We will introduce a local search approach in chapter 6 to get a good solution quickly. Then the good solution is used to improve the

efficiency of pruning. It turns out that an early good solution reduces lots of unnecessary exploration and makes pruning more efficient.

5.4 Pseudocode

The following procedures describe, in pseudocode form, our branch-and-bound algorithm.

```
PROCEDURE performBranchAndBoundExploration(assignment)
BEGIN
   sort sections in the order of popularity
   exploreSolutionSpace(assignment, first_section_in_sections, sections)
END

PROCEDURE exploreSolutionSpace(assignment, section, sections)
BEGIN
   IF isComplete(assignment) THEN
      IF value(assignment) > bestValue THEN
         update(bestAssignment, bestValue)
      ENDIF
   ELSE
      IF estimate(section.next()) >= bestValue THEN
         exploreSolutionSpace(assignment, section.next(), sections)
      ENDIF
      FOR EACH prof IN candidateProfs(section) DO
         IF canBeAssigned(section, prof) THEN
            IF estimate(section.next()) >= bestValue THEN
               exploreSolutionSpace(assignment, section.next(), sections)
            ENDIF
         ENDIF
      ENDFOR
   ENDIF
END
```

This algorithm tries every promising combination to find the exact optimal solution. When isComplete() returns true, which means a combination is completely built, we evaluate the combination to see if it is a better solution or not. At each node, we try every possible way to assign the current section. If the expression estimate(section.next()) >= bestValue is true, which means that a promising branch is found and it may lead to a better solution, we explore this promising branch in a recursive way; otherwise, we can safely ignore the branches that certainly not lead to a better solution. The function canBeAssigned() is used to avoid assigning more sections to a professor than he wants.

Certainly, there are many ways to improve a branch-and-bound algorithm. As we said at the beginning of this chapter, the goal of applying this approach is to get a reference point with respect to solution quality and execution time. The experimental results we get by this algorithm will be used to do comparisons with the local search algorithms we are going to propose in the following chapter.

CHAPTER VI

LOCAL SEARCH

We introduced several heuristic algorithms for solving CSPs in section 2.2.2. Many of them belong to the class of local search algorithms, such as min-conflicts, hill-climbing, and simulated annealing.

Here is an outline of the local search algorithm:

1. Start with an initial solution.
2. Move from the current solution to a successor solution based on a certain strategy.
3. Repeat step 2 until appropriate termination conditions are met.

Local search algorithms move from solution to solution in the space of candidate solutions based on a certain strategy until appropriate termination conditions are met (Blum and Roli, 2003). A candidate solution is not necessarily a neighbor of the current solution and a successor solution is not necessarily better than the current solution.

In this chapter, we start with a simple local search algorithm. Then we improve it step by step by using some local search techniques or some inspirations from them. Finally, we get an efficient algorithm that can find a good solution to our problem in a reasonable time.

6.1 Neighborhood structure

As we have seen, a solution is defined as follows:

```
TYPE Assignment = map{Section, Prof}
```

An assignment is a solution. An assignment consists of sections to which are associated professors. More precisely, a section's associated value could be a professor, or could be null meaning the course is not assigned to anyone. If two assignments differ by only one section's value, we say that they are neighbors.

6.2 Repair-and-stuff algorithm

We will introduce a local search algorithm that can find a good solution quickly. This algorithm is originally inspired by min-conflicts, which is known as one of the fastest local search algorithms for solving large CSP problems in practice (Russell and Norvig, 2002). Then this algorithm will be further improved in sections 6.3 and 6.4.

6.2.1 Origin

As we mentioned in section 2.2.2, min-conflicts is a popular heuristic algorithm for solving CSPs. Here is an outline of the min-conflicts algorithm:

1. Start with a random assignment.
2. Select randomly a variable whose value conflicts with some constraints.
3. Assign to this variable the value that minimizes the number of broken constraints.
4. Repeat steps 2, 3 until appropriate termination conditions are met.

Let us see this algorithm from the point of view of neighborhood. Min-conflicts avoids exploring the whole neighborhood of the current state. It evaluates only the neighbors that are related to a conflicting variable. For example, suppose a conflicting variable's domain contains five possible values, which means the current state has four neighbors. Min-conflicts explores these four neighbors then choose the best one to move to. If solutions are densely distributed in the search space, min-conflicts can be surprisingly effective.

6.2.2 Simple repair algorithm

Violation-free Assignment: A *Violation-free Assignment* is an assignment that satisfies all the rules (except Rule #5, which is treated differently, see section 3.3.1). These rules are sections assignment rules, which are designed as guide for assigning sections. These rules are applied in order to determine professors' priorities on a given section. Because of the characteristics of these rules, finding a violation-free assignment is an under-constrained problem, which means there are many *Violation-free Assignments* to a problem.

Based on the idea of min-conflicts, a possible heuristic algorithm for finding a *Violation-free Assignment* can be constructed as follows:

Simple repair algorithm:

1. Start with a random assignment.
2. Detect a violation of constraints at random. In other words, detect a section that is improperly assigned.
3. Fix the violation by reassigning the section to an appropriate professor.
4. Repeat steps 2, 3 until appropriate termination conditions are met.

The goal of the *simple repair* algorithm is to quickly find a *Violation-free Assignment*. The sections assignment rules play two roles in this algorithm: one role as constraints and the other as guide for assigning sections. As constraints, when we detect violations in step 2, the rules are applied in order on a section's *Winner Professors* and *Competing Professors* to determine whether a violation occurs. As guide for assigning sections, when we select an appropriate professor to assign in step 3, the rules are applied on the section's *Competing Professors* to find the professor with highest priority.

During the process of fixing violations, the total number of assigned sections may decrease. Some sections may have been reassigned to some other professors, and those professors may have been assigned enough sections they need, so the extra sections that are

assigned to those professors will be set to unassigned. For example, professor A needs two sections and has been assigned the sections indicated by his first and fourth choices, and then after repairing a violation, the section indicated by his third choice is reassigned to professor A. In this case, the section indicated by his fourth choice has to be set to unassigned.

After performing the repair process, there may exist some unassigned sections that either were not assigned or were newly set to unassigned. It is highly possible that some of them are needed by some *Need-not-met Professors*. In this situation, there is room to improve the assignment by assigning some of those unassigned sections. We introduce the following notions to help understand how we can improve a *Violation-free Assignment*:

Needed-and-available Section: If a section has one or more *Competing Professors*, then it is needed by one or more professors. If a section has not been assigned to any professor, then it is available for assigning. A *Needed-and-available Section* is a section that has one or more *Competing Professors* and has not been assigned to any professor.

Stuffed Violation-free Assignment: If a *Violation-free Assignment* does not contain any *Needed-and-available Section*, then it is a *Stuffed Violation-free Assignment*. We call it "stuffed" because we keep trying to fill an assignment tightly with *Needed-and-available Sections*.

Unstuffed Violation-free Assignment: If a *Violation-free Assignment* contains any *Needed-and-available Section*, then it is an *Unstuffed Violation-free Assignment*. For example, an empty assignment is an *Unstuffed Violation-free Assignment*.

Stuffing a violation-free Assignment

Assigning *Needed-and-available Sections* may cause violations. In our case, a violation is related to a section's *Winner Professor* and *Competing Professors*. An empty assignment is a *Violation-free Assignment* because it has no *Winner Professors*. Assigning *Needed-and-available Sections* adds *Winner Professors*, which may cause violations. Then we need to repair these violations, and then we may need to assign the *Needed-and-available Sections* created by the previous repair. This process repeats until there is no *Needed-and-*

available Section in the *violation-free assignment*, which turns out to be a *Stuffed Violation-free Assignment*.

Based on the *simple repair* algorithm, we can now introduce an improved algorithm that combines the repairing and stuffing processes together.

Repair-and-stuff algorithm:

1. Start with a random initial assignment.
2. Randomly detect a section that is improperly assigned then fix it by reassigning the section to an appropriate professor.
3. Randomly detect a *Needed-and-available Section* then assign the section.
4. Repeat steps 2, 3 until appropriate termination conditions are met.

This algorithm starts with an initial assignment, and then stuffs the assignment while repairing the violations in it until a *Stuffed Violation-free Assignment* is found. If the initial assignment is empty, then the *repair-and-stuff* algorithm turns into a *solution generator* algorithm, which can be used to quickly generate random initial solutions of good quality for other algorithms. According to our experimental data, *repair-and-stuff* algorithm runs surprisingly fast and the assignments it finds are of good quality.

6.2.3 Constraints

Our problem is also partially a CSP problem. In our case, there exist hard constraints and soft constraints:

Hard Constraints:

1. a professor cannot be assigned more courses than he really needs
2. a section assigned to a professor must be indicated as one of the professor's choices

Soft constraints:

the various constraints associated with the assignment rules from the department

For soft constraints, we use the *repair-and-stuff* algorithm to fix the violations of these constraints.

For hard constraints, we introduce a simple local consistency algorithm for maintaining the hard constraint mentioned above, which also means that *Need-met Professors* do not compete with the other professors for the sections indicated by the choices after the *Need-met Point*. In theory, the *repair-and-stuff* algorithm works also fine with the hard constraint though it takes more time.

Constraint propagation

Put simply, constraint propagation changes the problem without changing its solutions. By reducing the domains of variables, it reduces the search space and thus makes the problem easier to solve.

As we have seen, a solution is defined as follows:

```
TYPE Assignment = map{Section, Prof}
```

An assignment is a solution and consists of sections associated with professors. Based on the second hard constraint, a section's domain contains only the professors who have a choice indicating the section. Our goal is to use size-reduced and dynamically-maintained domains instead of the original domains.

Size-reduced and dynamically-maintained domain: A section's *size-reduced and dynamically-maintained domain* contains only the professors who have an *Effective Choice* indicating the section.

The sections' *size-reduced and dynamically-maintained domains* serve two purposes:

1. When a section is assigned, it speeds up detecting the violation.

2. When a section is not assigned, it speeds up assigning the section.

Section's *Competing Professors* are introduced in section 3.3 for determining violations. A section's dynamic domain contains only the section's *Competing Professors*. We call a section's dynamic domain a section's *Competing Professors List*. All sections' *Competing Professors Lists* can be regarded as the dynamic domains of all sections, which are used to maintain local consistency. The following table is an example demonstrating how a professor is counted as a section's *Competing Professor*.

