ASSIGNING PROCTORS TO EXAMS USING SCATTER SEARCH

by

MOUJAB MOUSTAPHA ZEIDAN

B.S., Computer Science, Lebanese American University, 1991

Project submitted in partial fulfillment of the requirements for the Degree of Master
of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June 2006
Project approval Form

Student Name: Moujab Moustapha Zeidan
I.D.: 198702220

Thesis Title: Assigning Proctors to Exams Using Scatter Search

Program: M.S. in Computer Science
Division/Dept: Computer Science and Mathematics
School: Arts and Sciences - Beirut

Approved by:

Nashat Mansour, Ph.D.
Professor of Computer Science, L.A.U.

Ibrahim Osman, Ph.D.
Professor of Business Information and Decision Systems, A.U.B.

Date: June 27, 2006
Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name:

Signature: Date:
I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.
To my parents & my wife
Acknowledgments

I would like to extend my deepest gratitude to Dr. Nashaat Mansour for his support and guidance throughout the execution of this project. I am also thankful to Dr. Ibrahim Osman for his valuable help.

I am tremendously grateful to my father Dr. Moustapha, my mother Mayada, my loving wife Rola and my precious kids Moustapha, Hussein and Karim for their continuous care, encouragements and enthusiasm.

I am greatly indebted to Mr. Antoine Lawandos for his long support all the way through my graduate studies and to BLOM Bank’s administration for granting me the financial means to pursue a higher education.

I would like to sincerely thank all my friends and colleagues for their time and their faith.
Abstract

Scatter search is an evolutionary algorithm used to solve a multitude of real-life problems. It can be classified as a population-based methodology in the field of meta-heuristics. The main characteristic of scatter search is that it attempts to find improved solutions by combining others which are non-randomly generated.

The scatter search methodology provides a general template consisting of five methods which can be fine tuned to fit the specific problem under consideration. This flexibility has allowed the application of the scatter search to many combinatorial and optimization problems.

One of the problems solved by the scatter search algorithm is the assignment of proctors to exams. This problem is encountered in every university at the time of final examinations when the scheduling of proctors is needed to oversee the students sitting for exams. This task has been reputed to be a very complex one because of the large number of exams and of the various constraints of this problem. Furthermore, when done manually, assigning proctors to exams is an extremely difficult and time-consuming job for university staff responsible for this task and this entails that there is a need to automate the allocation of proctors to exams.

The purpose of this project is to implement a computer-based decision support system that attempts to assign proctors to exams. The software was developed using Java, which is today's most advanced object-oriented language. The algorithm used is the scatter search methodology since it was found to provide good solutions for the problem under consideration. In addition, empirical results were obtained using real-life data and were found to give high-quality solutions that maximize the proctors' preferences while maintaining the fairness of the workload given to proctors.
## Contents

1. Introduction  
   2. Scatter Search Methodology  
      2.1. Historical Background  
      2.2. Scatter Search Basic Design  
      2.3. Scatter Search Advanced Design  
         2.3.1. Dynamic Reference Set Update  
         2.3.2. Reference Set Rebuilding  
         2.3.3. Advanced Subset Generation Method  
      2.4. Scatter Search Applications  
         2.4.1. Unconstrained Nonlinear Optimization  
         2.4.2. 0-1 Knapsack Problem  
         2.4.3. Project Scheduling Problem  
         2.4.4. Multi-Objective Bus Routing Problem  
3. Proctor Assignment to Exams  
   3.1. The Proctor Assignment Problem  
      3.1.1. Problem definition  
      3.1.2. Problem Formulation  
      3.1.3. Solution Evaluation  
   3.2. Scatter Search Solution of the PAP  
      3.2.1. Diversification Generator Method  
      3.2.2. Improvement Method  
      3.2.3. Reference Set Update Method  
      3.2.4. Subset Generation Method  
      3.2.5. Solution Combination Method  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>4</td>
</tr>
<tr>
<td>2.1.</td>
<td>4</td>
</tr>
<tr>
<td>2.2.</td>
<td>6</td>
</tr>
<tr>
<td>2.3.</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1.</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2.</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3.</td>
<td>12</td>
</tr>
<tr>
<td>2.4.</td>
<td>13</td>
</tr>
<tr>
<td>2.4.1.</td>
<td>13</td>
</tr>
<tr>
<td>2.4.2.</td>
<td>16</td>
</tr>
<tr>
<td>2.4.3.</td>
<td>19</td>
</tr>
<tr>
<td>2.4.4.</td>
<td>21</td>
</tr>
<tr>
<td>3.</td>
<td>24</td>
</tr>
<tr>
<td>3.1.</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1.</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2.</td>
<td>27</td>
</tr>
<tr>
<td>3.1.3.</td>
<td>29</td>
</tr>
<tr>
<td>3.2.</td>
<td>30</td>
</tr>
<tr>
<td>3.2.1.</td>
<td>32</td>
</tr>
<tr>
<td>3.2.2.</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3.</td>
<td>35</td>
</tr>
<tr>
<td>3.2.4.</td>
<td>36</td>
</tr>
<tr>
<td>3.2.5.</td>
<td>37</td>
</tr>
</tbody>
</table>
4. Java Implementation of the Proctor Assignment
   
   Problem using Scatter Search 40
   
   4.1. Class Diagram 40
   
   4.2. User Interface 41
      
   4.2.1. Input Data 41
   
   4.2.2. Output Data 44
   
   4.3. Diversification Generator Class 45
   
   4.4. Improvement Class 47
   
   4.5. Reference Set Update Class 50
   
   4.6. Subset Generation Class 51
   
   4.7. Solution Combination Class 54
   
   4.8. Overall Procedure 59
   
   4.9. Utilities Class 61
   
5. Empirical Results 67

6. Conclusion and Future Work 69

References 70
List of Figures

Figure 2.1 Overall algorithm for the scatter search method 5
Figure 2.2 The interact between the different methods of scatter search 7
Figure 2.3 The pseudo-code of the most basic design of scatter search 7
Figure 2.4 The four types of subsets 13
Figure 3.1 The steps for assigning proctors to exams using scatter search 31
Figure 3.2 The pseudo-code for the diversification method 33
Figure 3.3 The pseudo-code for the improvement method 34
Figure 3.4 The pseudo-code for the voting technique 37
Figure 3.5 The pseudo-code for the repair procedure 38
Figure 4.1 The Class Diagram for the PAP using SS 40
Figure 4.2 The software's user interface 41
Figure 4.3 Sample of the exam.txt input file 42
Figure 4.4 Sample of the preference input file 42
Figure 4.5 Sample of the TA input file 43
Figure 4.6 Sample of the TK input file 43
Figure 4.7 Sample of the Solution output file 44
Figure 4.8 Diversification Generator's pseudo-code 45
Figure 4.9 Improvement method's pseudo-code 48
Figure 4.10 Reference set generator's pseudo-code 50
Figure 4.11 Subset generation's pseudo-code 52
Figure 4.12 Combination method's pseudo-code 54
Figure 4.13 The Long Task's pseudo-code 59
Figure 4.14 The pseudo-code for the utilities class 61
List of Tables

Table 2.1 Relationship between the input and output solutions 6
Table 5.1 Results and Comparison for the solutions produced 67
Chapter 1

Introduction

Scatter search is an evolutionary algorithm used to solve a multitude of real-life problems. It can be classified as a population-based methodology in the field of meta-heuristics. The main characteristic of scatter search is that it attempts to find improved solutions by combining others which are non-randomly generated. Since its introduction in 1977, it has been used to solve many combinatorial and optimization problems because it is a very flexible methodology since it provides a general template and allows the fine-tuning of its various methods to fit the specific problem under consideration. The scatter search template is well-known to be made up of five methods, namely, the diversification generation method, the improvement method, the reference set update method, the subset generation method and the solution combination method. The reference set is a vital part of the scatter search since it contains the starting set of solutions that are considered for combination and that will eventually produce the high-quality and optimal solutions needed.

One of the problems addressed by scatter search is the assignment of proctors to exams. This problem arises in each university at the time final exams are held since each exam should have a specified number of proctors. In addition, proctors are usually teaching assistants (TAs) who have a specific amount of time during which they are available to proctor examinations. Moreover, TAs have to sit for their own exams and therefore, they are incapable of proctoring exams that conflict with one of their own (Marti et al., 2000).
The Proctor Assignment Problem (PAP) is regarded as an expansion of the Generalized Assignment Problem (GAP) (Marti et al., 2000).

In fact, the Generalized Assignment Problem aims at assigning tasks to agents while minimizing the cost of such assignment knowing that an agent has limited resources. Furthermore, the GAP requires that only one agent be assigned to a certain task (Lourenço et al., 1998).

The proposed solution of the GAP consists of applying adaptive heuristics and is made up of three steps. In the first step, initial solutions are obtained from a greedy randomized heuristic, then in the second step, a local search is used in order to improve these initial solutions and in the third step, parameters are updated and these steps are repeated until a stopping criterion is reached (Lourenço et al., 1998).

Therefore, the PAP is based on the GAP but it also allows more than one agent to be assigned to a task and it considers additional constraints. In addition the GAP consists of a single objective function and its purpose is to maximize the total profit while the PAP is made up of tow objective functions to maximize proctors’ preferences and workload fairness (Marti et al., 2000).

This project aims at implementing a computer-based system which solves the assignment of proctors to exams problem using the scatter search algorithm. This software was developed using Java, which is today’s state-of-the-art object-oriented programming language.

Chapter two provides an overview of the scatter search methodology by looking at its original proposal and by exposing the template that was adopted in modern-time applications. It also presents several well-know problems in the operations research field and depicts the way these problems can be solved using the scatter search framework.
Chapter three presents the problem of assigning proctors to exams. It formulates the problem's objective functions and constraints and sheds the light on the various methods of scatter search as adapted to solve this problem. Furthermore, the overall procedure of scatter search is discussed and the interactions among the five methods that make up the building blocks of any scatter search implementation are presented in details.

Chapter four outlays the software designed to provide a decision-support system for assigning proctors to exams. The input of the data to this system is discussed as well as the design of the classes that were coded to solve our problem. Furthermore, the pseudo-code of the Java code is presented to give a deep insight on the practical implementation details that were devised to produce the desired solutions to this problem.

Chapter five presents the empirical results that were obtained when testing the decision-support system developed with real-life data.

Finally, the results of our implementation are discussed and the recommendations for future work are revealed in Chapter six.
Chapter 2

Scatter Search Methodology

2.1 Historical Background

The origin of the scatter search methodology goes back to the year 1997. At that time, it was proposed as a heuristic to solve integer-programming problems (Glover, 1997).

In this first version of scatter search, solutions are non-randomly generated, and then reference points are chosen among the obtained good solutions. Then a weighted focal point is generated starting from these solutions. This center is used with sub-groups of the initial reference solutions thus resulting in new sub-groups of solutions. Then, centers of gravity are generated with respect to these new sub-regions and are used to determine the desired solutions (Glover, 1997).

In essence, this approach consists of merging several solutions to obtain the central point of the reference points in order to get solutions to the problem under consideration (Glover, 1997).

The original version of scatter search was proposed as follows by Glover in 1997:

- Step 1. Start with a vector of solutions using a heuristic specifically designed for the problem under consideration, and choose the best solutions to be reference solutions.

- Step 2. Create new solutions by combining subsets of the current reference solutions.
• Step 3. Take the best solutions obtained in Step 2 as an input for a repetition of the heuristic described above. These three steps are re-applied in sequence for a predefined number of loops.

Then, a scatter search template was proposed by Glover and it was used as a model for all modern implementations of the scatter search methodology.

In this template, the following functions are needed in order to apply the steps of scatter search (Glover, 1997):

1. A diversification generator which aims at generating diverse trial solutions starting from an arbitrary seed solution.

2. An improvement method which enhances the initial results. This method accepts infeasible solutions but will be expected to produce feasible solutions.

3. A reference set update method which constructs the reference set containing the b most attractive solutions obtained (b can take values from 20 to 40).

4. A subset generation method which produces sub-groups from the collection of solutions obtained above.

5. A solution combination method which performs the combination of the above subsets of solutions.

The overall algorithm for the scatter search method is as follows (Glover, 1997):

<table>
<thead>
<tr>
<th>Initial Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Seed Solution Creation: create one or more seed solutions to obtain a starting point of the remainder of the algorithm.</td>
</tr>
<tr>
<td>2. Diversification Generator: diverse trial solutions are generated starting from the seed solutions.</td>
</tr>
<tr>
<td>3. Improvement and Reference Set Update Methods: trial solutions are enhanced and the reference set is updated.</td>
</tr>
<tr>
<td>4. Steps 2 and 3 are repeated until the reference set is built.</td>
</tr>
</tbody>
</table>
Scatter Search Phase

5. Subset Generation Method: subsets of the reference set are generated.
6. Solution Combination Method: subsets produced in Step 5 are combined.
7. Improvement and Reference Set Update Methods: combined solutions are enhanced and they are considered for membership in the reference set.
8. Steps 5-7 are repeated until reaching a specified cutoff limit on the total number of iterations.

Figure 2.1 Overall algorithm for the scatter search method

The relationship between the input and output solutions is depicted in the following table (Glover, 1997):

<table>
<thead>
<tr>
<th>Source of Input Solutions</th>
<th>Source of Output Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary seed solutions</td>
<td>Diversification Generator</td>
</tr>
<tr>
<td>Diversification Generator</td>
<td>Improvement Method</td>
</tr>
<tr>
<td>Improvement Method</td>
<td>Reference Set Update Method</td>
</tr>
<tr>
<td>Reference Set Update Method</td>
<td>Subset Generation Method</td>
</tr>
<tr>
<td>Subset Generation Method</td>
<td>Solution Combination Method</td>
</tr>
<tr>
<td>Solution Combination Method</td>
<td>Improvement Method</td>
</tr>
</tbody>
</table>

2.2 Scatter Search Basic Design

The scatter search template discussed above can be adapted to several problems because its components can be fine tuned in various ways to suit the needs of the problem under consideration (Laguna et al., 2003).
The basic implementation of the scatter search consists of five methods which are: the diversification generation method, the improvement method, the reference set update method, the subset generation method and the solution generation method (Laguna et al., 2003).

The diagram below shows how these methods interact (Laguna et al., 2003):

![Diagram](image)

Figure 2.2 The interact between the different methods of scatter search.

In the most basic design of scatter search, these methods interact among each others as can be shown in the following pseudo-code (Laguna et al., 2003):

1. Start with $P = \emptyset$. Use the diversification generation method to construct a solution and apply the improvement method. Let $x$ be the resulting solution. If $x \notin P$ then add $x$ to $P$ (i.e., $P = P \cup \{x\}$), otherwise, discard $x$. Repeat this step until $|P| = P_{\text{size}}$.

2. Use the reference set update method to build $\text{RefSet} = \{x^1, \ldots, x^b\}$ with the 'best' $b$ solutions in $P$. Order the solutions in $\text{RefSet}$ according to their objective function value such that $x^1$ is the best solution and $x^b$ the worst.

Make $\text{NewSolutions} = \text{TRUE}$.

while ( $\text{NewSolutions}$ ) do


3. Generate NewSubsets with the subset generation method.
   Make NewSolutions = FALSE.
   while ( NewSubsets ≠ ∅ ) do
4. Select the next subset s in NewSubsets.
5. Apply the solution combination method to s to obtain one
   or more new trial solutions x. Apply the improvement
   method to the trial solutions.
6. Apply the reference set update method.
   if ( RefSet has changed ) then
7. Make NewSolutions = TRUE.
   end if
8. Delete s from NewSubsets.
end while
end while

Figure 2.3 The pseudo-code of the most basic design of scatter search.

As we can see, the procedure starts with the diversification generator in order

to build a collection P of diversified solutions. The dimension of this collection
(PSize) should be large and is typically 10 times greater than the reference set
(RefSet). Then the first reference set is constructed using the reference set update
method by taking the best b distinct and diverse solutions from P.

This is done by first taking the most attractive b₁ solutions from P which are
included into the reference set and removed from P. Then, the remaining solutions in
P are used to compute the least distance measure with respect to all solutions which
make up the current reference set. The solution with the greatest minimum distance
gains entry to the reference set and is removed from P and the least distances
measures are computed again. This procedure is iterated b₂ times knowing that b₂=b-
b₁. The resulting reference set would be made up of b₁ high-quality solutions and b₂
diverse solutions.

These solutions are sorted with respect to the goodness of their objective
function whereby the solution with the highest utility function figures in the first
position. Next the look-up begins with setting the Boolean variable NewSolutions to TRUE.

Afterwards, NewSubsets are constructed using the subset generation method constructed and NewSolutions is assigned the value of FALSE. The subset generation method in its simplest form generates all pairs of reference solutions. That is, \((b^2-b)/2\) NewSubsets of solutions are obtained. Then these pairs are subjected to the solution combination method to produce new trial solutions which are then subjected to the improvement method.

Then the reference set update method is run another time to get the new RefSet by taking the best solutions found from the current RefSet and the new trial solutions. These solutions are chosen according to their quality, i.e. according to their objective function. In case the RefSet changes at this stage, then NewSolutions is set to TRUE to indicate that a solution has gained membership in the reference set. The subset that was considered for combination is deleted from the collection of subsets or NewSubsets. Then all sub-collections constituting NewSubsets are combined and if no better solution is added to the reference set, the scatter search is terminated.

It is worthwhile noting that after the combination method is applied, the reference set is updated in a static way. This is so because when trial solutions are generated by the means of the combination method, they are placed in a solution pool. Then the b most attractive solutions found in the union of the reference set and the pool generated are chosen to construct the new reference set (Laguna et al., 2003).
2.3 Scatter Search Advanced Design

The advanced implementations of the scatter search design are concerned with the way the five methods of the basic design are conceived. It is worthwhile noting that improving performance in solving a given problem often conflicts with the easiness of the procedure being devised. Therefore, advanced designs introduce the drawback of a more complex methodology (Marti et al., 2006).

In the next sections, we will discuss some of the possible modifications that can be applied to the basic methods of the scatter search algorithm.

2.3.1 Dynamic Reference Set Update

The reference set is the main building block of the scatter search algorithm. That is, the reference solutions are not of a high quality, the scatter search will not be able to improve these solutions even when using a sophisticated combination method. Hence, the reference set should be built and maintained in an appropriate way during the search in order to get better trial solutions using the combination method (Marti et al., 2006).

In the simple version of the scatter search, the solutions that are included in the RefSet are combined after the combination of the whole sub-groups in NewSubsets. Then, the reference set is re-constructed by choosing the most attractive solutions in the merger of Pool and the current reference set. The latter way of updating the reference set is called the static update (Marti et al., 2006).

Another option for updating the reference set is based on dynamically updating it by combining solutions in a more rapid manner than in the simple
design. This means that whenever a solution is added to the reference set, it is combined with other solutions as quickly as possible. Therefore, when a solution gains membership into the reference set, it directly becomes part of this set without waiting until all combinations have been performed. As a result, an intermediary pool of solutions is not needed to store new trial solutions as they can be directly bypassed or added to the reference set.

The benefit of the dynamic reference set revision is that the least attractive solutions are rapidly substituted by better solutions and subsequent combinations are directly performed using the better solutions. The drawback of this method is that some combinations could produce good solutions but they are discarded as solutions are deleted from the current reference set. Moreover, the implementation of the dynamic reference set revision is found to be more complicated than the static one. In addition, the order in which the combinations are performed is very important in the dynamic update as it imposes which combinations are going to be skipped, whereas, in the static update method, combinations can be done in any order as the reference set is not changed before the totality of the combinations are done. Therefore, when performing a dynamic update of the reference set, we could try combinations in more than one sequence in order to obtain better outcomes (Marti et al., 2006).

2.3.2 Reference Set Rebuilding

When the combination method is applied, we reach the stage where the reference collection of solutions is not changing anymore because trial solutions being generated are not of sufficient quality to replace others in the current reference
set. When we reach this situation, the reference set could be partially rebuilt using the diversification generation method. Assuming that the reference set consists of \( b = b_1 + b_2 \) solutions, the reference set rebuilding procedure starts by deleting solutions \( x^{b_2+1} \ldots x^b \) from the RefSet. Then, the diversification generation method is applied to produce diversified solutions when compared with solutions \( x^1 \ldots x^{b_1} \). A set \( P \) containing new solutions is constructed, then \( b_2 \) solutions are selected from \( P \) and added to the RefSet in order to maximize the diversity of these by maximizing the minimum distance of these solutions to all the elements in the current reference set (Marti et al., 2006).

### 2.3.3 Advanced Subset Generation Method

The solution combination method can combine more than two solutions. Hence, the subset generation method is not limited to generating subsets containing 2 solutions, rather, it can be designed in a way to produce subsets of different sizes.

The advanced subset generator works on expanding pairs of solutions into larger collections and at the same time limiting the resulting number of subsets (Marti et al., 2006).

Therefore, in this strategy, not all of the subsets made up of two, three and so on elements are produced as this approach would produce a very large number of subsets (for example, there exists \( 1013 \) possible combinations of solutions found in a set containing \( b = 10 \) elements).

In essence, sample subsets of various sizes are chosen by defining subset types as follows (Marti et al., 2006):
• Subset Type 1: all 2-element subsets.
• Subset Type 2: 3-element subsets derived from the 2-element subsets by augmenting each 2-element subset to include the best solution not in this subset.
• Subset Type 3: 4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solutions not in this subset.
• Subset Type 4: the subsets consisting of the best $i$ elements, for $i = 5$ to $h$.

Figure 2.4 The four types of subsets

2.4 Scatter Search Applications

2.4.1 Unconstrained Nonlinear Optimization

The "unconstrained non-linear optimization problem" can be formulated in the following way:

Minimize $f(x)$

Subject to $1 \leq x \leq u$

In this mathematical formulation, $f(x)$ consists of a non-linear function of $x$ and $x$ is a continuous and bounded variable.

As a numerical example, we consider the following formulation:

Minimize

$$100(x_2 - x_1)^2 + (1 - x_1)^2 + 90(x_4 - x_2)^2 + (1 - x_2)^2 + 1.01(0.9(x_2 - 1)^2 + (x_4 - 1)^2) + 19.8(x_2 - 1)(x_4 - 1)$$
Subject to

\[-10 \leq x_i \leq 10 \quad \text{for } i=1,\ldots,4\]

The diversification generator uses a frequency-based memory approach which consists of splitting the region spanned by the variables \((u_i-l_i)\) into four sub-regions. Then a sub-range is randomly selected and a value is randomly generated within the chosen sub-range.