Professor A (Professor's Need = 2)	**State (if the section is assigned to professor A)**	**The number of assigned sections from the first to the n^{th} choice**	**What kind of choice is it?**	**Is the professor any section's Competing Professor?**
1^{st} choice (section S1)	not assigned	0	Effective Choice (Rejected Choice)	**Yes (S1)**
2^{nd} choice (section S2)	assigned	1	Effective Choice (Accepted Choice)	No
3^{rd} choice (section S3)	not assigned	1	Effective Choice (Rejected Choice)	**Yes (S3)**
4^{th} choice (section S4)	assigned	**2 (Need-met Point)**	Effective Choice (Accepted Choice)	No
5^{th} choice (section S5)	not assigned	2	Ineffective Choice	No

Table 6.3 An example of a section's Competing Professors

If a professor's choice is a *Rejected Choice*, then this professor is a *Competing Professor* of the section related to the *Rejected Choice*. In the above example, Professor A is section S1's and section S3's *Competing Professor* because sections S1 and S3 are related to Professor A's *Rejected Choices*.

There are many powerful consistency algorithms, such as Arc Consistency Algorithm #3 (Mackworth, 1977). But in our case, we only need a simple algorithm to dynamically maintain the local consistency. The following algorithm is introduced to maintain all sections' *size-reduced and dynamically-maintained domains* during the process of

assignment. This method has two parts: one is to initialize the dynamic domains, the other is to maintain them, whose time complexity is $O(1)$ with the use of a hash table.

The first thing to do is to initialize *Sections' Competing Professors Lists* by identifying each *Rejected Choice*.

Initialize Sections' Competing Professors Lists:

1. Start with an empty *Sections' Competing Professors Lists*.
2. Identify one professor's choices.
    2.1. If a choice is a *Rejected Choice*, then put the professor in the related section's *Competing Professors List*.
    2.2. Repeat step 2.1 until all the choices of the professor have been scanned.
3. Repeat step 2 until all professors' choices have been identified.

Once it is done, all that is left to do is to maintain the lists. When a change occurs to a professor (getting a section or losing a section), we re-identify all this professor's choices in order to do some necessary changes in the related sections's *Competing Professors lists*.

Maintain Sections' Competing Professors lists:

1. Re-identify each choice of the professor who just got or lost a section.
    - In case the professor lost a section:
        - If an Accepted Choice or an Ineffective Choice turns into a Rejected Choice, put the professor in the related section's *Competing Professors List*.
    - In case the professor got a section:
        - If a Rejected Choice turns into an Accepted Choice or an Ineffective Choice, remove the professor from the related section's *Competing Professors List*.
        - If an Accepted Choice turns into an Ineffective Choice, set the related section unassigned.
2. Repeat step 1 until all the choices of the professor have been re-identified.

In this maintenance process, some sections may be set to unassigned. For example, professor A needs two sections and has been assigned two sections. After repairing a violation, the section indicated by one of his *Rejected Choices* is reassigned to professor A, the *Rejected Choice* then became an *Accepted Choice*, which is in front of the old *Need-met Point*. Thus, the last *Accepted Choice* before the old *Need-met Point* became the new *Need-met Point*. The old *Need-met Point* turns into an *Ineffective Choice* and the section it indicated has to be set to unassigned.

It is worth noting that the hard constraints in this case cannot be handled by simply recording and checking how many sections each professor has been assigned, because if a professor's preferred choice is assigned, he should give up his less preferred choice in case he already has been assigned enough sections.

6.2.4 Repair-and-stuff-with-dynamic-domains algorithm

After adding maintenance of dynamic domains to *repair-and-stuff* algorithm, we get a complete and efficient *Repair-and-stuff-with-dynamic-domains* algorithm.

Repair-and-stuff-with-dynamic-domains algorithm:

1. Start with a random initial assignment.
2. Initialize the dynamic domains (*Sections' Competing Professors Lists*).
3. Randomly detect a section that is improperly assigned then fix it by reassigning the section to an appropriate professor (with the help of the dynamic domains).
4. Maintain the dynamic domains (*Sections' Competing Professors lists*).
5. Randomly detect a *Needed-and-available Section* then assign the section.
6. Maintain the dynamic domains (*Sections' Competing Professors lists*).
7. Repeat steps 3 to step 6 until appropriate termination conditions are met.

The dynamic domains of the sections are size-reduced and efficiently-maintained. It improves the performance of the *repair-and-stuff* algorithm by tightening the search space. Our experimental results show this algorithm can quickly repair an assignment or generate a good random assignment.

6.3 Virtual-neighborhood Hill-climbing

The solutions found by the *Repair-and-stuff-with-dynamic-domains* algorithm are good. But we desire better solutions. The *Repair-and-stuff-with-dynamic-domains* algorithm is designed to find a *Stuffed Violation-free Assignment*. On top of satisfying constraints, we also need to optimize our assignment, such as maximizing the number of assigned sections. The *repair-and-stuff* algorithm is such an improvement on the *simple repair* algorithm in terms of

the total number of assigned sections. An improved algorithm will be introduced in this section to further improve the *repair-and-stuff-with-dynamic-domains* algorithm.

In this section, we see the assignment found by the *repair-and-stuff-with-dynamic-domains* algorithm as a random starting point. We start from there to look for a better assignment.

The first idea is to use a simple hill-climbing algorithm to find a better assignment. Unfortunately, we soon find that a simple hill-climbing algorithm hardly works because there often are few neighbors or even none with higher or equal evaluation value in the close neighborhood. It is not easy to move from where we already are, because most possible moves may cause violations of rules or decrease in evaluation value. With no special help for escaping local optimum, we are highly likely to get stuck there.

6.3.1 Origin

Hill-climbing (HC) is a well-known local search algorithm. The basic idea is to always move towards a state that is better than the current one (Russell and Norvig, 2002). Hill-climbing explores the neighborhood and moves to a solution with a better evaluation value and repeats exploring and moving until appropriate conditions are met.

Variable Neighborhood Search (VNS) is an explorative local search method proposed by Hansen and Mladenovic (1998). A local optimum in one neighborhood structure may no longer be a local optimum in a different neighborhood structure. The general idea is to dynamically change the neighborhood structures to avoid getting stuck in a single neighborhood structure.

We are not going to apply VNS directly in our case. What we learned from VNS is that we may define a different neighborhood structure that helps to escape from a local optimum.

Another idea is to take more than one step if it is impossible to escape a local optimum by one step. Sometimes, the situation is so bad that there is no better or even equal neighbor in the neighborhood. In that case, one move is impossible to escape local optimum. We need to take a bunch of moves to escape the local optimum.

6.3.2 Virtual neighborhood

Our problem is a CSP-based optimization problem. Our goal is to find an optimal assignment among all *violation-free assignments*.

Our idea is to ignore any non-violation-free assignments and to move from one *violation-free assignment* to another *violation-free assignment* in one action, which may consist of multiple moves. We introduce the following notions to explain the idea.

Move-and-repair: An action that consists of a move (reassigning a section) and a following repair (*Repair-and-stuff-with-dynamic-domains*).

Virtual Neighbor: An assignment's *Virtual Neighbors* are the assignments that can be reached in one *move-and-repair* from the assignment. We do not care what the actual distances are between the assignment and its *Virtual Neighbors*.

We present the following algorithm that uses virtual neighborhood:

Virtual-neighborhood Hill-climbing (HC) algorithm:

1. Start with a random initial assignment that is generated by the *repair-and-stuff-with-dynamic-domains* algorithm.

2. Save the current assignment *A*.

3. Make an appropriate move, for which we have a neighbor assignment *A'*.

4. Repair the neighbor assignment *A'* using the *repair-and-stuff-with-dynamic-domains* algorithm (the repair process does nothing if the assignment contains no conflicts).

5. If the repaired assignment *A''* (*Virtual Neighbor*) is better than the saved one (*A*), accept *A''* as the current assignment *A* (stay); otherwise, replace the *A''* with *A* (go back).

6. Repeat steps 2 to step 5 until appropriate termination conditions are met.

As we explained before, the *repair-and-stuff-with-dynamic-domains* algorithm can function as a random solution generator or a random solution repairer. A move may break rules in the new assignment, and it is followed by a repair process to fix the potential violations. If a move does not cause any violation, the repairer process does nothing. The *repair-and-stuff-with-dynamic-domains* algorithm is nondeterministic in order to avoid cycling in step 4. The Virtual Neighbor can only be evaluated after the algorithm moves there. If the *Virtual Neighbor* has a higher evaluation value, we stay there; otherwise we go back where we were. Because the repair process is nondeterministic, we cannot go back by traveling the same path. An easy solution is to backup the assignment and then restore it if needed. The repair process and the backup-and-restore procedure are the major cost associated with using virtual neighborhood. So choosing an appropriate move is important to the algorithm's performance.

Making an appropriate move:

We have to be careful about how we make the next move. A move is a change in the value of a section, which means reassigning a section.

Immediate Violation: If we assign a section to a professor whose priority is lower than one of the section's *Competing Professors*, then this move causes an *Immediate Violation*.

We should avoid a move that causes an *Immediate Violation* because the new violation will soon be detected and the new assignment will be fixed back to the same old one.

A move that does not cause an *Immediate Violation* may cause other violations in the assignment. For example, a professor lost a section due to a move that does not cause an *Immediate Violation*. Probably, he has an Ineffective Choice turned to an Effective Choice. This change may cause a violation. The point is that a move that does not cause an *Immediate Violation* and its following repair process will lead us to a different *Stuffed Violation-free Assignment*, which is what we expect.

The following selection process is introduced to avoid cycling by choosing a move that does not cause an *Immediate Violation*.

Make an appropriate move:

1. Randomly select an assigned section.

2. Try to find a Competing Professor of the section who has equal priority with the Winner Professor.

3. If such a professor is found, we reassign the section to this professor; otherwise, we repeat steps 1, 2 until appropriate termination conditions are met.

For any assigned section, there is a *Winner Professor* and probably there are also some *Competing Professors* with the same priority. What we are going to do is to find those *Competing Professors* with the same priority, then re-assign the section to one of those

professors. By doing it this way, it will not cause *Immediate Violation* on that section, though it may cause violations on other sections. The point is that we are unlikely to be moved back to the same place by the subsequent repairing process.

6.4 Simulated Annealing

Simulated Annealing (SA) is one of the earliest algorithms that explicitly deal with escaping local optimum. This approach simulates the controlled cooling process in metallurgy that increases the size of a crystal (Kirkpatrick et al., 1983). The process starts at a high temperature and then is slowly cooled. Simulated annealing allows moves that result in solutions with worse evaluation value than the current one in order to escape local optimum. However, the probability of taking such a move gradually decreases while the temperature drops.

Simulated Annealing:

1. Start with a random initial state.
2. Set the initial temperature.
3. Take a random move.
4. If the new state has a better evaluation value, accept it; otherwise, accept it with a probability that decreases while the temperature drops.
5. Update the temperature.
6. Repeat steps 3 to step 5 until appropriate termination conditions are met.