Since the variables are chosen within the allowed range, the resulting construction of the solution is guaranteed to be feasible and hence, the improvement method does not have to deal with infeasible solutions.

The improvement method applied is a classical local optimizer for this kind of problems.

Then, the initial reference set (containing \(b=b_1+b_2\) elements) is constructed by taking the best \(b_1\) solutions from the diverse solutions generated. Then, for each remaining solution, the minimum distances to all entries in the present reference set are calculated. That is:

\[
d_{\text{min}}(x) = \min_{y \in \text{ref set}} \{d(x, y)\}
\]

where \(d(x, y) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}\)

Then the solution having the largest distance is chosen to be included in the reference set and this process is repeated \(b_2\) times until we get a reference set of size \(b\).

Then the reference set is passed to the subset generator which aims at getting the two-by-two combinations from the current reference set. That is, this method will result in \((b^2-b)/2\) subsets.
Once the subsets are generated, the combination method can be applied to them. The combination method is problem-specific and can be designed in a way to produce one or more solutions.

The method considered in this problem consists of creating three trial solutions for every combination of two reference solutions that are denoted by \( x \) and \( x' \). These trial solutions are a linear combination of reference solutions and are obtained as follows:

- **C1.** \( x = x' - d \)
- **C2.** \( x = x' + d \)
- **C3.** \( x = x'' + d \)

Where \( d = r \frac{x'' - x'}{2} \), \( r \) is randomly chosen in the range going from 0 to 1.

Solutions obtained at this step are improved before being tested for entry to the reference set. This is done by applying the static update of the reference set by taking the most attractive \( b \) solutions found in the union of the present reference set and the output solutions obtained from the combination method.

The search then continues in a cycle in such a way as to apply the combination method, the improvement method and the reference set update method in sequence until the reference set becomes stable, i.e. no new elements are added to it. At this stage, the reference set can be rebuilt using the best \( b_1 \) solutions in the obtained reference set and using the diversification generation method to get the remaining \( b_2 \) diverse solutions. The whole scatter search process is restarted until reaching a specified maximum iteration limit (Laguna et al., 2003).
2.4.2 0-1 Knapsack Problem

The knapsack problem is very famous in the area of operations research. It consists of choosing elements from a set of items that maximize the objective function while respecting a capacity constraint. The problem can be applied to the situation whereby a hiker has to select items to carry on his backpack while controlling the total weight carried. In other words, if each item $i$ has a utility $c_i$ and a weight $a_i$, then the hiker has to choose items that maximize the total utility and that keep the weight under a defined constant $b$ (Laguna, 2002).

Therefore, the 0-1 knapsack problem is formulated as follows:

Maximize $\sum_i c_i x_i$

Subject to $\sum_i a_i x_i \leq b$

$x_i \in \{0,1\} \forall i$

As a numerical example, we consider the following 0-1 knapsack problem:

Maximize $11x_1 + 10x_2 + 9x_3 + 12x_4 + 10x_5 + 6x_6 + 7x_7 + 5x_8 + 3x_9 + 8x_{10}$

Subject to $33x_1 + 27x_2 + 16x_3 + 14x_4 + 28x_5 + 30x_6 + 31x_7 + 33x_8 + 14x_9 + 18x_{10} \leq 100$

$x_i = \{0, 1\}$ for $i = 1, \ldots, 10$.

The diversification generator applied to this problem consists of choosing a value for a variable $h$ such that $h \leq n - 1$ where $n$ represents how many variables there are in the problem.

The initial seed is set as $x = (0,0,0,\ldots,0)$. Then, for every $h$, two types of solutions are computed. The first type is obtained by initializing $x'_i = 0$ for all $i$. Then these solutions are modified according to the rule:
\[ x'_i = 1 - x_i \]

\[ x'_{i+k} = 1 - x'_{i+k} \quad \text{for} \quad k=1 \ldots n/h \]

The second type of solutions is given by computing the complement of type 1 solutions or in other words: \( x'_i = 1 - x_i \).

The solutions obtained with the diversification generator are subjected to the improvement method. If this solution isn’t feasible, an attempt to turn it into a feasible one is made by the improvement method which toggles variables from zero to one so that the problem’s constraint become respected. The variables to be changed are considered starting with the one with the smallest profit-to-weight proportion to the one with the highest.

Then, when the solution turns feasible, the improvement method tries to obtain a better solution by changing variables from zero to one. The variables are considered starting with the one having the largest profit-to-weight proportion to that with the smallest proportion.

The reference set update method is then applied in order to generate the reference collection of solutions. This collection is divided into two parts, the first part consists of \( b_1 \) highly attractive solutions and the second one is made up of \( b_2 \) diverse solutions.

The set \( b_2 \) of diverse solutions is found by defining a diversity measure as being equal to the distance among solutions computed as the sum of the absolute differences between the variables.

The solutions to be comprised in the set of diverse solutions are the ones that have the greatest distance measure to other solutions in the reference set.

Once the reference set is obtained, the subset generation method can be applied in order to generate all 2-element subsets, all 3-element subsets derived from
the 2-element subsets by augmenting each 2-element subset to include the best solution not in this subset, all 4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solution not in this subset and all other subsets consisting of the best $i$ elements (as measured by the objective value), for $i = 5$ to $b$.

The subsets obtained above are then combined to get new trial solutions. The combination method consists of calculating a score for each variable in a given trial solution. The score for variable $i$ in a solution is given by:

$$
\text{score}(i) = \frac{\sum_{j \in S} OV(j) \cdot x'_j}{\sum_{j \in S} OV(j)}
$$

In this formula, $OV(j)$ is the objective value of solution $j$ and $x'_j$ is the value of the $i^{th}$ variable in solution $j$.

The new combined solution is the approximation of the score of the variable to the nearest integer:

$$
x'_j = \begin{cases} 
1 & \text{score}(i) > 0.5 \\
0 & \text{score}(i) \leq 0.5 
\end{cases}
$$

After a new trial solution is constructed, it is improved by the improvement method and it is considered for membership in the reference group. A solution gains membership in the reference group if its objective value is better that the solutions currently considered as high-quality ones or if it improves the diversity of the solutions considered to be highly diverse ones.

If at this stage, new solutions enter the reference set, then the subset generation, the combination and the improvement methods are re-used on the new reference set until it converges which means that no new solutions are added to it.
Afterwards, the diversification generator can be repeated starting from another starting point and the whole scatter search is run till reaching a specified number of iterations (Laguna, 2002).

### 2.4.3 Project Scheduling Problem

In general, a project is made up of various tasks that have precedence over one another, that should be completed in a fixed interval and that make use of scarce resources. The project scheduling problem consists of determining the starting time of each activity in the project and to minimize the overall duration of the project or the project’s makespan (Yamashita et al., 2006).

The resource availability cost problem or RACP aims at minimizing the price of the resources needed by the job so that it delivered by a specified deadline.

Let us consider a project that consists of \( n \) activities related by precedence relations \((i, j) \in H\) whereby tasks 1 and \( n \) are dummy tasks that are used as an indication of the starting and finishing times of the project.

In this project, each activity \( i \) has duration \( d_i \) and requires \( r_{ik} \) units of resource of type \( k \) (\( k = 1, \ldots, m \)). \( A_i \) designates the activities undertaken in the interval \((t-1, t]\) and \( D \) is the project’s deadline. \( C_k(a_k) \) is a discrete non-decreasing cost function associated with the availability \( a_k \) of resource type \( k \) and \( f_i \) represents the finishing times of activities. The problem’s formulation is as follows:

\[
\text{Minimize } \sum_k C_k(a_k)
\]

Subject to

\[
f_j \geq f_i + d_j \quad \forall (i, j) \in H
\]
\[ f_i = 0 \]
\[ f_n \leq D \]
\[ \sum_{k} r_{kt} \leq \alpha_k \quad k = 1, \ldots, m \quad \text{and} \quad t = 1, \ldots, f_n \]

The set \( P \) of diverse trial solutions is obtained by defining \( [z_k^o, z_k^f] \) as an interval containing possible values of resource availability of type \( k \). This interval is divided into \( g \) sub-intervals and a frequency matrix \( M(k)[i] \) (\( k = 1, \ldots, m \), \( i = 1, \ldots, g \)) is constructed in order to store the number of times \( \alpha_k^{sol} \) takes a value in the interval \( [z_k^o, z_k^f] \). Then, a sub-interval is selected according to a probability function inversely proportional to its frequency in the matrix and a value for \( \alpha_k^{sol} \) is randomly generated in this interval.

After the initial set of trial solutions is generated, the reference set update method is applied to create the reference set of solutions which gain membership into this set according to their quality or their diversity. To achieve this goal, \( b_1 \) solutions are chosen from \( P \) according to their objective function value and \( b_2 \) solutions from \( P \) that that maximize the minimum distance to the current reference set. The distance \( d(a_i^t, a_j^t) \) between two solutions \( a_i \) and \( a_j \) is computed as:

\[ d(a_i^t, a_j^t) = \sum_{k=1}^{m} | a_i^t - a_j^t | \]

After the initial reference set is constructed, the subset generation method is applied in order to get the four types of subsets. These subsets are then combined so that new trial solutions are obtained. We let \( s = (a^1, \ldots, a^r) \) be the set of \( r \) solutions that are subjected to the combination method.
New solutions can be constructed by starting from one solution and generating a path in its neighborhood that leads toward the other solutions.

First an initial solution $a^1$ is chosen from $s$ in a way as to maximize the minimum distance among all elements in $s$. Then, a path beginning at $a^1$ and ending at each of the other solutions in $s$ is generated. As new solutions are obtained along this path, the best feasible solution found or the least infeasible solution relative to the deadline is stored. Then, the best solution found along a given path is subjected to the improvement method. This process is repeated until all $(r-1)$ trajectories staring from a solution $a^1$ to all solutions $a^i$ and it returns the best feasible improved solution found along these trajectories (Yamashita et al., 2006).

### 2.4.4 Multi-Objective Bus Routing Problem

School bus routing is concerned with the way students are transported to and from schools. The scheduling and routing of buses has multiple objectives which are to minimize the transportation cost and to minimize the transportation time.

Therefore the objective of this problem is to minimize the total number of buses in operation and at the same time, to minimize the time that a student spends in the bus while keeping in mind that the buses have different capacities (Corberan et al., 2002).

The following notations are used in the problem formulation:

- $N = \{ 0, 1, \ldots, n \}$ represents locations where 0 is the school and $j$ (for $j = 1, \ldots, n$) is where students live.
- $M = \{ 1, \ldots, m \}$ represents the set of buses.
\( R_i = \{ r_i(1), \ldots, r_i(n_i) \} \) gives the route for bus \( i \) where \( r_i(j) \) is the \( j^{th} \) location visited

\( t_{jk} \) is the time it takes to go from location \( j \) to location \( k \)

\( c_i \) is the capacity of bus \( i \)

\( q_j \) is the number of students to be picked up at location \( j \)

\( \text{length}(i) \) is the length of route \( i \) where

\[
\text{length}(i) = \sum_{j=1}^{n_i} t_{r_i(j)r_i(j+1)}
\]

The problem can be formulated as:

Minimize \( m \)

Minimize \( t_{\text{max}} = \max_{n,K}\{\text{length}(i)\} \)

Subject to

\[
\sum_{j=1}^{n_i} q_{r_i(j)} \leq c_i \quad i = 1, \ldots, m
\]

\[
\sum_{i=1}^{n} n_i = n
\]

\( r_i(j) \neq r_k(j) \quad \forall i, k \in M, \forall j \in N \)

The first objective function is to minimize the number of buses and the second to minimize the time spent on the bus. The constraints make sure that the bus capacity is not exceeded, that all students are picked up and that a location cannot be visited by more than a route.

The common way to deal with problems that have multiple objective functions is to combine these functions into a single one:

Minimize \( m + \lambda t_{\text{max}} \)

The scatter search approach to solve this problem used the following methods:

- H1 and H2 which are heuristics responsible for generating routes
- SWAP which is an exchange procedure to find a local optimum for the route’s length
• INSERT which is an exchange method used to improve $t_{\text{max}}$

• COMBINE which aims at combining solutions in the reference set to obtain new ones

First, the heuristics H1 and H2 are used to get the routes for each bus by varying the value of $t_{\text{max}}$. It is to be noted that when the value is $t_{\text{max}}$ is large, solutions with a small number of routes are generated as opposed to the case when $t_{\text{max}}$ is small, solutions with a large number of routes are generated.

These solutions are then passed to an improvement procedure which starts by applying SWAP to each route and then applying INSERT to the obtained solution. In case a route is changed during the INSERT step, then SWAP is re-applied to the changed routes.

Then, solutions with $m$ routes are considered and the best $b$ solutions are chosen to initialize the reference set. Solutions are chosen according to $t_{\text{max}}$ as the solutions considered have all the same number of routes.

Afterwards, the pairs of solutions in the current reference set are combined using the COMBINE routine. Then these combined solutions are subjected to improvement as discussed previously, that is, by applying SWAP then INSERT and then SWAP to the changed routes. Then the best $b$ solutions are chosen from the union of the current reference set and the set of combined solutions. Combination of solutions is repeated and the reference set is updated until no new solutions are added to it (Corberan et al., 2002).
Chapter 3

Proctor Assignment to Exams

3.1 The Proctor Assignment Problem

The problem of assigning proctors to exams is known to be a complex task as the number of exams is quite large and the constraints to be respected to make the assignments are usually complicated. As a result, the university staff in charge of making the assignments of proctors to exams were spending a large amount of time to achieve this goal manually (Awad et al., 1998).

Therefore, automating this task would not only save time but would also produce assignments that optimize the proctors’ preferences.

In order to solve the proctor assignment problem, we could consider to list all the possible assignments of proctors to exams. If we do so, we will find that the resulting number of combinations is extremely high. For instance, if our problem consists of assigning $p$ proctors to $t$ time slots during which exams are scheduled to take place and we have $a$ different assignments in each time slot, then there will be

$$\frac{t \times p^t}{(p-a)!}$$

different combinations of assignments. For example, if we take $p=100$, $t=45$, and $a=30$, we will get $3.5 \times 10^{59}$ solutions (Awad et al., 1998).

As we can see, enumeration of all solutions and deterministic algorithms are not suitable for this kind of problems. Instead, methods that sample the solution space are found to be more convenient. Such methods include random search, heuristics, simulated annealing, tabu search, and genetic algorithms (Awad et al., 1998).
Specifically, a basic genetic algorithm coupled with problem-specific heuristics was found to give acceptable results for generating assignments of proctors to exams. The purpose of the heuristics is to find good initial solutions to be passed to the genetic algorithm which then attempts to improve these solutions by applying basic crossover and mutation methods (Awad et al., 1998).

3.1.1 Problem Definition

In each university, final exams are scheduled and teacher assistants (TAs) are required to proctor them. Each TA has a predefined number of hours during which he can proctor exams. Furthermore, TAs also have exams so they cannot proctor exams that conflict with theirs (Marti et al., 2000).

In addition, each exam needs a certain number of proctors. Therefore, the Proctor Assignment Problem deals with the problem of assigning TAs to exams, taking into account the following constraints:

- Each exam has to be proctored by a given number of TAs.
- A TA has a maximum number of hours that he can dedicate to proctor exams.
- A TA can only proctor one exam at a time.
- A TA can only proctor an exam that does not conflict with his exams.

Knowing that TAs have to sit for their own exams, it is normal that they have preferences to proctor on certain days and to avoid other days. For instance, a TA prefers not to proctor an exam that is scheduled the day before one of his exams.
Consequently, one objective of the proctor assignment problem is to assign proctors to exams in such a way as to maximize these preferences.

A second consideration to be taken into account while assigning proctors to exams is to make the workload balanced between TAs because unfair workloads would result in a clash between the TAs and between the TAs and the university’s administration office.

In order to quantify the fairness of the workload resulting from the assignments of proctors to exams, we could minimize the difference between the TA who has been given the largest workload and the TA who has been given the smallest workload. In other words, we can formulate our model so that we maximize the minimum workload assigned to each TA. This workload can be computed as the ratio of the assigned hours to each TA by the number of available hours of each TA.

The Proctor Assignment Problem (PAP) under consideration is based on the Generalized Assignment Problem (GAP) while allowing more than one proctor to be assigned to an exam and preventing the assignment of a TA to more than one exam that are scheduled during the same period.

In addition, in the proctor assignment problem, we have two objective functions, which are to maximize TAs’ preferences and workload fairness. This multi-objective combinatorial optimization problem can be solved using the scatter search method. This method together with its application to solve the PAP will be described in the following section (Marti et al., 2000).
3.1.2 Problem Formulation

In general, an exam day can be split into two periods: the first from 8:00 AM till 2:00 PM and the second from 2:00 PM till 9:00 PM. Therefore, if we denote by \( d \) the number of exam days, then the number of exam periods will be \( k = 2d \).

It is to be understood that an exam starts and ends in the same day but can take place over one or two periods. Furthermore, TAs express their preferences for certain periods and these are subsequently interpreted as preferences for exams. For example, if a TA has a high preference for a period, then we can consider that the TA has a high preference for all the exams that are scheduled during this period. In the case where an exam is scheduled over two periods, then we assume that the lower value of the preference among the two periods will be the preference for this exam.

If we denote by \( J \) be the set of \( m \) exams \((j = 1, \ldots, m)\) and \( I \) the set of \( n \) TAs \((i = 1, \ldots, n)\), then the following symbols constitute the parameters for this problem:

\[
\begin{align*}
a_i &= \text{maximum number of available hours for TA } i \\
b_j &= \text{number of hours associated with exam } j \\
t_j &= \text{number of TA's required for exam } j \\
c_{ij} &= \text{preference of TA } i \text{ for the exam } j
\end{align*}
\]

\( P_i = \text{the set of exams that overlap with any TA } i \text{'s exams} \)

\( T_k = \text{the set of exams scheduled in period } k \)

\( d = \text{number of exam days} \)

In addition we let \( x_{ij} \) be a binary variable such that:

\[
\begin{align*}
x_{ij} &= 1 \text{ if TA } i \text{ is assigned to exam } j \\
x_{ij} &= 0 \text{ otherwise}
\end{align*}
\]

The constraints of the proctor assignment problem are as follows:
Max $f(x) = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$ \hspace{1cm} (1)

Max $g(y) = y$ \hspace{1cm} (2)

Subject to:

\[ \sum_{j=1}^{m} b_{ji} x_{ij} \leq \alpha_i \hspace{1cm} i = 1, \ldots, n \] \hspace{1cm} (3)

\[ \sum_{j=1}^{m} b_{ji} x_{ij} - a_i y \geq 0 \hspace{1cm} i = 1, \ldots, n \] \hspace{1cm} (4)

\[ \sum_{i=1}^{n} x_{ij} = \tau_j \hspace{1cm} j = 1, \ldots, m \] \hspace{1cm} (5)

\[ \sum_{j \in \mathcal{T}_i} x_{ij} \leq 1 \hspace{1cm} i = 1, \ldots, n; k = 1, \ldots, 2d \] \hspace{1cm} (6)

\[ x_{ij} \in \{0,1\} \hspace{1cm} i = 1, \ldots, n; j = 1, \ldots, m \] \hspace{1cm} (7)

\[ x_{ij} = 0 \hspace{1cm} j \in P_i \] \hspace{1cm} (8)

\[ y \geq 0 \] \hspace{1cm} (9)

The objective functions of the problem are given by equations (1) and (2). Equation (1) gives the sum of the preferences of the TAs’ assignments to exams and equation (2) depicts the minimum TA utilization which is given by the ratio of the hours that a TA is assigned to proctor exams over his/her available hours. The purpose of the PAP is to maximize these objective functions simultaneously.

Equations (3) till (9) represent the problems’ constraints. Constraint (3) makes sure that the number of assigned hours for each TA does not exceed his/her total available hours.

Constraint (4) determines the minimum TA utilization. Constraint (5) guarantees that each exam is assigned the needed number of TAs. Constraint (6) prevents a TA to proctor more than one exam during a given period. Constraint (7) enforces the binary
condition on the decision variable. Constraint (8) makes sure that proctors are not assigned to proctor exams that conflict with their own exams. Constraint (9) restricts the y-variable to positive values only.

Since we have two objective functions, we need to combine them into one weighted objective function as is the case with all multi-objective combinatorial optimization problems.

The weighted function for the proctor assignment problem is:

\[ h(x) = f'(x) + \alpha g(y) \]  

where

\[ f'(x) = \frac{\sum_{j=1}^{n} \sum_{j=1}^{m} c_{ij} y_{ij}}{\sum_{j=1}^{n} \sum_{j=1}^{m} t_{ij}} \]  

and

\[ c_{\text{max}}(f) = \max_{j} (1, c_{ij}) \]

\( f'(x) \) and \( g(y) \) are both bounded between 0 and 1 as the minimum preference value is zero. If we take \( \alpha \geq 0 \), we bias one of the terms of the combined objective function. If \( \alpha > 1 \), then assignments will be made in a way to make a uniform distribution of the workload. Alternatively, if \( \alpha < 1 \), then assignments will be made to maximize the preference values of TAs (Marti et al., 2000).

### 3.1.3 Solution Evaluation

When we solve the multi-objective optimization problem, we accept the solution that is not dominated. In other terms, if a feasible solution \( X \) is dominated by
another feasible solution $Y$ in a multi-objective optimization problem having $K$ objective functions, then we have $f_k(y) \geq f_k(x)$ for all objective functions and $f_k(y) \leq f_k(x)$ for at least one objective function. Furthermore, a feasible solution is said to be efficient if there is no other feasible solution that dominates it.

When we find several efficient solutions, we provide the decision-maker with several options and therefore, he would have the freedom to choose between various sets of good solutions (Marti et al., 2000).

### 3.2 Scatter Search Solution of the PAP

The scatter search methodology can be regarded as a population-based algorithm that constructs solutions by combining others in order to obtain better solutions. This methodology includes the following steps:

- Generate a population $P$
- Extract a reference set $R$
- Combine elements of $R$ and maintain and update $R$

Scatter search attempts to find improved solutions by combining solutions in the reference set $R$. These solutions are divided into high-quality and diverse solutions.