6.4.1 Virtual-neighborhood SA Algorithm

In the previous section, we introduced *Virtual Neighborhood* to escape the local optimum mainly caused by the assignments with violations. In this section, we further improve our algorithm by using a characteristic of Simulated Annealing, which can help us escape from some local optimums in the *Virtual Neighborhood*.

Virtual-neighborhood SA algorithm:

1. Start with a random initial assignment generated by the *repair-and-stuff-with-dynamic-domains* algorithm.

2. Set the initial temperature T to T_0.

3. Save the current assignment A.

4. Make an appropriate move, for which we have a neighbor assignment A'.

5. Repair the neighbor assignment A' using the *repair-and-stuff-with-dynamic-domains* algorithm (the repair process does nothing if the assignment contains no conflicts).

6. If the repaired assignment A'' (*Virtual Neighbor*) is better than the saved one (A), accept A'' as the current assignment A (stay); otherwise, replace the A'' with A (go back) with probability $1 - p(T, A, A')$ and accept A'' as the current assignment A (stay) with probability $p(T, A, A')$.

7. Update the temperature T.

8. Repeat steps 3 to step 7 until appropriate termination conditions are met.

SA is integrated into our *Virtual-neighborhood Hill-climbing* algorithm. Because the repair process in the step 5 may cause a number of moves, we have to save the current assignment in order to get back from a *Virtual Neighbor*. In step 6, when the *Virtual Neighbor* has a lower evaluation value, the probability to accept it is $p(T, A, A')$, and then the probability to reject it is $1 - p(T, A, A')$.

6.5 Vector-oriented Local Search

Simulated Annealing helps us escape from some local optimums in a *Virtual Neighborhood*. In our case, our solution's value is represented as a vector, such as $[V_1, V_2, V_3]$, with which even SA does not work efficiently.

In section 3.3, we presented the Simplified Solution Value as $V_{Simp} = [V_{R5}, V_{Sect}, V_{Pref}]$:

- V_{R5} indicates the number of professors who have been assigned at least one section from one of their first two choices.
- V_{Sect} indicates the total number of assigned sections
- V_{Pref} indicates the total weight of every assigned section relative to the professors' preferences

Suppose we have five assignments with the following *Simplified Solution Values*:

[2, *9*, 2], [2, *8*, 2], [2, *7*, 2], [2, *6*, 2], [*3*, 6, 2].

With Simulated Annealing, the probability of moving along the path of [2, *9*, 2] → (accept with low probability) [2, *8*, 2] → (accept with low probability) [2, *7*, 2] → (accept with low probability) [2, *6*, 2] → (accept) [*3*, 6, 2] is very low because it has to continually move from an assignment with a higher value to an assignment with a lower value before reaching the assignment with the highest value among these five assignments. Thus, we need a special technique to deal with vector value.

6.5.1 Origin

Guided Local Search (GLS) is a general heuristic search technique, which was designed to help local search process to escape from local optimum (Voudouris and Tsang, 1999). When a local search gets stuck in a local optimum, it changes the objective function by adding some penalties, which makes the current local optimum not as good as it was. The changed objective function will hopefully help the local search to escape the local optimum.

Again, we are not going to apply GLS directly since it has its own weaknesses. GLS can escape from several local optimums it encountered in the early stage. However, it may also get stuck in a group of local optimums it visited. What we learned from GLS, however, is that we may change the objective function at an appropriate time.

In a simulated annealing algorithm, the temperature is dropping during the search process. The earlier in the process, the higher the probability that a bigger move may happen. Consider that we have an objective function with controllable precision. In the early stage, the objective function's precision is rough, which makes the landscape smooth. The precision of the objective function increases while the search proceeds until it reaches its full precision. For example, as an analogy, suppose we have an evaluation value V, which is an integer value. In the early stage of the search, we know V is about five hundred; in the middle stage of the search, we know V is about five hundred and forty; in the late stage of the search, we know V is five hundred and forty-two. In the early stage, this is sufficient to know that V is better than 300 or 400, or later that V is better than 510 or 520, etc.

6.5.2 Vector-oriented Local Search

Inspired by GLS's changeable objective function and SA's controlled cooling, we introduce the following algorithm, which is designed to deal with vector value:

Vector-oriented Hill-climbing algorithm:

1. Start with a random initial assignment A.
2. Set the initial precision P of the objective function $f()$ to P_0.
3. Make a move, for which we have a new assignment A'.
4. Compare A and A' with the objective function whose current precision is P. If $f(A', P) \geq f(A, P)$, accept A'.
5. Repeat steps 3, 4 until appropriate termination conditions are met.
6. Increase the precision P of the objective function.
7. Repeat steps 3 to step 6 until appropriate termination conditions are met.

This algorithm is designed to deal with vector values. Increasing the precision is done by ignoring fewer low-components in the vector value. Suppose we have an objective function $f(V, P)$, then we have the following examples:

- $f([5, 4, 2], 1) = [5, 0, 0]$,
- $f([5, 4, 2], 2) = [5, 4, 0]$,
- $f([5, 4, 2], 3) = [5, 4, 2]$.

The goal of this algorithm is to avoid getting stuck in a local optimum too early. In step 4, a skid, which is a move to an assignment with equal evaluation value, is allowed in this algorithm. Skidding is important to this algorithm because moving around increases the possibility of finding a better solution. The low precision of the objective function in the early stage makes the landscape smooth and thus helps quickly find a plateau where good solutions may exist by ignoring trivial differences. This strategy is especially useful in a rugged landscape.

Looking back upon the example we gave at the beginning of this section, suppose we have five assignments of the following *Simplified Solution Value*:

[2, 9, 2], [2, 8, 2], [2, 7, 2], [2, 6, 2], [3, 6, 2].

The moves along the path of [2, 9, 2] → [2, 8, 2] → [2, 7, 2] → [2, 6, 2] → [3, 6, 2] in the early stage become [2, -, -] → [2, -, -] → [2, -, -] → [2, -, -] → [3, -, -], where "-" indicates a value which is ignored (don't care). Thus, the probability of encountering such a series of moves is higher in this algorithm than in any other algorithms presented so far.

6.5.3 Vector-oriented Virtual-neighborhood HC algorithm

By integrating *Vector-oriented Hill-climbing* algorithm into *Virtual-neighborhood Hill-climbing* algorithm, we get the following algorithm:

Vector-oriented Virtual-neighborhood HC algorithm:

1. Start with a random initial assignment A.
2. Set the initial precision P of the objective function $f()$ to P_0.
3. Proceed to a local search by using a *Virtual-neighborhood Hill-Climbing* algorithm until appropriate termination conditions are met.
4. Increase the precision P of the objective function.
5. Repeat steps 3, 4 until appropriate termination conditions are met.

We use the *Virtual-neighborhood HC* algorithm introduced in the previous section to proceed to a local search with rough precision to find a plateau, then increase the precision to eventually find a peak. By using the *vector-oriented* local search technique, we have improved our local search algorithm even further. Now the solutions found by our algorithm are impressively good.

6.6 Summary

In this chapter, we introduced several well-known local search heuristic algorithms, such as Min-conflicts, Hill-climbing, Various Neighborhood Search, Guided Local Search, and Simulated Annealing. What we did is not simply apply those algorithms and combine them together. We also tried to *inspire* ourselves from these algorithms to work out some new ideas suitable for our own problem.

1. We started with the *simple repair* algorithm that is based on min-conflicts. Thanks to our under-constrained rules, this algorithm guarantees a *Violation-free Assignment*.

2. We improved the *simple repair* algorithm to obtain a *repair-and-stuff* algorithm which tries to assign more sections.

3. *Size-reduced and dynamically-maintained domain* was introduced to improve the *repair-and-stuff* algorithm's performance, which resulted in the *repair-and-stuff-with-dynamic-domains* algorithm.

4. The *repair-and-stuff-with-dynamic-domains* algorithm was then used as a random solution generator and a random solution repairer in the following algorithms.

5. *Virtual-neighborhood* was introduced to avoid getting stuck by *non-violation-free assignments*, which resulted in the *virtual-neighborhood hill-climbing* algorithm.

6. Simulated Annealing was used to generally avoid some local optimums in *virtual-neighborhood*, which resulted in the *virtual-neighborhood SA* algorithm.

7. The *vector-oriented local search* was introduced to deal with the ruggedness of vector value. By lowering the objective function's precision in the early stage, it avoids getting stuck too early and thus allows to move or skid towards a

plateau where good solutions may exist. After combining the *vector-oriented local search* with *virtual-neighborhood HC*, we eventually obtained our final algorithm: *vector-oriented virtual-neighborhood HC* algorithm.

In the next chapter, we will compare three algorithms: branch-and-bound, *repair-and-stuff-with-dynamic-domains*, and *vector-oriented virtual-neighborhood HC*. We compare these three algorithms because branch-and-bound can find exact optimal solutions to small scale problems and *repair-and-stuff-with-dynamic-domains* can quickly find a good solution and *vector-oriented virtual-neighborhood HC* is the best local search algorithm we proposed in terms of solution quality.

CHAPTER VII

EXPERIMENTAL RESULTS

In this chapter, we present some experimental data obtained by analyzing the behavior of the algorithms we introduced in the previous chapters. This chapter is divided into two parts; the first is based on some real data from the Département d'informatique, the second is based on randomly generated data. In each part, we compare results obtained by various algorithms. We compare algorithms by measuring how fast they run and how good the resulting solutions are.

Experimental setup

All the algorithms we presented were written in Java and were executed on a PC with Pentium III 866MHz CPU and 512M RAM. The Windows XP operating system was running on the PC. The Java Virtual Machine version 5.0 was used as the Java runtime environment.

Simplified Solution Value

In this chapter, for convenience's sake, we use *Simplified Solution Value* $V_{Simp} = [V_{RS}, V_{Sect}, V_{Pref}]$, which was introduced in section 3.3:

- V_{R5} indicates the number of professors who have been assigned at least one section from one of their first two choices.
- V_{Sect} indicates the total number of assigned sections
- V_{Pref} indicates the total weight of every assigned section relative to the professors' preferences

The assignments obtained by hand or our algorithms often contain zero or few violations. The *Simplified Solution Value* is easier to read. If an assignment contains one or two violations, we add a comment after the *simplified solution value*, such as [*V1, V2, V3*] with one violation of Rule #4.

7.1 Results based on the real data

Our real data comes from the Département d'informatique. We used the data for the courses assignment for the semesters of fall 2004 and winter 2004 to perform some comparisons.

The fall 2004 semester had the following characteristics:

- 34 professors made choices for some sections
- 77 different sections had been chosen by some professors

The winter 2004 semester had the following characteristics:

- 37 professors made choices for some sections
- 75 different sections had been chosen by some professors

Results obtained manually

The results obtained manually, with the help of an Excel worksheet, were the following:

V_{M_Fall} = [30, 54, 174] with one violation of Rule #3.

- 30 professors had been assigned at least one section from one of their first two choices
- in total, 54 section had been assigned
- the total weight of assigned sections was 174 (the bigger, the better)

V_{M_Winter} = [34, 55, 177] with one violation of Rule #7.

Result obtained by our branch-and-bound algorithm:

No result could be obtained by the branch-and-bound algorithm. We terminated the program after it had run for one day without finding any solution. Because the time complexity of this algorithm is exponential, it takes a long time to find an optimal solution to a problem of even normal size.

Result obtained by our local algorithm (*vector-oriented virtual-neighborhood HC*)

The result obtained by our *vector-oriented virtual-neighborhood HC* algorithm was the following:

V_{P_Fall} = [34, 54, 181] and there is no violation of any rule.

Execution Time = 0.481 second

V_{P_Winter} = [36, 55, 186] and there is no violation of any rule.

Execution Time = 0.491 second

Explanation and Analysis

The assignments obtained by hand are surprisingly good, which show people with strong experience can do a really good job. Our branch-and-bound algorithm failed because it takes too long to find a solution. Our local search algorithm found a better assignment for

either semester in less than one second. For the fall 2004 semester, the main difference is on V_{R5} (34 > 30, 54 = 54, 181 > 174). In total, 34 professors made choice for sections and the result obtained by our algorithm (V_{R5}=34) shows all professors have been assigned at least one section from one of his first two choices. Although the two assignments have the same number of assigned sections, they have different meanings. Considering that the assignment obtained by hand has one rule violation and did not assign all professors at least one section from his first two choices, the assignment obtained by our local search algorithm assigned the same amount of sections in a situation where more rules with higher priority are satisfied. Also, by observing V_{Pref} (181 instead of 174), we know that our local search algorithm assigned more sections at the beginning of the professors' choice-lists. For the winter 2004 semester, we are in a similar situation (36 > 34, 55 = 55, 186 > 177).

According to the results we obtained from the real data, our local search algorithm can find a good assignment in a short time as we expected. Obviously, our local search algorithm is able to solve problems of greater scale. In the next section, we present experimental data based on randomly generated problems.

7.2 Results based on randomly generated data

In this section, we first generate randomly some problems, and then we run various algorithms to solve them, and compare the solutions they found.

The randomly generated problems are similar to the real world problem in many respects. The main difference is their size. We use a number of random factors to generate a problem, as explained below.

Explanations of the random factors

The random factors we use are rough approximations of the real data. The key factor is the number of sections. Let N represent the number of sections, which is the size of the problem to generate. The rest of the factors are related to N:

- The number of different courses C is N * 50%;

- The number of professors P is N * 50%;
- The number of professors who need 1 section is P * 30%;
- The number of professors who need 2 sections is P * 60%;
- The number of professors who need 3 sections is P * 10%;
- The number of new professors is P * 20% (Rule #3);
- The number of professors involved in graduate programs is P * 50% (Rule #3);
- The number of course coordinators is P * 40% (Rule #4);
- The number of professors who choose to give the same course again is P * 40% (Rule #6);
- The number of graduate courses is C * 20% (Rule #3 and Rule #7);
- The number of pre-assigned sections is N * 10% (Rule #0);

7.2.1 Small scale problems ($N < 25$)

The purpose of solving small scale problems is that they can be solved by a branch-and-bound algorithm. Since the solution that our branch-and-bound algorithm found is an exact optimal solution, it can thus be used to measure the quality of the solutions found by the local search algorithms.

Branch-and-bound algorithm (B&B)

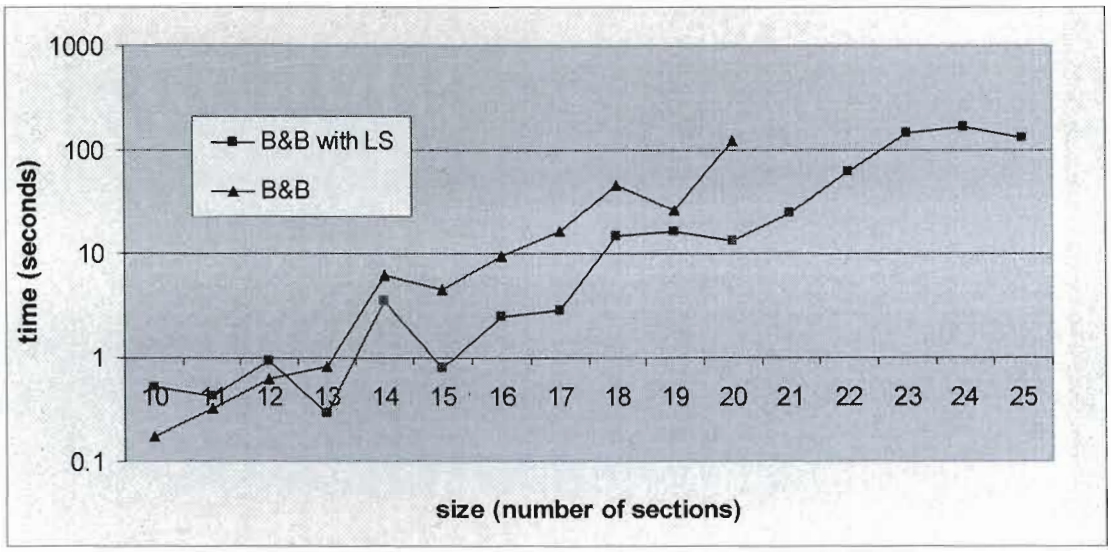


Figure 7.2 Comparison of B&B and B&B with LS in execution time

B&B algorithm's time complexity is exponential. With the help of a local search (LS) algorithm (*repair-and-stuff* algorithm), we obtain a good solution quickly. Then that solution helps us prune a lot of the unnecessary search space. Thus, B&B with LS runs faster than a normal B&B. Unfortunately, it still takes exponential time to find the exact optimal solution due to the nature of B&B algorithm.

Local Search Algorithms

Size	Repair-and-stuff with dynamic domains	Vector-oriented virtual-neighborhood	Branch-and-bound
10	0.01	0.12	0.2
15	0.01	0.09	4.5
20	0.01	0.13	123
25	0.01	0.17	

Table 7.4 Execution time of the three algorithms on small problems (seconds)

Size	Repair-and-stuff with dynamic domains	Vector-oriented virtual-neighborhood	Optimal Solution Value
10	5, 10, 21	5, 10, 23	5, 10, 23
15	7, 13, 29	7, 14, 31	7, 14, 31
20	10, 18, 43	10, 19, 42	10, 19, 42
25	12, 23, 63	12, 24, 62	12, 24, 62

Table 7.5 Solution quality of the three algorithms on small scale problems

As we can see in tables 7.4 and 7.5, the *repair-and-stuff with dynamic domains* algorithm can quickly find a solution that is close to the optimal solution. Meanwhile, the *vector-oriented virtual-neighborhood* algorithm stands a good chance to find the optimal solution though it takes more time than the *repair-and-stuff with dynamic domains* algorithm.

7.2.2 Large scale problems (N > 100)

Because B&B cannot handle large scale problems, we only present the experimental data of our two local search algorithms in this subsection and compare these two algorithms in terms of execution time and solution quality.

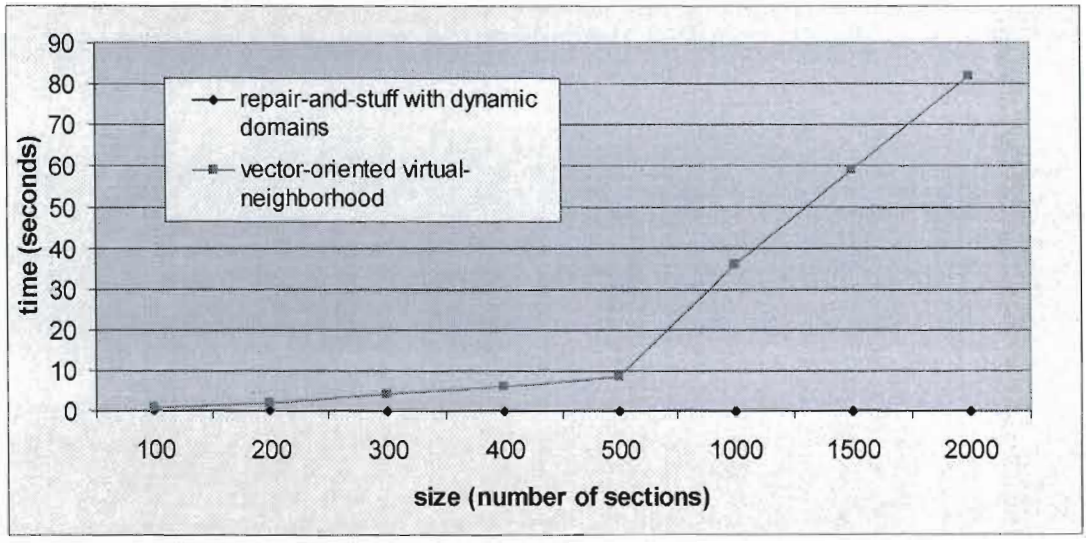


Figure 7.3 Execution time of the two local search algorithms on large scale problems

Note: the time that the *repair_and_stuff with dynamic domains* algorithm takes is too little to be seen clearly in the graph: [size = 500, time = 0.5s], [size = 1000, time = 0.11s], [size = 1500, time = 0.13s], [size = 2000, time = 0.18s].

Size	Repair-and-stuff with dynamic domains	Vector-oriented virtual-neighborhood
100	47, 87, 230	48, 89, 230
200	94, 168, 455	95, 174, 460
300	142, 247, 662	143, 254, 675
400	192, 321, 912	194, 334, 921
500	231, 391, 1092	233, 403, 1124
1000	477, 795, 2250	482, 818, 2229
1500	708, 1202, 3332	715, 1223, 3387
2000	948, 1594, 4544	973, 1646, 4562

Table 7.6 Solution quality of the two local search algorithms on large scale problems

The r*epair-and-stuff with dynamic domains* algorithm runs amazingly fast. It took only 0.18 seconds to solve a problem of size of 2000. The *vector-oriented virtual-neighborhood* algorithm takes much more time than the r*epair-and-stuff with dynamic domains* algorithm. The execution time of both algorithms increases approximately linearly. The improvement made by the *vector-oriented virtual-neighborhood* algorithm does not seem to be "huge". The reason is that the solution found by the *repair-and-stuff with dynamic domains* algorithm is already a good solution and the room for improvement is limited, especially on small scale problems. While the size of the problems grows, the improvement becomes obvious. For example, for the problem of size of 2000, 52 more sections (1646 - 1594) were assigned and the execution time of 82 seconds was still reasonable for such a problem.