The scatter search methodology starts by generating a population $P$ by using a diversification generator to produce diverse solutions. Then the reference set $R$ is constructed by taking the best $r_1$ solutions in $P$ and the most diverse $r_2$ solutions in $P$. Afterwards, the reference set $R$ is maintained and updated by using the subset generation method and combining solutions in these subsets. If these solutions improve the quality of the ones in the reference set, they are added to the reference set $R$ (Marti et al., 2000).
When applied to assigning proctors to exams, the scatter search procedure consists of the following steps:

1. Read data
2. Preprocess the data in case we want to assign the TAs to the courses they teach
3. Change = True
4. Generate a seed solution
5. Apply the diversification generator
   5.1. Create the set $P$ of different solutions using the frequency generator
   5.2. Apply the improvement method to each solution and update the frequency function.
5. While (Change == True) do
   6.1. Construct the reference set $R$ considering the best $r_1$ solutions in $P$ and the most $r_2$ diverse solutions
   6.2. While (Subset counter < max subset) do
      6.2.1. Apply the subset generation method
      6.2.2. For each subset, use the solution combination method to get new trial solutions using a voting mechanism that takes into account the violation of the new solution with respect to constraints (3) and (6)
      6.2.3. Apply the improvement method to each trial solution. If this solution is not already in the reference set and it is a no-dominated solution then:
         6.2.3.1. Add this solution to $R$ and remove the solution which is dominated by this new solution
         6.2.3.2. Set subset counter = 0 and repeat from 6.2. while skipping subsets already analyzed
   6.3 If at least one no-dominated solutions in $R$ has changed:
      6.3.1 Build a new set $P$ considering the best $r_1$ no-dominated solutions currently in $R$ as an input to the diversification generator method in order to construct new solutions
   6.4 Else, Change = False

Figure 3.1 The steps for assigning proctors to exams using scatter search
In the next sections, the details of the above procedure will be discussed.

### 3.2.1 Diversification Generation Method

The diversification generator is used to construct an initial population of solutions. The generator that will be used aims at producing solutions using modified frequencies in order to get good quality and diverse solutions. The following frequency function will be used:

\[ f_y = \sum_{x \in P} x_y \]  

(13)

The role of this frequency value is to prevent the assignment of the same TA i to the same exam j in several solutions, and therefore to produce diversity in the new solutions being subsequently generated when compared with the ones that are already included on the population P.

As we have seen earlier, the objective of the problem is to maximize the preference of the TAs’ assignments to exams. The preference \( c_{ij} \) of TA i for exam j is a number between 0 and 5. This preference is calculated as the difference between the period in which exam j is supposed to take place and the period of the nearest exam that TA i must take. If this difference is greater than 5 periods, then it is set to 5. The period that directly precedes one of a TA’s exam is assigned a preference of 0.

In order to take into account the previous assignment of TA i to exam j, the preference \( c_{ij} \) is changed as follows:

\[ c'_{ij} = c_{ij} - \beta \left( \frac{\max_{t \neq i} c_{ij}}{\max_{t \neq i} f_y} \right) f_y \]  

(14)

As we can see, \( c'_{ij} \) is an adaptive function as it takes into account previous assignments of proctors to exams. In case the generator constructs the same solution
more than one time, we can change the value of $\beta$ to induce more diversification. For instance, we could start with $\beta=0.4$ and increase it by 0.1 every time the generator generates the same solution more than once.

The diversification generation method attempts to generate PopSize solutions using the modified preferences values $c'_{ij}$ (Marti et al., 2000).

The pseudo-code of this method is as follows:

```
Initialization
    Solutions = 0
    $f_{ij}=0$ for all $i,j$
While (Solutions < PopSize )
{
    For k=1 to 2d
    {
        Order the exams in period $k$ by decreasing number of required TA’s
        For each exam $j$ in period $k$
        {
            Construct the list of TA’s that can proctor $j$
            Order the list according to $c'_{ij}$
            Assign the first $t_j$ TAs to proctor exam $j$
            Update TA and exam information
        }
        Add the solution to the population
    }
    Make Solutions = Solutions + 1
    Update the corresponding $f_{ij}$
}
```

Figure 3.2 The pseudo-code for the diversification method
3.2.2 Improvement Method

The improvement method is used whenever we want to improve a given solution. This improvement is done by replacing one proctor from the TAs assigned to proctor an exam by another TA found by a simple local search in a 2-opt neighborhood.

In order to have solutions where the TAs have similar workload, the exchange is done by moving exams from TAs having a high workload to the ones with small workload.

This is done by accepting a solution which has a better objective function $f(x)$ and which does not decrease the value of $g(x)$. Therefore, if exam $j$ is to be changed from the exams list of TA $i$ to the exams list of TA $k$, then we accept this exchange in case $c_i < c_k$ and $g(y) \leq \frac{\sum_{j} b_j x_j - b_j}{a_i}$ which represents the utilization of TA $i$ when we remove an exam $j$ from the list of exams he has to proctor. When we exchange TAs to proctor a given exam, the utilization of TA $k$ will increase but the utilization of TA $i$ will decrease. In addition, if the improvement method detects that a TA is assigned to proctor more than one exam in the same period, then one of the exams is removed and assigned to another TA even if the solution does not improve (Martí et al., 2000).

For each proctor, calculate the available capacity

$$ACAP_i = a_i - \sum_{j} b_j x_j \quad i=1 \ldots n$$  \hspace{1cm} (16)

Let $g(x)$ be the minimum TA utilization value of the solution $x$

Order the proctors by ascending order of the available capacity. Let $i=1$

While $i \leq n$

\{ 

For each exam $j$ proctored by $i$ ($x_i,j=1$)
Check feasibility:
Find the proctor \( k \neq i \) with larger available capacity that can proctor exam \( j \)
If \( c_{ij} < c_{ki} \) and the new utilization value of \( i \) is \( \geq g(x) \), let
\[
\begin{align*}
x_i &= 0 \quad \text{and} \quad x_k = 1 \\
& \quad \text{Update ACAF} \\
& \quad i = 1 
\end{align*}
\]
Else, let \( i = i + 1 \)

Figure 3.3 The pseudo-code for the improvement method

### 3.2.3 Reference Set Update Method

After constructing a population \( P \), we need to extract from it a subset called the reference set \( R \) which contains high quality and diverse solutions.

We select high quality solutions and add them to the reference set if they are strongly dominated. The solutions in the reference set are ordered according to the value of the combined objective function \( h(x) \) while having the best solution as the one with higher \( h(x) \).

In order to evaluate the diversity of the solutions in the reference set, a distance function \( \partial(x', x^*) \) calculates the distance between two solutions \( x' \) and \( x^* \):

\[
\partial(x', x^*) = \sum_{i=1}^{m} \sum_{j=1}^{n} |x'_{ij} - x^*_{ij}|
\]  
(17)

It is to be noted that a large distance between two solutions does not mean that there is a large difference between their respective objective functions.

To construct the reference set \( R \) with \( |R| = r \), we first select the non-dominated solutions and add them to \( R \). Then, for each solution \( x \) in \( P-R \), we calculate the distance to all the elements in \( R \).
Let the minimum distance \( d_{\text{min}}(x) \) of a solution \( x \) in \( P-R \) to all solutions \( x \) in \( R \) as follows:

\[
    d_{\text{min}}(x) = \min_{x \in R} \{d(x, x')\} \tag{18}
\]

We then select the solution \( x^* \) having the maximum distance \( d_{\text{min}}(x) \) of all \( x \) in \( P-R \) until \( |R| = r \). In general, the size of the reference set is taken as \( r = 20 \).

As we have seen, a solution is only inserted in the reference set if it is no dominated in the strong sense or if it is diverse from the solutions already in the set (Marti et al., 2000).

### 3.2.4 Subset Generation Method

The scatter search algorithm attempts to find good solutions by combining the solutions found in the reference set.

First, subsets of solutions are generated, then they are combined into new solutions. Each new solution is compared with the worst solution already in the reference set \( R \). If it is found to be a better solution, it replaces the one that it dominates.

Subsets of the reference set are generated in a way to have useful properties while making sure that subsets previously generated are not repeated. These subsets are organized into four types as follows:

- **Subset Type 1**: all 2-element subsets

- **Subset Type 2**: 3-element subsets derived from the 2-element subsets by augmenting each 2-element subset to include the best solution not in this subset
• SubsetType 3: 4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solution not in this subset

• SubsetType 4: the subsets consisting of the best $i$ elements, for $i = 5$ to $r$

We need to keep in mind that the reference set is constantly changing as new solutions are added to it when they qualify to be among the $r$ best solutions and old ones are deleted from it (Marti et al., 2000).

3.2.5 Solution Combination Method

The solution combination method consists of taking each subset and applying a voting schema to it. In this voting, each solution votes for specific assignments of TAs to exams and the assignment with the largest vote is chosen as follows:

```plaintext
For each exam j
{
    Find the assignment $x_j$ with the largest vote
    Assign the exam j to TA i
}
```

Figure 3.4 The pseudo-code for the voting technique

The vote of the assignments $x_j$ is represents the merit of assigning TAs to exam $j$. Therefore, the vote considers the preference of the TAs for exams and it penalizes the assignments that result in violations of one or more of the problem constraints.
The overall voting process consists of \( m \) steps, \( m \) being the number of exams. At each step, a solution votes for its column \( x_j \) which represents all the TA assignments associated with exam \( j \). The vote of solution \( x \) at step \( j \) is given by the following formula:

\[
V_j(x) = \sum_{j=1}^{n} c_j x_j - \phi \left( \max \left( 0, \sum_{j=1}^{k} b_j y_j + b_j x_j - a_j \right) \right) - \gamma \left( \max \left( 0, \sum_{k \in E_j} \left( \sum_{y_y+y_x-1} \right) \right) \right)
\]

(19)

In this formula, the first term sums up the preferences of the TAs assigned to exam \( j \) in solution \( x \).

The second term adds the number of hours assigned to each TA in the solution that is being generated (represented by the \( y \)-variable), it also adds the number of hours for the current exam \( j \) and then subtracts the available hours for the TA. If the hours already assigned plus the hours related to the current exam exceed the available number of hours, then the excess hours are multiplied by a penalty factor in order to penalize the violation of constraint (3).

The third term multiplies the number of times a TA is assigned to proctor more than one exam during the same period by a constant in order to penalize violations of constraint (6).

If the newly generated solution violates constraints (3) and (6), then a repair procedure is performed as follows:

```
For each period \( k \)
{
    For each exam \( j \) scheduled in period \( k \)
    {
        If any TA assigned to \( j \) is also assigned to another exam in period \( k \) or his/her capacity has been exceeded
        {
            Find the next available TA that can be feasibly assigned
        }
    }
}
```
If during one scatter search iteration a new solution is inserted in the reference set, then the process can be repeated with the set of non-dominated solutions in the reference set knowing that a solution is inserted in the reference set if it is non-dominated in the strong sense. The rest of the population is generated using the diversification generator function and the scatter search process is repeated. Finally, the scatter search stops when there are no new solutions that are being added to the reference set (Marti et al., 2000).
Chapter 4

JAVA Design of the PAP using SS

4.1 Class Diagram

Figure 4.1 The Class Diagram for the PAP using SS
4.2 User Interface

![Software User Interface](image)

Figure 4.2 The software's user interface

4.2.1 Input Data

The data is conveyed into the software by using text files. The software reads those files and stores them into arrays. There are four types of files `exam.txt`, `pref.txt`, `ta.txt` and `tk.txt`.