To sum up, the *vector-oriented virtual-neighborhood* algorithm we proposed in this thesis does a good job in terms of solution quality and execution time and our goal is achieved — a good solution to a real problem can be found in less than one second. Furthermore, the results also show that our algorithm can find good solutions to large scale problems in a reasonable time.

CONCLUSION

Each semester, the Chair of the Computer Science Department of UQAM must appropriately assign sections to the professors according to their requests. The section assignment must obey the rules from the Département d'informatique. In addition, the section assignment should also be optimized in terms of the total number of assigned sections and the professors' preferences. As we saw, this problem turned out to be a CSP-based optimization problem. Our goal was thus to find an assignment of *good* quality in a reasonable time.

We presented use cases and a class diagram to help understand our problem's context and associated data. Some basic primitive types (Set, Map, Sequence, etc.) were used to better define our problem. The lexicographic ordering was then introduced to help evaluate a possible solution. Some important notions, such as Need-met Point, Effective Choice, Ineffective Choice, Competing Professor, and Winner Professor, were presented to help understand rule violations.

Based on our understanding of the problem, we studied some approaches that had been used to solve similar problems. We chose three of them for further study, which are maximum matching algorithm, branch-and-bound algorithm, and local search algorithm. We first tried to adapt the Munkres assignment algorithm to suit our needs. It turned out that the Munkres algorithm had difficulty dealing with constraints. Then we presented a branch-and-bound algorithm, which was implemented for the purpose of evaluating the other approaches we later proposed.

Our main work was focused on local search algorithms. We started with a *simple repair* algorithm that is based on min-conflicts. Thanks to under-constrained rules, this algorithm guarantees a Violation-free Assignment. We then improved the *simple repair* algorithm to obtain a *repair-and-stuff* algorithm by trying to assign more sections. *Size-*

reduced and dynamically-maintained domain was introduced to improve the *repair-and-stuff* algorithm's performance, which resulted in the *repair-and-stuff-with-dynamic-domains* algorithm that was then used as a random solution generator and a random solution repairer in the algorithms we later proposed. *Virtual-neighborhood* was introduced to avoid getting stuck by non-violation-free assignments. Simulated Annealing was used to generally avoid some local optimums in *Virtual Neighborhood. Vector-oriented local search* was then proposed to deal with the ruggedness of vector value. Lowering the objective function's precision in the early stage avoids getting stuck too early and thus allows to move or skid towards a plateau where good solutions may exist. After combining these various techniques, we eventually obtained our final local search algorithm: *Vector-oriented Virtual-neighborhood HC* algorithm.

The results we obtained from the real data and randomly-generated data show that our algorithm does a good job in terms of solution quality and execution time and our goal is thus achieved — a good solution to a real problem can be found in less than one second. Furthermore, the results also show that our algorithm can find good solutions to large scale problems in a reasonable time.

Due to the lack of time, we did not change the random factors to see how the algorithm's performance would be affected. Also, we performed comparisons on only two semesters' data because the data we collected for the other semesters was either incomplete or inconsistent.

Genetic algorithms, which are inspired by evolutionary biology, have been gaining popularity recently (Goldberg, 1989). A combination of genetic algorithm and local search process would be an interesting topic for future work.

APPENDIX A

DEPARTMENT RULES FOR COURSES ASSIGNMENT

EXTRAIT du procès-verbal de la 5e assemblée régulière du Département d'informatique, de l'Université du Québec à Montréal, tenue mercredi le 31 mars 2004, à la salle PK-5115.

Politique relative à l'attribution des charges d'enseignement

PROPOSITION Nº R-03-161

Sur proposition du Comité exécutif ;

ATTENDU la réduction du nombre de tâches au premier cycle (suite aux baisses de clientèle) qui entraîne un plus grand nombre de conflits pour l'attribution de certains cours ;

ATTENDU l'augmentation du nombre de tâches aux cycles supérieurs ;

ATTENDU l'importance de l'accréditation de nos programmes de baccalauréat et les critères des différents organismes d'accréditation ;

ATTENDU l'engagement du Département et des directions de programmes à donner suite aux évaluations d'enseignement, et ce à tous les cycles ;

ATTENDU la nécessité de disposer de règles d'attribution claires, applicables sans ambiguïté ;

Il est proposé que l'attribution des charges d'enseignement pour les différents groupes-cours soit faite en

fonction de la politique exprimée par les règles suivantes et, pour les règles 2-7, appliquées dans l'ordre indiqué :

Choix des professeurs :

1. Chaque professeur doit normalement soumettre un nombre minimum de choix de groupes-cours spécifié comme suit:

   - Pour enseigner 1 cours : 3 choix
   - Pour enseigner 2 cours : 5 choix
   - Pour enseigner 3 cours : 7 choix

Règles relatives à la qualité de l'enseignement :

2. Sur recommandation du comité de soutien pédagogique, un professeur qui aurait reçu à deux reprises pour deux ou trois prestations successives d'un cours donné une lettre indiquant une évaluation insatisfaisante pour ce cours pourra se voir refuser l'attribution de ce cours pour une période de six sessions suivant la dernière attribution.

3. Dans le cas de cours d'études avancées, une priorité sera accordée aux nouveaux professeurs et aux professeurs activement engagés dans l'encadrement d'étudiants du programme commanditaire principal du cours et l'attribution sera faite en consultation avec le directeur du programme commanditaire (article 10.24 de la convention collective).

Règles relatives à l'attribution selon le choix des professeurs :

4. En conformité avec la politique départementale qui veut qu'un coordonnateur donne le cours coordonné au moins une fois par an, le professeur coordonnateur d'un cours à groupes multiples a priorité pour le choix de son groupe ou pour le choix de son premier

groupe s'il demande plus d'un groupe du même cours, toutefois ce groupe devra faire partie de ses deux premiers choix.

5. Tout professeur devrait se voir attribuer au moins un de ses deux premiers choix.

6. Un professeur qui donne un cours pour une première fois a priorité pour redonner ce cours deux autres fois subséquemment, et perd ensuite sa priorité. Un professeur qui donne de nouveau un cours qu'il a déjà donné dans le passé a priorité pour le redonner une autre fois subséquemment, mais perd ensuite sa priorité.

7. Un professeur qui se voit attribuer un cours aux études avancées n'a pas de priorité pour d'autres cours aux études avancées.

Adoptée à l'unanimité

APPENDIX B

TÂCHES D'ENSEIGNEMENT

UQÀM
Université du Québec à Montréal

TÂCHES D'ENSEIGNEMENT
ÉTÉ 2005
Département d'informatique

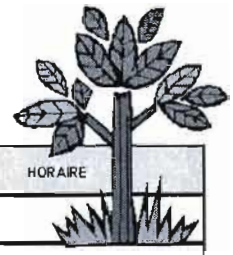
IDENTIFICATION

Nom : _____ Prénom : _____

S.v.p. nous indiquer :

- 3 choix pour 1 cours, 5 choix pour 2 cours ou 7 choix pour 3 cours ,
- par un astérisque les cours d'études avancées ayant fait l'objet d'entente avec le directeur de programme ;
- s'il y a conflit d'horaire parmi les groupes-cours que vous demandez ;
- **si vous bénéficiez d'un dégrèvement, s.v.p. l'indiquez-le et spécifiez-en la nature ;**
- si vous souhaitez utiliser une réserve ou contracter une dette indiquez-le ;
- **si vous dispensez des groupes-cours dans d'autres départements ou facultés vous devez l'indiquer et spécifier leurs horaires.**

Combien de groupes-cours vous souhaitez dispenser ☐



VOS CHOIX DE COURS

PRIORITÉ	GROUPE-COURS DÉGRÈVEMENT	HORAIRE	HORAIRE

Remarques : _____

_____

La date retenue pour transmettre vos choix est fixée au 14 janvier 2005
Merci de votre collaboration !

APPENDIX C

USE CASE

Attribution des charges d'enseignement

1. Cas d'utilisation sommaire

Cas 1 – Recevoir les demandes des professeurs et assigner les groupes-cours aux professeurs (Entreprise & Nuage)

Portée: Université

Niveau: Sommaire

Acteurs: Professeur (P), Directeur de département (DDD), Directeur de programme (DDP), Agente d'administration (ADA)

Préconditions: Les renseignements de base des profs sont prêts et les règles sont mises.

Scénario nominal:

(Traiter des informations pour les trimestres précédents)

1. ADA indique les cours assignés aux professeurs sur les trimestres précédents.
2. DDP indique les avertissements pour mauvaises évaluations sur les trimestres précédents.

(Définir les informations pour le trimestre courant)

3. DDP indique le niveau d'implication des professeurs du trimestre courant.
4. ADA définit les groupes-cours du trimestre courant.
5. ADA indique les coordonnateurs des cours du trimestre courant.

(Faire la demande)

6. P indique les choix des cours.
7. DDD vérifie les choix des professeurs.

(Faire l'assignation)

8. DDD pré-assigne des cours à des professeurs.
9. DDD effectue l'assignation des cours.

2. Spécification des cas d'utilisation

Cas 2 – Indiquer les cours assignés aux professeurs sur les trimestres précédents (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Agent d'administration (ADA)

Préconditions: Le fichier externe qui contient des informations sur les cours assignés est prêt.

Scénario nominal:

1. L'agente s'identifie.
2. Le système authentifie l'agente.
3. L'agente spécifie le fichier indiquant les tâches des trimestres précédents.
4. Le système supprime les informations sur les choix des trimestres précédents puis ajoute les informations sur les cours passés.

(Il suppose que beaucoup de données vient de l'extérieur dans la forme de fichier externe.)

Cas 3 – Indiquer les avertissements pour mauvaise évaluation sur les trimestres précédents (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Directeur de programme (DDP)

Préconditions: Le fichier externe qui contient des informations sur les mauvaises évaluations est prêt.

Scénario nominal:

1. Le directeur de programme s'identifie.
2. Le système authentifie le directeur de programme.
3. L'agente spécifie le fichier indiquant les avertissements des trimestres précédents.
4. Le système ajoute les informations sur les cours passés.

Cas 4 – Indiquer le niveau d'implication des professeurs du trimestre courant (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Directeur de programme (DDP)

Préconditions: Le fichier externe qui contient des informations sur les niveaux d'implication est prêt.

Scénario nominal:

1. Le directeur de programme s'identifie.

2. Le système authentifie le directeur de programme.
3. L'agente spécifie le fichier indiquant le niveau d'implication des professeurs.
4. Le système supprime les informations sur le degré engagement précédent puis ajoute les informations sur le degré engagement.