The first file `exam.txt` contains data related to exams. The first line in the file contains the number of exams that exists in the file and the remaining lines represent data for each exam. For each exam there is the number of proctors needed, the month and the day when the exam is to be given and the starting and ending time of the exam.
<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>23</th>
<th>900</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>23</td>
<td>900</td>
<td>1100</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1130</td>
<td>1330</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>23</td>
<td>1130</td>
<td>1330</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>23</td>
<td>1500</td>
<td>1700</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1530</td>
<td>1730</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1730</td>
<td>1930</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1900</td>
<td>2100</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>25</td>
<td>900</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>25</td>
<td>1130</td>
<td>1330</td>
</tr>
</tbody>
</table>

Figure 4.3 Sample of the exam.txt input file

The second file `pref.txt` contains the preferences of each employee for each exam. The rows denote data for each proctor, while the columns contain the data related to each exam.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>..................</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>..................</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>-1</td>
<td>..................</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>..................</td>
</tr>
<tr>
<td>-1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>..................</td>
</tr>
</tbody>
</table>

Figure 4.4 Sample of the preference input file

The third file `ta.txt` includes data related to the teaching assistances. The first row has the number of teaching assistances ready to be assigned exams and have
their data present in the file. Each of the remaining rows has its data partitioned in the following manner. The first number holds the max number of available hours for each teaching assistant to proctor. The next number represents the number of exams the teaching assistant has to take. The third and the fourth number are the day and the month when the teaching assistance has an exams. The fifth and the sixth are the starting and ending time of the exam that the teaching assistant has to take. If a teaching assistant has more exams to attend, the related data will have the same format of format of third, fourth, fifth and sixth numbers and appended on the same row.

<table>
<thead>
<tr>
<th>7</th>
<th>24</th>
<th>2</th>
<th>6</th>
<th>23</th>
<th>1000</th>
<th>1200</th>
<th>6</th>
<th>25</th>
<th>1000</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1600</td>
<td>1800</td>
<td>6</td>
<td>25</td>
<td>1000</td>
<td>1200</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>1</td>
<td>6</td>
<td>25</td>
<td>1700</td>
<td>1900</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>1</td>
<td>6</td>
<td>25</td>
<td>1000</td>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1000</td>
<td>1200</td>
<td>6</td>
<td>26</td>
<td>1000</td>
<td>1200</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>1</td>
<td>6</td>
<td>26</td>
<td>1600</td>
<td>1800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>2</td>
<td>6</td>
<td>23</td>
<td>1000</td>
<td>1200</td>
<td>6</td>
<td>25</td>
<td>1000</td>
<td>1200</td>
</tr>
</tbody>
</table>

Figure 4.5 Sample of the TA input file

The fourth file tk.txt reveals the distribution of the exams over their corresponding period. The first row represents the number of periods available, while each of the remaining rows show the exams that are assigned to the same period.

<table>
<thead>
<tr>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 1 4 2 3</td>
</tr>
<tr>
<td>4 5 6 7 8</td>
</tr>
<tr>
<td>3 9 10 11</td>
</tr>
</tbody>
</table>
4.2.2 Output Data

The output form this software is going to be saved in a text file called solution.txt. It will contain sets of records each set will have the following identifier at the first row $x = \text{number}$, and the rest of the group will show each proctor the exams that are assigned. The rows are the exams and the columns are the exams.

Figure 4.6 Sample of the TK input file

Figure 4.7 Sample of the Solution output file
4.3 Diversification Generator Class

This class is used to generate diversified solutions that are generated in a controlled randomization manner. The class implements a frequency-based memory algorithm. This class is called by the 'LongTask Class'. In turn it references 2 classes and they are the improvement class and the utilities class.

In this class we generate \textit{popsiz}e number of solutions. The \textit{popsiz}e is assigned a value of twenty. Each solution is produced by going through 'tk' array. The tk array is sorted by the decreasing number if required \textit{ta}. The criteria to select a \textit{ta} to proctor an exam is that he must have the greater preference, his remaining hours must be greater than zero and has must not have a conflict with an exam he is taking.

After finishing the solution we must check if the solution is not generated before. The new solution is added after passing it to the Improvement class. The preference used above is derived according the formula (14).

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Diversification Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>\textit{i}</td>
<td>number of teaching asstas.</td>
</tr>
<tr>
<td>\textit{j}</td>
<td>number of exams.</td>
</tr>
<tr>
<td>\textit{d}</td>
<td>number of exam days.</td>
</tr>
<tr>
<td>\text{exams}</td>
<td>array of exams related information.</td>
</tr>
<tr>
<td>\text{tk}</td>
<td>array containing the distribution of the exams over the different periods.</td>
</tr>
<tr>
<td>\text{pref}</td>
<td>array of the initial preference table of teaching asstas.</td>
</tr>
<tr>
<td>\text{ta}</td>
<td>array of teaching asstas related information.</td>
</tr>
<tr>
<td>\text{popSize}</td>
<td>desired population size.</td>
</tr>
<tr>
<td>\text{beta}</td>
<td>variable used to hold the value that encourages diversification.</td>
</tr>
<tr>
<td>\text{betaInterval}</td>
<td>variable used to increment the \text{beta}.</td>
</tr>
<tr>
<td>OutPut</td>
<td></td>
</tr>
<tr>
<td>\text{x}</td>
<td>array that contains the diversified solutions.</td>
</tr>
</tbody>
</table>

\textbf{void mainClass()}

begin method
do a while loop as long as the solution is less than \text{popSize}
begin
build an array 'hrsRemaning' that contains the number of hours
each ta has remaining;
go through each period k until reaching 'd'
  begin
    assign the variable 'util' to be of type 'Utilities'
    order the exams in period 'k' by decreasing number of
    required TAs using 'util.sortArray()'
    and put the values
    in 'sortedArray';
    go through each exam of the current period 'k'
    begin
      get the 'numberOfProctorsNeeded';
      construct the list of Ta's that can proctor
      the current exam and assign it to array 'subTA'
      each proctor must have no exams assigned in
      the current period and
      have preference > -1;
      sort array subTA and assign it to
      'sortedTasCanProctor'
    using the
    'util.bubbleSort'
    method;
      assign the first TA's to proctor exam j
      with each proctor assigned decrease the
      'numberOfProctorsNeeded';
      decrease the number of hours available for the
      proctor in the
      'hrsRemaining';
      use the function 'util.getDifferenceHours' to
      know the number of
      hrs the exams needs;
      update the x array;
      end go
  end go
if the current solution is not different (using
util.differentSolution method)
  Then
  improve the solution using the improvement variable
  and assign the new solution to x;
  update f array but adding the previous value of each
  entry to the current solution found
improved
  increment solution by 1;
else
  increment beta by betainterval;
end if
calculate c'
go through every i
  begin
    go through every j
      begin
if \( f > 0 \) then
    \( c' = \text{previous } c - (\beta \cdot \text{maxf} \cdot f) \)
    if \( c' < -1 \) then
        \( c' = -1 \)
    end if
else
    \( c' = c \)
end if
pref2 = c'
end for

end method

getMinTaUtilization()
begin method
    return this.improvement.getMinTaUtilization().
end method

getPeriod(examIndex)
begin method
    search the tk table to find the exam's period;
end method

getPreference ()
begin method
    return pref2;
end method

getX()
begin method
    return x;
end method

noExamsAtSamePeriod(solutionIndex, taIndex, examIndex)
begin method
    search in the x table of the solutionIndex for a value = 1
    for the taIndex, and compare it to the period of
    examIndex.
    If matching periods return false else return true.
end method

Figure 4.8 Diversification Generator's pseudo-code

4.4 Improvement Class

As the name implies this class enhance the produced solution by the diversification class and the combination class.
We go through every ta to get the next ta that has a larger capacity, check their preferences such that the new ta's must be greater than the old. In addition the new utilization of the new ta must be greater than the minimum utilization of the solution. The formulas used are (15) and (16).

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td></td>
</tr>
<tr>
<td>solution:</td>
<td>number of solutions passed</td>
</tr>
<tr>
<td>i:</td>
<td>number of teaching assistances</td>
</tr>
<tr>
<td>x:</td>
<td>array that contains the solutions.</td>
</tr>
<tr>
<td>exams:</td>
<td>array of exams related information.</td>
</tr>
<tr>
<td>ta:</td>
<td>array teaching assistances related information.</td>
</tr>
<tr>
<td>pref:</td>
<td>array of the preference of the teaching assistances</td>
</tr>
<tr>
<td>tk:</td>
<td>array containing the distribution of the exams</td>
</tr>
<tr>
<td>over the</td>
<td>different periods.</td>
</tr>
<tr>
<td>solutionIndex:</td>
<td>the index of the solution to be improved.</td>
</tr>
</tbody>
</table>

| **Output** | array that contains the diversified solutions. |

```java
void mainClass()
begin method
    display a message in the monitor of the interphase "Performing Improvement";
    int s = this.solutionIndex; //
go through every ta
    begin
        go through every exam and calculate the hours the
        teaching assistant has to proctor and assign it to 'sum_b_x';
        calculate the available capacity and assign it to the 'ACAP'
        array using the formula 'ACAP' = available hours that the
        teaching assistant have - 'sum_b_x';
        calculate the ta utilization 'gOfY' = 'sum_b_x' / teaching
        assistance hours to proctor;
        if 'gOfY' < minimum utilization of the solution then 'gOfX' =
        'gOfY'
        sort the 'ACAP' table using the method 'util.sortArray30' |
        int p=0;
do while loop for every proctor
    begin
        go through every exam that the ta proctors
        begin
            // check feasibility
            get the next ta that has a larger ACAP using
```
getLargerACAP();
get the utilization using the method
getNewUtilization
();
if the preference of the current ta is less than
the preference of
the new ta and the
new utilization is
greater than the current
utilization then
give the exam to the new
ta;
update the ACAP table for both tas using
getNewACAP(s,p);
sort the ACAP table using util.sortArray3();
end if
go
doon end method

getImprovedSolution()
begin method
return x;
end method

getLargerACAP()
begin method

  go through every ta
  get the next ta that does not proctor the same exam
  and has larger ACAP and not assigned
  to another exam in the same period ;

end method

getMinTaUtilization()
begin method
return gOfX;
end method

getNewUtilization()
begin method

  go through exams excluding
  the current exam and get
  the number of hours the ta proctors and assign it to 'sum_b_x'
  calculate the ta utilization using the following formula 'sum_b_x' / ta's
  max number of hours available for TA;

end method

getPeriod()
begin method
    go through the tk array and get the periods the exam is
    assigned to;
end method

noExamAtSamePeriod()
begin method
    go through each exam that the ta has and check if he has an
    exam in it other than the one being assigned to;
end method

Figure 4.9 Improvement method’s pseudo-code

4.5 Reference Set Update Class

Using this class a subset is of the general population set \( P \) is produced. The produced subset contains diverse and high quality solutions. 10 diverse solutions are generated by this class. This class is called by LongTask class.

The 10 diverse solutions are selected using the distance formula (17). The solutions with the minimum value produced by the above formula are chosen.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>ReferenceSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>number of teaching assistances</td>
</tr>
<tr>
<td>j</td>
<td>number of exams</td>
</tr>
<tr>
<td>solutions</td>
<td>number of expected solutions</td>
</tr>
<tr>
<td>x</td>
<td>array that contains the solutions.</td>
</tr>
<tr>
<td>referenceSet</td>
<td>an array containing the solution index of the reference set along with its</td>
</tr>
<tr>
<td></td>
<td>sortedSubhO(X) the number of</td>
</tr>
<tr>
<td>remainingSolutions</td>
<td>array containing number of the remaining solutions</td>
</tr>
<tr>
<td>numberOfBestSolutions</td>
<td>number of best solutions</td>
</tr>
<tr>
<td>numberOfDiverseSolutions</td>
<td>number of diverse solutions</td>
</tr>
</tbody>
</table>

OutPut
referenceSet : array containing the solution index along with \( h(x) \)

void mainClass()
begin method
display a message in the update monitor "+Generating Reference Set";
initialize minDistances;
go for all number of diverse solutions
    begin
        initialize the 'minDistances' array to be -1
        go for solutions
            if remainingSolutions >= 0 then
                sumMin = 0;
                go through the referenceSet
                sum = 0;
                go through all of the tas
                    go through all exams and sum
                        the distances between the
                        remaining solution and the
                        current x using the formula
                        sum = sum + Math.abs(x-x);
                end go
                if sumMin is greater than sum then
                    sumMin = sum;
                end if
                end go
                assing the values of the distance to the
                minDistance array which are the remaining
                solution index, minSum;
            end if
        end go
        go through minDistances array and choose the max values then add it
to the referenceSet
    end go
end method

getReferenceSetArray ()
begin method
    return this.referenceSet;
end method

Figure 4.10 Reference set generator’s pseudo-code

4.6 Subset Generation Class

This class generates 4 types of subsets from the reference set. It is called by the
LongTask class. Each of the 4 sets is a 2 dimensional array where each row contains
2 or more reference to the indexes of solutions in the reference set. The type 1 subset
must contain 190 elements i.e. array of 190x2 dimensions. The type 2 subset is an
array of 171x3 elements. The type 3 is an array of 153x3 elements. Finally type 4 is
an array of 16x20 elements. These calculations are based on reference set of 20
elements. The algorithms that produce these sets are described in the pseudo below.

<table>
<thead>
<tr>
<th>Class Name: Subset Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
</tr>
<tr>
<td>numberOfRefSetElements : contain the number of expected solutions</td>
</tr>
<tr>
<td>Output</td>
</tr>
<tr>
<td>subSetType1: Array containing combination of type 1</td>
</tr>
<tr>
<td>subSetType2: Array containing combination of type 2</td>
</tr>
<tr>
<td>subSetType3: Array containing combination of type 3</td>
</tr>
<tr>
<td>subSetType4: Array containing combination of type 4</td>
</tr>
</tbody>
</table>

getSubSetType1 ()
begin method
  display a message in the update monitor "subsetup type 1";
  go starting from counter1 = 0 to the numberOfRefSetElements
  begin
    go starting from counter2 = the above counter + 1
    begin
      and while the counter2 is less that numberOfRefSetElements
      begin
        subSetType1 [rowIndex][0] = counter1;
        subSetType1 [rowIndex][1] = counter2;
        increment rowIndex;
      end go
    end go
  end method

getSubSetType2 ()
begin method
  display a message in the update monitor "subsetup type 2";
  go starting from counter1 while counter1 is less than
  numberOfRefSetElements
  begin
    go starting from counter2 = counter1+1 while counter2 is less
    than numberOfRefSetElements
    begin
      sumSubSetType2Temp [rowIndex][0] = 0;
      sumSubSetType2Temp [rowIndex][1] = counter1;
      sumSubSetType2Temp [rowIndex][2] = counter2;
      increment rowIndex by 1;
    end go
  end go
create a new array subSetType2 and
assign the values from
sumSetType2Temp this is done in order
to remove unused memory array
  elements;
end method

getSubSetType3 ()
begin method
display a message in the update monitor "subsetup type 3";
  numberOfRows = numberOfRefSetElements - 4;
  endingColumn = 4;
  go starting from counter1 = 2 and while counter1 is less than
    endingColumn ++;
  begin
    go starting from counter2 = counter1 + 1 and while counter2 is
      less than numberOfRefSetElements
      begin
        subSetType3Temp [rowIndex][0] = 0;
        subSetType3Temp [rowIndex][1] = 1;
        subSetType3Temp [rowIndex][2] = counter1;
        subSetType3Temp [rowIndex][3] = counter2;
        increment rowIndex by 1;
      end go
  end go
create a new array subSetType3 and assign the values from
  sumSetType3Temp this is done in order to remove unused memory
  array elements;
end method

getSubSetType4 ()
begin
  display a message in the update monitor "subsetup type 4";
  go starting from counter1 = 0 and while counter1 less than
    numberOfRows;
  begin
    increment endingColumn;
    go starting counter2 = 0 and while counter2 less than
      endingColumn
      begin
        subSetType4 [counter1][counter2] = counter2;
      go starting counter3 = counter2 while counter3 less
        than numberOfRefSetElements
        begin
          subSetType4 [counter1][counter3] = -1;
        end go
      end go
  end go
end method

Figure 4.11 Subset generation's pseudo-code
4.7 Solution Combination Class

In this class I have implemented the logic that will produce a new solution out of the solutions that are referenced to in each row from the sub sets. This class is called by the LongTask class. The algorithm for combining a number of solutions is performed by comparing the votes calculated (using formula 19) from columns representing the tas that proctor the same exam from each solution. The column that has the highest vote is selected to be the corresponding exam column in the new solution. After that the newly produced solution is passed to the repair method. This method repairs the solution if it violates either equation (3) or (6). The next step is to perform improvement on the combined new solutions. As a last step this class selects 20 solutions out of the 4 subsets and the original reference set. The selection is based on the h(x). The solution that has the higher h(x) is selected. In the end a text file solution.txt is produced.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Solution Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>subSet1</td>
<td>array of subsettype 1</td>
</tr>
<tr>
<td>subSet2</td>
<td>array of subsettype 2</td>
</tr>
<tr>
<td>subSet3</td>
<td>array of subsettype 3</td>
</tr>
<tr>
<td>subSet4</td>
<td>array of subsettype 4</td>
</tr>
<tr>
<td>x</td>
<td>array that contains the solutions</td>
</tr>
<tr>
<td>ta</td>
<td>array teaching assistances related information</td>
</tr>
<tr>
<td>referenceSet</td>
<td>array containing the reference set alone with the H(x)</td>
</tr>
<tr>
<td>pref</td>
<td>array of the preference of the teaching assistances</td>
</tr>
<tr>
<td>exams</td>
<td>array of exams related information</td>
</tr>
<tr>
<td>tk</td>
<td>array containing the distribution of the exams over the</td>
</tr>
<tr>
<td>alpha</td>
<td>a factor used in a improvement method</td>
</tr>
</tbody>
</table>

| Output     |                      |
| finalReferenceSet | array containing the final solution result. |

void mainClass()
    begin method
display a message in the update monitor "+Generating combinations";
display a message in the update monitor " subType 1";
go through the 'solutionSubSet1'
   by taking each pair and perform
   voting using the method 'performVotingForSubSet1'
   and assign the value to solutionsFromSubSet1;
find the h(x) for the of the solution of subtype1 and assign them
to hOfXSubSet1;
go through the 'solutionSubSet2'
   by taking each triplet and perform
   voting using the method 'performVotingForSubSet2'
   and assign the value to solutionsFromSubSet2;
find the h(x) for the of the solution of subtype2 and assign them
to hOfXSubSet2;
go through the 'solutionSubSet3'
   by taking each quadruplet and perform
   voting using the method 'performVotingForSubSet3'
   and assign the value to solutionsFromSubSet3;
find the h(x) for the of the solution of subtype3 and assign them
to hOfXSubSet3;
go through the 'solutionSubSet4'
   by taking elements starting by 5 elements incrementing
   by 1 till 20 elements and perform voting using the
   method 'performVotingForSubSet4' and assign the value
   to solutionsFromSubSet4;
find the h(x) for the of the solution of subtype4 and assign them
to hOfXSubSet4;
sort the referenceSet using the method util.bubbleSort();
   // use a new data structure in order to compare
the generated
   // combinations with the old reference set ....
   // it is a 2 dimension array ....
   // index 0 pointer to the array from which we chosen the
solution
   // if index 0 = 0 get it from the old reference set
   // if index 0 = 1 get it from subtype1
   // if index 0 = 2 get it from subtype2
   // if index 0 = 3 get it from subtype3
   // if index 0 = 4 get it from subtype4
   // index 1 points to the index of the solution in the appropriate
   //array
   // index 2 points to h(x) of the corresponding solution
initialize the finalReferenceSet to the values of the current
solution perform the last stage using the method
handleLastStage() for each of the subSet1 till 4;
write the solutions in a text file;

end method

void handleLastStage ()
begin method
display a message in the update monitor " updating the reference set";
go through each element in the subSet passed
begin
display a message in the update monitor " checking differentSolution2";
if the solution in the subset is not present in the reference set
then
display a message in the update monitor " in differentSolution2";
go through each solution in the refereceset starting from below
if H(X) of the current solution is greater than
finalReferenceSet entry
Then
put the current solution in the final Reference Set;
end if
end go
end if
end go
end method

performVotingForSubSet1()
begin method
go through each each exam
begin
get the vote for each exam using the method getvotePerExam()
and save the value in arrayToFindBiggest;
get the 'solutionIndex' of the biggest Vote using the util.findTheBiggest(arrayToFindBiggest);
fill the 'newX' array with the biggest solution;
end go
repair the newX solution using the method solutionRepair();
perform improvement using the method improvement.getImprovedSolution();
return newX;
end method

performVotingForSubSet2()
begin method
same algorithm as the previous method
end method

perform VotingForSubSet3()
begin method
same algorithm as the previous method
end method

perform VotingForSubSet4()
begin method
same algorithm as the previous method
end method

getVotePerExam()
begin method

// for each ta
get 'periodOfExam' using the method
getPeriodOfExam(examIndex);
initialize vote;
go through the all the tas
begin
bigTerm1 = 0;
bigC = pref of this ta and this exam
bigX = x of the current solution;
// calculate term1
bigTerm1 = bigC multiplied by bigX;
bigPhy = 0.1;
bigGamma = 0.5;
bigSumInTerm2 = 0;
bigTerm2 = 0;
bigA = max available hours of the current ta from ta
table;
// calculate term2
go through all the exams before the current
begin go
bigB = number of proctors needed from exams;
bigY = of the current solution;
bigSumInTerm2 = bigSumInTerm2 + (bigB
multiply bigY);
end go
bigB = number of hours of the current exam;
bigTerm2 = bigSumInTerm2 + (bigB multiply (bigB
multiply bigX));
bigTerm2 = bigTerm2 subtract bigA;
bigTerm2 = (max of (bigTerm2,0D)) multiply bigPhy;
// this sum is for the exams that span over more than
// one period
sumBigTerm3 = 0;
go through all the periods that the current exam spans

over
begin
  sumY = 0;
  bigTerm3 = 0;
go through the exams starting from the first one
till the one before the current
begin
  if the exam is in the period of the above
  exam using using the 'getInPeriod'
  then
    bigY = newX[of the current
    ta[of t the current exam];
    sumY = sumY add bigY;
  end if
end go
bigTerm3 = (sumY add bigX) subtract (1));
sumBigTerm3 = sumBigTerm3 add (bigTerm3);
end go
sumBigTerm3 = max (sumBigTerm3,0);
sumBigTerm3 = sumBigTerm3 multiply bigGamma;
vote = vote add (bigTerm1 subtract (bigTerm2 subtract
sumBigTerm3));
end go
return vote;
end method

getPeriodOfExam()
begin method
return the periods of a specific exam;
end method

getInPeriod()
begin method
returns true if an exam in the same period specified;
end method

solutionRepair()
begin method
go through every for every period
begin
  go through every entry in each period
  begin
    go through every ta
    begin
      if a ta is scheduled for more than one
      exam in the same period and he has
      exceeded his number of hours that he has
      then
      go through the tas starting from the next
      one