Cas 5 – Définir les groupes-cours du trimestre courant (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Agent d'administration (ADA)

Préconditions: Le fichier externe qui contient des informations sur les groupes-cours du trimestre courant est prêt.

Scénario nominal:

1. L'agente s'identifie.
2. Le système authentifie l'agente.
3. L'agente spécifie le fichier indiquant les groupes-cours du trimestre courant.
4. Le système ajoute les informations sur les groupes-cours commandés.

Cas 6 – Indiquer les coordonnateurs des cours du trimestre courant (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Agent d'administration (ADA)

Préconditions: Les groupes-cours sont prêts et le fichier externe qui contient des informations sur les coordonnateurs est prêt.

Scénario nominal:

1. L'agente s'identifie.
2. Le système authentifie l'agente.
3. L'agente spécifie le fichier indiquant les coordonnateurs des cours du trimestre courant.
4. Le système ajoute les informations sur les cours courants.

Cas 7 – Indiquer les choix de cours (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Professeur (P)

Préconditions: Les groupes-cours sont prêts.

Scénario nominal:

1. Le professeur s'identifie.
2. Le système authentifie le professeur.

3. Le professeur indiquer combien de groupes-cours qui souhaite dispenser, son dégrèvement, et sa réserve-dette qu'il veut utiliser.
4. Le système ajoute les informations sur les tâches.
5. Le professeur indiquer ses choix de groupes-cours, et les conflits parmi ses choix si nécessaire.
6. Le système ajoute les informations sur les choix.

Cas 8 – Vérifier les choix des professeurs (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Directeur de département (DDD)

Préconditions: Les choix des professeurs sont prêts.

Scénario nominal:

1. Le directeur de département s'identifie.
2. Le système authentifie le directeur de département.
3. Le système affiche les informations sur les choix, les réserves-dettes, et les dégrèvements.
4. Le directeur de département assure que les informations sont vraies.

Cas 9 – Pré-assigner des cours à des professeurs (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Directeur de département (DDD)

Préconditions: La vérification de choix des professeurs a réussi.

Scénario nominal:

1. Le directeur de département s'identifie.
2. Le système authentifie le directeur de département.
3. Le directeur de département pré-assigne des cours à des professeurs.
4. Le système modifie les informations sur les choix.

Cas 10 – Effectuer l'assignation des cours (Système & Mer)

Portée: SYS

Niveau: Usager

Acteurs: Directeur de département (DDD)

Préconditions: Pré-assignation est finie.

Scénario nominal:

1. Le directeur de département s'identifie.
2. Le système authentifie le directeur de département.

3. (Facultatif) Le directeur de département ajuste des paramètres.
4. Le directeur de département effectue l'algorithme de l'assignation.
5. Le système donne des solutions.
6. Le directeur de département choisit une des solutions.
7. Le directeur de département choisit de publier la décision.
8. Le système crée le fichier qui contient la décision puis l'envoyer à l'agente.