58
begin
    if the ta is not assigned the exam
        then assign the exam to
            the ta if formulas (6) and
            (3) and is preferred by the ta
    end if
end go
end go
end method

Figure 4.12 Combination method's pseudo-code

4.8 Overall Procedure

This is the main class that calls all of the above mentioned class. It starts by calling
the ReadInputClass that reads the text files. The next class that is activated is the
diversification class. After that the ReferenceSet will be activated. The last two
classes to be run are the SubSetGeneration and the Combination.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>LongTask</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td></td>
</tr>
<tr>
<td>exams</td>
<td>array of exams related information.</td>
</tr>
<tr>
<td>tic</td>
<td>array containing the distribution of the exams over the</td>
</tr>
<tr>
<td>ta</td>
<td>array containing the teaching assistances related</td>
</tr>
<tr>
<td>information</td>
<td></td>
</tr>
</tbody>
</table>

**Output**

*ActualTask()

begin
    read inputfiles using the method ReadInputClass.newRead;
    if cancel button was pressed displayMessage in the massage pane using the
    method Utilities.displayMessages "Process was canceled";
else
    fill the exam array using the method
        ReadInputClass.getExams();
    fill the ta array using the ReadInputClass.getTK();
    initialize diversification with the following values ( |
        ta.length, |
        exams.length, |
perform the diversification method using
Diversification.mainClass();
'x' array = Diversification.getX();
solutions = x.length;
get the number of tas 'i' = x[solutions-1].length;
get the number of exams 'j' = x[this.solutions-1][i-1].length;
get the 'gOfX' array using
'Diversification.getMinTaUtilization()';
get the 'pref' array using 'Diversification.getPreference()';
find the H(x) for each solution and assign it to the 'hOfX'
array using the method Utilities.getHofX();
sort the solutions using Utilities.bubbleSort() in descending
order of h(x) then assign the values to
'sortedSubhOfX';
create the referenceSet by first filling it with the best solutions
from 'sortedSubhOfX'.
perform the ReferenceSet.mainClass();
get the ReferenceSet.getReferenceSetArray() and assigned it
to the referenceSetArray;
get subsettype1 using the method
SubSetGeneration.getSubSetType1();
get subsettype2 using the method
SubSetGeneration.getSubSetType2();
get subsettype3 using the method
SubSetGeneration.getSubSetType3();
get subsettype4 using the method
SubSetGeneration.getSubSetType4();
initiate the combination class and call
CombinationMethod.mainClass();

end if
method end

void go()
begim method
run actual class
end method

Figure 4.13 The Long Task's pseudo-code
4.9 Utilities Class

This class contains methods that are repeatedly accessed by all classes.

```
Class Name : Utilities

// a method to get the difference in hours between two given hours
int getDifferenceHours(String str1, String str2)
begin Method
    initialize a calendar variable c1 and assign to it the time from str1
    initialize a calendar variable c2 and assign to it the time from str2
    return the difference between c1 and c2 in hours;
end method

// a method to append a zeros at the beginning of a numeric string
String setFullField(String str1)
begin method
    if str1 less than 3 append then
        '0' at the beginning of the string
    end if
    return the new string;
end method

// Purpose a method to check if the produced solution is not in the reference set
boolean differentSolution(int lastSolution, int[][], int x, int i, int j)
begin method
    go through each solution in the reference set
    check for each value in the current solution with the solution array to test if difference values then
        go to the next solution
    increment numberOfNotEqualSolution
    end if
go
if numberOfNotEqualSolution < Total Number of solutions
    return false
else
    return true
end if
end method

// It is a method to check if the produced solution is not present in the
// reference set ....
// It is used in the combination method. May be i must do optimization over
// here.
// return returns true if the solution is not found in the reference set
boolean differentSolution2(double[][], referenceSet,
```
int [] subSet,
int [] subSet2,
int [] subSet3,
int [] subSet4,
int [] x,
solutionToCheck;
solutionData;
begin Method
go through each solution in the subsets. Here according to the variable solutionToCheck we use a specific subset
if difference values then
go to the next solution
increment numberOfNotEqualSolution
endif
go if numberOfNotEqualSolution < Total Number of solutions
return false
else
return true
endif
end method

// a method to find h(x)
double [][] getAllOfHOFX(
    double [][] pref,
    int [][] solutionsFromSubSet1,
    int [] exams,
    int [] ta,
    int maxFractionDigits,
    int alpha)
begin method
go through solutionsFromSubSet1
get the max utilization 'gOY' using the method getGOFY();
assign to hOfy[y][0] the solution index y;
assign to hOfy[y][1] the value of getHofX();
end go
return hOfy;
end method

// a method to find H(x). Note here that alfa = 10
def HofX(
    double [][] pref,
    int [] x,
    int [] exams,
    int maxFractionDigits,
    int alfa,
    double gOFY)
begin method
set i to pref.length;
set j to pref[i-1].length;
set answer to 0;
set subAnswerNu to 0;
set sumAnswerNu to 0;
set sumAnswerDe to 0;
set fPrimeOfX to 0;
// find the numerator
go through all of the tas
  go through all of the exams
    subAnswerNu = (current preference) multiply (value of the current solution);
    set e max of j = l
go through all tas
    if current pref > e max of j then
      e max of j = current preference;
    end if
  end go
  set subAnswerNu to subAnswerNu divide e max of j ;
  set sumAnswerNu to sumAnswerNu add subAnswerNu;
end go

// find the denominator
go through all exams
  set the values sumAnswerDe to sumAnswerDe add number of proctors of the current exam;
end go
// here answer is f(x)
set fPrimeOfX to be sumAnswerNu divide by sumAnswerDe;
set initialy answer to be gOfY;
set answer to answer multiply by alfa;
set answer to answer add fPrimeOfX;
return answer;

end method

// A method to calculate the formula (15) ta utilization
getGOFY (int[][]) exams,
   int [][] x,
   int [][] ta,
   int maxFractionDigits)
begin method
assign gOfX to 0;
go through every ta
  set the value of sum_b_x to 0;
  set the value of gOfY to 0;
go through every exam
  set the value of sum_b_x to sum_b_x after adding the number of hour the exam multiplied by the current solution value;
return gOfX;
end method
end go through
// calculate the ta utilization
set the value of gOfY to sum_b_x / max number of hours the ta has;
if (gOfX is greater than gOfY and gOfY is greater 0) or (gOfX equals 0))
then
  set gOfX to be gOfY;
end if

end go
return gOfX;
end method

// a modification of the famous bubbleSort here 2 values are being changed
// instead of one
double [][] bubbleSort(double [][] subArray)
begin method
end method

// a modification of the famous bubbleSort here 3 values are being changed
// instead of one
double [][] bubbleSort2(double [][] subArray)
begin method
end method

// a sort technique used by the diversification method. Different
// than the others because it reverses the order and it sorts 2 arrays.
int [] sortArray(int [] tksub, int [][] exams)
begin method
  return tksub;
end method

// a sort technique used by the improvement class.
double [][] sortArray3(double [][] arrayToSort)
begin method
  return arrayToSort;
end method

// a method to find the biggest value in an array
// returns the value and index in the table
int findTheBiggest (int [] subArray)
begin method
end method

// This method is used to verify formula (3), i.e. returns true is the assigned
// hours for a ta did not exceed the max number of hours a ta has.
boolean constraint3ForASingleTa(int i,
                      int j,
                      int [] [] x,
                      int [][] exams,
                      int [][] ta,
                      int indexOfTa)
begin method
    set the sumOfHoursAssigned to 0;
    go through all exams
        if the ta is assigned the current exam
            then
                increment sumOfHoursAssigned by the number of
                hours the current exam need
            end if
        end go
    if the number of hours assigned to a ta is greater than the number max
    of hours this ta has
        then
            return false;
        else
            return true;
        end if
end method

// Check of a ta proctors an exam at the same period
boolean constraint6(  int [][] tk,
                      int indexOfTk,
                      int [] [] x,
                      int indexOfTa,
                      int indexOfCurrentExamToCheck)
begin method
    go through all of the exams in a certain period
        if the current exam is assigned to this ta and different than the
        exam that we are checking against
            then
                foundAnother = true;
            end if
    return foundAnother;
end method

// a method to display messages in the output monitor
void displayMessages(String st)
begin method
end method

// a method to display messages in the task monitor
void updateMonitor(String st)
begin method
end method

Figure 4.14 The pseudo-code for the utilities class
Chapter 5

Empirical Results

Professor Helena Lourenco contributed to this project by providing input data files. Those files contain the same data that was used in the original conference paper, and were samples of real occurrence at the Pompeu Fabra University in Barcelona, Spain of the proctor allocation dilemma. In order to verify the results from the implemented algorithm I had to manually go through each line of code and compare the results with those produced by the computer. Moreover I included in my comparison the results that were found by professor Helena.

My test collection consisted of 6 scenarios, and the resulted data could be viewed in Table 1. The first three columns represented the scenario's number, the number of teaching assistants and the number of exams respectively. Column four represented the result obtained from the software that was made by Professor Helena and her colleagues. As for column five it showed the results that was produced by my code. Finally the last column contained data obtained after using the Cplex version 6.5 software that could be used to solve mixed integer problems.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>TA's</th>
<th>Exams</th>
<th>Scatter Search by Helena</th>
<th>Scatter Search by Me</th>
<th>Cplex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>21</td>
<td>0.764</td>
<td>0.9894</td>
<td>0.953</td>
</tr>
<tr>
<td>2</td>
<td>59</td>
<td>52</td>
<td>1.000</td>
<td>1.017</td>
<td>1.393</td>
</tr>
<tr>
<td>3</td>
<td>59</td>
<td>44</td>
<td>1.000</td>
<td>1.012</td>
<td>1.165</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>21</td>
<td>0.754</td>
<td>0.9469</td>
<td>0.815</td>
</tr>
<tr>
<td>5</td>
<td>46</td>
<td>42</td>
<td>0.952</td>
<td>0.9612</td>
<td>1.268</td>
</tr>
</tbody>
</table>
After looking at table 1 the results obtained using the SS by Helena were lower than that obtained by the Cplex. On the other hand the figures in column SS by me were of better values than those of 'SS by Helena', even in the case of example 1 they were better than the Cplex's.

The 'SS by me' program was coded using the Java programming language and was executed on a Mobil AMD Duron system running at a speed of 1 GHz.
Chapter 6

Conclusion and Future Work

The main advantage of using the Scatter Search technique is that it can produce high quality solutions from which the person in charge can select the most appropriate one. This flexibility is highly appreciated when the decision is based on subjective constraints as in the problem of assigning proctors to exams where the proctor expresses his personal preference to the exams.

It has been noted that scenario 1 gave a better result than that of the Cplex software. Therefore to induce a better validation of the result it is very important to implement a database structure in order to control the results visually and at each stage.

Moreover I have noticed this implementation needed a more powerful computer since in some scenarios such as example 5 it took more than 1 hour to reach the final results.

Also I encourage the possibility of acquiring the Cplex software because it can be used for working out any IP models, and to assist in further development of the assigning teaching assistance to exams using the scatter search algorithm.

In the end I have to express my willingness to participate in any further development to my current implementation of the algorithm.
References