APPENDIX D

EXCERPT FROM THE JAVA SOURCE CODE

The following code is for the algorithms we proposed (branch-and-bound, *repair-and-stuff with dynamic domains*, *vector-oriented virtual-neighborhood*, etc.) and the calculation of rule violations.

```
package algorithm;

import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;

import rules.Value;
import type.*;

public class Branch {

   protected Sections sections;
   protected Profs profs;
   protected ProfChoices profChoices;
   protected ProfNeeds profNeeds;
   protected Assignment preAssigned;

   protected SectionChoices sectionChoices;
   protected Object [] arrSections;
   protected int count;

   private Assignment assignment;

   protected Evaluator ev;
   protected Value bestValue;
   protected Map<Prof, Integer> numCoursesAssigned;

   Comparator comparator;
```

```
public Branch( Sections sections, Profs profs,
        ProfChoices profChoices, ProfNeeds profNeeds,
        Assignment preAssigned, Comparator comparator) {
   this.sections = sections;
   this.profs = profs;
   this.profChoices = profChoices;
   this.profNeeds = profNeeds;
   this.preAssigned = preAssigned;
   this.comparator = comparator;
   this.ev = new Evaluator( sections, profs,
            profChoices, profNeeds,
            preAssigned, new ProfComparator());
}

protected SectionChoices profChoicesToSectionChoices(ProfChoices profChoices) {
   SectionChoices sectionChoices = new SectionChoices();
   for( Section section : sections) {
      sectionChoices.put( section, new Profs());
   }
   for( Prof prof : profs) {
      Iterator<Section> choices = profChoices.get(prof).iterator();
      while( choices.hasNext()) {
         Section section = choices.next();
         sectionChoices.get(section).add(prof);
      }
   }
   for( Section section : sections) {
      section.setNumber( sectionChoices.get(section).size());
   }
   return sectionChoices;
}

public void performBranchAndBoundExploration( Assignment assignment) {
   sectionChoices = profChoicesToSectionChoices(profChoices);
   arrSections = sectionChoices.keySet().toArray();
   Arrays.sort( arrSections);
   numCoursesAssigned = new LinkedHashMap<Prof, Integer>();

   this.assignment = new Assignment();
   bestValue = ev.evaluate( assignment);
   count = 0;
   exploreSolutionSpace( this.assignment, 0, sectionChoices, arrSections);
   System.out.println("count: " + count);

   System.out.println( bestValue);
}

public void exploreSolutionSpace(Assignment assignment, int nextSect,
                        SectionChoices sectionChoices,
                        Object [] arrSections) {
   count++;
   if( nextSect == arrSections.length) {
      Value v = ev.evaluate( assignment);
      if( v.compareTo(bestValue) > 0) {
         bestValue = v;
      }
      return;
   } else {
      if( estimate( assignment, nextSect + 1, sectionChoices,
            arrSections).compareTo( bestValue) >= 0) {
         exploreSolutionSpace( assignment, nextSect + 1, sectionChoices, arrSections);
      }
      Profs profs = sectionChoices.get( arrSections[nextSect]);
      for( Prof prof : profs) {
         if( canBeAssigned( (Section)arrSections[nextSect], prof)) {
            assign( assignment, (Section)arrSections[nextSect], prof);
            if( estimate( assignment, nextSect + 1, sectionChoices,
                  arrSections).compareTo( bestValue) >= 0) {
               exploreSolutionSpace( assignment, nextSect + 1, sectionChoices, arrSections);
```

```
            }
            deassign( assignment, (Section)arrSections[nextSect], prof);
          }
        }
      }
    }

    public void assign( Assignment a, Section s, Prof p) {
      a.put(s, p);
      if( numCoursesAssigned.containsKey( p)) {
        numCoursesAssigned.put( p, numCoursesAssigned.get( p) + 1);
      } else {
        numCoursesAssigned.put( p, 1);
      }
    }

    public void deassign( Assignment a, Section s, Prof p )
    {
      a.remove(s);
      int num = numCoursesAssigned.get( p);
      numCoursesAssigned.put( p, num - 1);
      assert num >= 1;
    }

    protected Value estimate( Assignment assignment, int nextSect, SectionChoices sectionChoices,
                        Object [] arrSections){
      Value v = new Value(bestValue);
      v.match = assignment.size() + arrSections.length - nextSect;
      return v;
    }

    protected boolean canBeAssigned( Section s, Prof p) {
      if( numCoursesAssigned.get( p) == null) {
        return true;
      }
      if( numCoursesAssigned.get( p) < profNeeds.get( p)) {
        return true;
      } else {
        return false;
      }
    }

}

package algorithm;

import rules.Value;
import type.*;

public interface Comparator {

    int compare(Prof prof1, Prof prof2, Section section, Value value,
            ProfChoices profChoices, ProfNeeds profNeeds,
            Assignment preAssigned, Assignment assignment);

}
package algorithm;

import java.util.Iterator;

import type.*;
import rules.*;

public class Evaluator {

    protected Sections sections;
    protected Profs profs;
    protected ProfChoices profChoices;
    protected ProfNeeds profNeeds;
```

```
protected Assignment preAssigned;

private Assignment assignment;
private WaitingLists waitingLists;

Comparator comparator;

public Evaluator( Sections sections, Profs profs,
        ProfChoices profChoices, ProfNeeds profNeeds,
        Assignment preAssigned, Comparator comparator) {
   this.sections = sections;
   this.profs = profs;
   this.profChoices = profChoices;
   this.profNeeds = profNeeds;
   this.preAssigned = preAssigned;
   this.comparator = comparator;
}

protected void updateWaitingList(Prof prof) {
   int numSectionsAssigned = 0;
   Iterator<Section> choices = profChoices.get(prof).iterator();
   while( choices.hasNext()) {
      Section section = choices.next();
      if( prof.equals( assignment.get(section))) {
         if( numSectionsAssigned < profNeeds.get(prof)) {
            numSectionsAssigned += 1;
            waitingLists.get(section).remove(prof);
         } else {
            assignment.remove(section);
         }
      } else {
         if( numSectionsAssigned < profNeeds.get(prof)) {
            waitingLists.get(section).add(prof);
         } else {
            waitingLists.get(section).remove(prof);
         }
      }
   }
}

public Value evaluate( Assignment assignment) {
   this.assignment = assignment;

   waitingLists = new WaitingLists();
   for( Section section : sections) {
      waitingLists.put( section, new Profs());
   }
   for( Prof prof : profs) {
      updateWaitingList( prof);
   }

   Value v = new Value();
   for( Section section : sections) {
      Prof prof = assignment.get( section);
      if( prof != null) {
         for( Prof profWaiting : waitingLists.get( section)) {
            comparator.compare(prof, profWaiting, section, v,
                  profChoices, profNeeds, preAssigned, assignment);
         }
         v.match++;    // number of total matching
         v.pref -= profChoices.get(prof).indexOf(section);
         if( profChoices.get(prof).get(0).equals(section)    // first choice
            || ( profChoices.get(prof).size() > 1 && profChoices.get(prof).get(1).equals(section)
            && !prof.equals(assignment.get(profChoices.get(prof).get(0))))){
               v.firstTwo++;    // number of profs who have been assigned at least one course
         }
      }
   }
```

```
        return v;
    }

}

package algorithm;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import rules.Level;
import rules.Value;
import type.*;

public class Improver {

    protected Sections sections;
    protected Profs profs;
    protected ProfChoices profChoices;
    protected ProfNeeds profNeeds;
    protected Assignment preAssigned;

    private Assignment assignment;
    private WaitingLists waitingLists;

    Comparator comparator;

    public Improver( Sections sections, Profs profs,
            ProfChoices profChoices, ProfNeeds profNeeds,
            Assignment preAssigned, Comparator comparator) {
        this.sections = sections;
        this.profs = profs;
        this.profChoices = profChoices;
        this.profNeeds = profNeeds;
        this.preAssigned = preAssigned;
        this.comparator = comparator;
    }

    protected void updateWaitingList(Prof prof) {
        int numSectionsAssigned = 0;
        Iterator<Section> choices = profChoices.get(prof).iterator();
        while( choices.hasNext()) {
            Section section = choices.next();
            if( prof.equals( assignment.get(section))) {
                if( numSectionsAssigned < profNeeds.get(prof)) {
                    numSectionsAssigned += 1;
                    waitingLists.get(section).remove(prof);
                } else {
                    assignment.remove(section);
                }
            } else {
                if( numSectionsAssigned < profNeeds.get(prof)) {
                    waitingLists.get(section).add(prof);
                } else {
                    waitingLists.get(section).remove(prof);
                }
            }
        }
    }

    protected void assign(Prof prof, Section section) {
        assignment.put(section, prof);
        updateWaitingList(prof);
    }

    protected void cancel(Prof prof, Section section) {
        assignment.remove(section);
```

```
    updateWaitingList(prof);
}

protected void move(Prof prof1, Prof prof2, Section section) {
    cancel(prof1, section);
    assign(prof2, section);
}

protected Prof findBestProf(Prof profHi, Profs waitingProfs, Section section) {
    Prof profCur = profHi;
    List<Prof> listProfs = new ArrayList<Prof>(waitingProfs);
    Collections.shuffle( listProfs);
    if( profCur == null) {
        profCur = listProfs.get(0);
    }
    for( Prof prof : listProfs) {
        if( comparator.compare(profCur, prof, section, new Value(),
                profChoices, profNeeds, preAssigned, assignment) < 0) {
            profCur = prof;
        }
    }
    return profCur;
}

protected Prof findAnyProf( Profs waitingProfs, Section section) {
    if( waitingProfs.isEmpty()) {
        return null;
    }
    List<Prof> listProfs = new ArrayList<Prof>(waitingProfs);
    Collections.shuffle( listProfs);
    return listProfs.get(0);
}

public void improve(Assignment assignment) {
    this.assignment = assignment;

    waitingLists = new WaitingLists();
    for( Section section : sections) {
        waitingLists.put( section, new Profs());
    }
    for( Prof prof : profs) {
        updateWaitingList( prof);
    }

    Prof profCur, profHi;
    boolean perfect = false;
    while( !perfect) {
        perfect = true;
        ArrayList<Section> listSections = new ArrayList<Section>(sections);
        Collections.shuffle( listSections);
        for( Section section : listSections) {
            if( !waitingLists.get(section).isEmpty()) {
                if( assignment.containsKey(section)) {
                    profCur = assignment.get(section);
                    profHi = findBestProf(profCur, waitingLists.get(section), section);
                    if( !profCur.equals(profHi)) {
                        move( profCur, profHi, section);
                        perfect = false;
                    }
                } else {
                    profHi = findBestProf(null, waitingLists.get(section), section);
                    assign( profHi, section);
                    perfect = false;
                }
            }
        }
    }
    assert checkImprovePostCondition();
}
```

```
// try all possible equal moves
public void localSearch(Assignment assignment) {
   this.assignment = assignment;

   // construct waitingList for all sections
   waitingLists = new WaitingLists();
   for( Section section : sections) {
      waitingLists.put( section, new Profs());
   }
   for( Prof prof : profs) {
      updateWaitingList( prof);
   }

   Prof profCur, profHi;
   for( int attempt = 0; attempt < MAX_ATTEMPT; attempt++) {
      Assignment copy = (Assignment)assignment.clone();
      ArrayList<Section> listSections = new ArrayList<Section>(sections);
      Collections.shuffle( listSections);
      int steps = 1;
      for( Section section : listSections) {
         if( !waitingLists.get(section).isEmpty()) {
            if( assignment.containsKey(section)) {
               profCur = assignment.get(section);
               profHi = findAnotherBestProf(profCur, waitingLists.get(section), section);
               if( !profCur.equals(profHi)) {
                  move( profCur, profHi, section);
                  steps -= 1;
                  if( steps == 0) {
                     break;
                  }
               }
            }
         }
      }
      improve( assignment);

      Evaluator ev = new Evaluator(sections, profs, profChoices,
            profNeeds, preAssigned, new ProfComparator());
      Value vOld = ev.evaluate(copy);
      System.out.println("valueBefore: " + vOld);
      Value vNew = ev.evaluate(assignment);
      System.out.println("valueAfter: " + vNew);
      int res = vNew.compareTo_Matches(vOld);
      if( res < 0) {
         assignment.clear();
         assignment.putAll(copy);
         System.out.println("moved back");
      } else if( res > 0) {
         attempt = 0;
         System.out.println("moved forward");
      } else {
         System.out.println("moved around");
      }
   }

   System.out.println("assignment: " + assignment.size());
}

// try all possible equal moves
public void localSearch_Level(Assignment assignment) {
   this.assignment = assignment;

   // construct waitingList for all sections
   waitingLists = new WaitingLists();
   for( Section section : sections) {
      waitingLists.put( section, new Profs());
   }
   for( Prof prof : profs) {
```

```
            updateWaitingList( prof);
        }

        int maxAttempts = 39;
        Prof profCur, profHi;
        for( Level level : Value.listLevel) {
            for( int attempt = 0; attempt < maxAttempts; attempt++) {
                Assignment copy = (Assignment)assignment.clone();
                ArrayList<Section> listSections = new ArrayList<Section>(sections);
                Collections.shuffle( listSections);
                int steps = 1;
                for( Section section : listSections) {
                    if( !waitingLists.get(section).isEmpty()) {
                        if( assignment.containsKey(section)) {
                            profCur = assignment.get(section);
                            profHi = findAnotherBestProf(profCur, waitingLists.get(section), section);
                            if( !profCur.equals(profHi)) {
                                move( profCur, profHi, section);
                                steps -= 1;
                                if( steps == 0) {
                                    break;
                                }
                            }
                        }
                    }
                }
                improve( assignment);

                Evaluator ev = new Evaluator(sections, profs, profChoices,
                        profNeeds, preAssigned, new ProfComparator());
                Value vOld = ev.evaluate(copy);
                Value vNew = ev.evaluate(assignment);
                int res = vNew.compareTo(vOld, level);
                if( res < 0) {
                    assignment.clear();
                    assignment.putAll(copy);
                } else if( res > 0) {
                    attempt = 0;
                }
            }
        }

        System.out.println("assignment: " + assignment.size());
    }

    protected Prof findAnotherBestProf(Prof profHi, Profs waitingProfs, Section section) {
        Prof profCur = profHi;
        for( Prof prof : waitingProfs) {
            if( comparator.compare(profCur, prof, section, new Value(),
                    profChoices, profNeeds, preAssigned, assignment) == 0) {
                profCur = prof;
                break;
            }
        }
        return profCur;
    }

}

package algorithm;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import type.*;
import rules.*;

public class ProfComparator implements Comparator{
```

```
   private List<Rule> rules;

   public ProfComparator() {
      rules = new ArrayList<Rule>();

      // the order of rules can be changed if needed
      // first added, first considered
      rules.add(new RulePreAssigned());
      rules.add(new RuleFirstTwo());
      rules.add(new RuleOneGraduate());
      rules.add(new RuleNewEngaged());
      rules.add(new RuleCoordinator());
      rules.add(new RuleRegiving());
   }

   public int compare(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      int result = 0;
      Iterator<Rule> it = rules.iterator();
      while( it.hasNext()) {
         Rule rule = it.next();    // get next rule
         result = rule.Check( prof1, prof2, section, value,
               profChoices, profNeeds, preAssigned, assignment);
         if( result != 0) {
            result *= rules.indexOf(rule) + 1;
            break;    // stop comparing
         }
      }
      return result;
   }

}

package rules;

import type.*;

public interface Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment);

}
package rules;

import type.*;

public class RuleConflict implements Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      // profs.regiving has been pre-processed for current semester
      boolean b1 = (prof1.regiving != null && prof1.regiving.contains( section.getCourse()));
      boolean b2 = (prof2.regiving != null && prof2.regiving.contains( section.getCourse()));
      if( b1 != b2) {
         if( b2) {    // p1 < p2, breaking the rule
            value.regiving--;
            return -1;
         } else {    // p1 > p2, OK, return
            return 1;
         }
      } else {    // equal, continue to compare
         return 0;
```

```
    }
  }

}
package rules;

import type.*;

public class RuleCoordinator implements Rule {

  public int Check(Prof prof1, Prof prof2, Section section, Value value,
      ProfChoices profChoices, ProfNeeds profNeeds,
      Assignment preAssigned, Assignment assignment) {

    boolean b1 = (prof1.crdnt != null && prof1.crdnt.contains(section));
    if( b1) {   // coordinator
      // the coordinated course must be in first two choices
      // and only the first choice is valid for same course, different groups
      Section choice1 = profChoices.get(prof1).get(0);
      Section choice2 = profChoices.get(prof1).get(1);
      // check first choice
      if( !(section.equals( choice1))) {   // if it's not the first choice
        // check seconde choice, if it exists
        if( profChoices.get(prof1).size() > 1 && section.equals( choice2)) {
            // check if the first and the second are same course
            // and check if the first is assigned
            if( choice1.getCourse().equals( choice2.getCourse()) &&
                assignment.get( choice1).equals(prof1)) {
              // already been assigned a coordinated section for the course
                b1 = false;
             }   // different courses, valid
        } else {
            b1 = false;   // not one of first two, invalid
        }
      }   // if it's the first choice, valid, continue
    }

    boolean b2 = (prof2.crdnt != null && prof2.crdnt.contains(section));
    if( b2) {   // coordinator
      // the coordinated course must be in first two choices
      // and only the first choice is valid for same course, different groups
      Section choice1 = profChoices.get(prof2).get(0);
      Section choice2 = profChoices.get(prof2).get(1);
      // check first choice
      if( !(section.equals( choice1))) {   // if it's not the first choice
        // check seconde choice, if it exists
        if( profChoices.get(prof2).size() > 1 && section.equals( choice2)) {
            // check if the first and the second are same course
            // and check if the first is assigned
            if( choice1.getCourse().equals( choice2.getCourse()) &&
                assignment.get( choice1).equals(prof2)) {
                // already been assigned a coordinated section for the course
                b2 = false;
             }   // different courses, valid
        } else {
            b2 = false;   // not one of first two, invalid
        }
      }   // if it's the first choice, valid, continue
    }

    if( b1 != b2) {
      if( b2) {   // p1 < p2, breaking the rule
        value.coordinator--;
        return -1;
      } else {   // p1 > p2, OK, return
        return 1;
      }
    } else {   // equal, continue to compare
      return 0;
```

```
    }
  }

}

package rules;

import type.*;

public class RuleFirstTwo implements Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      boolean s = false;
      // true means prof1 still has one assigned in first two choices by loosing the section
      boolean b1 = false;
      {
         int index = profChoices.get(prof1).indexOf(section);
         Section choice1 = profChoices.get(prof1).get(0);
         if( index > 1) {    // not first 2, ok to loose
            b1 = true;
         } else if( prof1.equals(assignment.get(choice1))) {
            if( profChoices.get(prof1).size() > 1) {
               if( prof1.equals(assignment.get(profChoices.get(prof1).get(1)))) {
                  b1 = true;    // both first and second are assigned, ok to loose one
               }
               if( !assignment.containsKey(profChoices.get(prof1).get(1))) {
                  b1 = true;    // possibly, he will get the second choice, ok to loose one
                  s = true;
               }
            }
         }
      }

      // true means prof2 will change from 0 in first2 to 1 in first2 by getting the section
      boolean b2 = false;
      {
         int index = profChoices.get(prof2).indexOf(section);
         if( index < 2) {    // first 2, good to get one
            b2 = true;
            Section choice1 = profChoices.get(prof2).get(0);
            if( prof2.equals(assignment.get(choice1))) {
               b2 = false;
            }
            if( profChoices.get(prof2).size() > 1) {
               Section choice2 = profChoices.get(prof2).get(1);
               if( prof2.equals(assignment.get(choice2))) {
                  b2 = false;
               }
            }
         }
      }

      if( b1 == b2) {
         if( b2) {    // p1 < p2, breaking the rule
            value.firstTwo_--;
            if( !s) {
               return -1;
            }
         } else {    // p1 > p2, OK, return
            return 1;
         }
      } else {    // equal, continue to compare
         return 0;
      }
   }
```

```
}

package rules;

import type.*;

public class RuleNewEngaged implements Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      // if section is advanced
      boolean c = section.getCourse().isAdvanced();

      // if they are new or engaged profs?
      boolean b1 = prof1.newProf || (prof1.engaged != null &&
            prof1.engaged.contains(section.getCourse()));
      boolean b2 = prof2.newProf || (prof2.engaged != null &&
            prof2.engaged.contains(section.getCourse()));
      if( c && b1 != b2) {    // if it's graduate course
         if( b2) {    // p1 < p2, breaking the rule
            value.newEngaged--;
            return -1;
         } else {    // p1 > p2, OK, return
            return 1;
         }
      } else {    // not a gruadate course or equal priority, continue to compare
         return 0;
      }
   }

}

package rules;

import java.util.Iterator;

import type.*;

public class RuleOneGraduate implements Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      boolean c = section.getCourse().isAdvanced();

      if( !c) {
         return 0;    // not a graduate course
      }

      // true means prof already got an graduate course
      boolean b1 = false;
      Iterator<Section> choices1 = profChoices.get(prof1).iterator();
      while( choices1.hasNext()) {
         Section tempSection = choices1.next();
         if( tempSection.equals(section)) {
            break;
         }
         if( prof1.equals(assignment.get(tempSection)) &&
               tempSection.getCourse().isAdvanced() ) {
            b1 = true;
            break;
         }
      }

      // true means prof already got an graduate course
      boolean b2 = false;
```

```
      Iterator<Section> choices2 = profChoices.get(prof2).iterator();
      while( choices2.hasNext()) {
         Section tempSection = choices2.next();
         if( tempSection.equals(section)) {
            break;    // break: if we only check choices before section
                      // continue: if we check all choices but section
                      // if depends on how we interpret the rule
         }
         if( prof2.equals(assignment.get(tempSection)) &&
               tempSection.getCourse().isAdvanced() ) {
            b2 = true;
            break;
         }
      }

      if( b1 != b2) {    // a graduate course
         if( b1) {    // p1 < p2, breaking the rule
            value.oneGraduate--;
            return -1;
         } else {    // p1 > p2, OK, return
            return 1;
         }
      } else {    // equal, continue to compare
         return 0;
      }
   }

}

package rules;

import type.*;

public class RulePreAssigned implements Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      boolean b1 = prof1.equals( preAssigned.get(section));
      boolean b2 = prof2.equals( preAssigned.get(section));
      if( b1 != b2) {
         if( b2) {    // p1 < p2, breaking the rule
            value.preAssigned--;
            return -1;
         } else {    // p1 > p2, OK, return
            return 1;
         }
      } else {    // equal, continue to compare
         return 0;    // they are equal
      }
   }

}

package rules;

import type.*;

public class RuleRegiving implements Rule {

   public int Check(Prof prof1, Prof prof2, Section section, Value value,
         ProfChoices profChoices, ProfNeeds profNeeds,
         Assignment preAssigned, Assignment assignment) {

      // profs.regiving has been pre-processed for current semester
      boolean b1 = (prof1.regiving != null && prof1.regiving.contains( section.getCourse()));
      boolean b2 = (prof2.regiving != null && prof2.regiving.contains( section.getCourse()));
      if( b1 != b2) {
```

```
            if( b2) {    // p1 < p2, breaking the rule
               value.regiving--;
               return -1;
            } else {    // p1 > p2, OK, return
               return 1;
            }
         } else {    // equal, continue to compare
            return 0;
         }
      }

}
package rules;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Value implements Comparable {

   public int preAssigned;    // pre-assigned courses shouldn't be assigned to other profs
   public int newEngaged;    // new and engaged profs; -3 means this rule has been broken 3 times
   public int coordinator;    // coordinators; -3 means this rule has been broken 3 times
   public int firstTwo;    // the number of pros who got at least one course of their first two choices
         // it's always a non-negative number
   public int firstTwo_;    // approximate equivalent of rule5
   public int regiving; // given the course in the past; it's awalys a non-positive number
   public int oneGraduate; // assigned an advanced course; it's awalys a non-positive number

   public int match;    // the number of assigned courses
   public int pref;

   public static final List<Level> listLevel = new ArrayList<Level>();

   static {
      listLevel.add( new LevelFIrstTwo());
      listLevel.add( new LevelMatches());
      listLevel.add( new LevelPrefs());
   }


   public Value() {
      preAssigned = 0;
      newEngaged = 0;
      coordinator = 0;
      firstTwo = 0;
      firstTwo_ = 0;
      regiving = 0;
      oneGraduate = 0;
      match = 0;
      pref = 0;
   }

   public Value( Value v) {
      preAssigned = v.preAssigned;
      newEngaged = v.newEngaged;
      coordinator = v.coordinator;
      firstTwo = v.firstTwo;
      firstTwo_ = v.firstTwo_;
      regiving = v.regiving;
      oneGraduate = v.oneGraduate;
      match = v.match;
      pref = v.pref;
   }

   public int compareTo(Value v, Level level) {
      int res = 0;
      Iterator<Level> it = listLevel.iterator();
      while( it.hasNext()) {
```

```
            Level levelCur = it.next();
            res = levelCur.compare( this, v);
            if( res != 0 || levelCur.equals(level)) {
               break;
            }
         }
         return res;
      }

      public int compareTo(Object o) {
         Value v = (Value)o;
         if( preAssigned != v.preAssigned) {    // pre-assigned courses
            return preAssigned - v.preAssigned;
         } else if( firstTwo != v.firstTwo) {    // at least 1 out of first 2
            return firstTwo - v.firstTwo;
         } else if( oneGraduate != v.oneGraduate) {    // 1 graduate course
            return oneGraduate - v.oneGraduate;
         } else if( newEngaged != v.newEngaged) {    // new or engaged profs
            return newEngaged - v.newEngaged;
         } else if( coordinator != v.coordinator) {    // coordinators
            return coordinator - v.coordinator;
         } else if( regiving != v.regiving) {    // regiving courses
            return regiving - v.regiving;
         } else if( match != v.match){
            return match - v.match;
         } else {
            return pref - v.pref;
         }
      }

   }
```

REFERENCES

Blum, C., and A. Roli. 2003. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Computing Surveys, Vol. 35, No. 3, Pages: 268 - 308, September.

Edmonds, J. 1965. Paths, trees, and flowers. Canadian J. Math., 17:449-467.

Freuder, C., J. Wallace, and R. Heffernan. 2003. Ordinal constraint satisfaction. In: Fifth Internat. Workshop on Soft Constraints - SOFT'02.

Goldberg, D. 1989. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional.

Hansen, P. and N. Mladenovic. 1998. An introduction to variable neighborhood search. In S. Voss, S. Martello, I. H. Osman, and C. Roucairol, editors, Meta-heuristics, Advances and Trends in Local Search Paradigms for Optimization, pages 433--458. Kluwer Academic Publishers

Hopcroft, E., and M. Karp. 1973. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. SIAM J. Comput. 2:225-231.

Larman, C. 2004. Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition. Prentice Hall.

Johnson, D., and C. McGeoch. 1993. Network Flows and Matching: First DIMACS Implementation Challenge, volume 12. American Mathematics Society, Providence RI.

Kumar, V. 1992. Algorithms for Constraint Satisfaction Problems: A Survey. Artificial Intelligence Magazine, 1:32-44.

Kirkpatrick, S., and C. Gelatt, and M. Veechi. 1983. Optimization by Simulated Annealing. Science, 220:671-681.

Mackworth, K. 1977. Consistency in networks of relations. Artificial Intelligence, 8:99-118.

Minton, S. 1992. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. Artificial Intelligence, 52:161-205.

Munkres, J. 1957. Algorithms for the assignment and transportation problem. SIAM Journal on Computing, 5(1):32-38.

Neapolitan, N. 1997. Foundations of Algorithms using C++ pseudocode, 2nd Ed. Jones and Bartlett Publishers

Palacios, H., and H. Geffner. 2002. Planning as Branch and Bound: A Constraint Programming Implementation. XVIII Latin-American Conference on Informatics (CLEI-2002), Montevideo, Uruguay.

Rudova, H. 1998. Constraints with variables annotations and constraint hierarchies. Springer-Verlag LNCS 1521:409-418.

Russell, S., and P. Norvig. 2002. Artificial Intelligence: A Modern Approach. Chapter 5: Constraint Satisfaction Problems. Prentice Hall.

Tremblay, G. 2004. Modélisation et spécification formelle des logiciels (édition revue et augmentée). Loze-Dion Éditeurs, Inc.

Tsang, E. 1993. Foundations of Constraint Satisfaction. Academic Press.

Voudouris, C. and E. Tsang. 1999. Guided local search. Guided Local Search and its application to the Travelling Salesman Problem. In European Journal of Operational Research, Anbar Publishing, Vol.113, Issue 2 (March), 469-499.

Wilson, M. and A. Borning. 1993. Hierarchical Constraint Logic Programming, TR 93-01-02a, Department of Computer Science and Engineering, University of Washington.