Genetic Algorithms For Nurse Duty Rostering

3rd year BEng project

2004

Philip Howson
(ph201)

Supervisor: Dr Istvan Maros
Second Marker: James Jacobson
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>6</td>
</tr>
<tr>
<td>4.2</td>
<td>7</td>
</tr>
<tr>
<td>4.3</td>
<td>8</td>
</tr>
<tr>
<td>4.4</td>
<td>8</td>
</tr>
<tr>
<td>4.5</td>
<td>10</td>
</tr>
<tr>
<td>4.6</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>5.1</td>
<td>13</td>
</tr>
<tr>
<td>5.2</td>
<td>13</td>
</tr>
<tr>
<td>5.3</td>
<td>14</td>
</tr>
<tr>
<td>5.4</td>
<td>17</td>
</tr>
<tr>
<td>5.5</td>
<td>19</td>
</tr>
<tr>
<td>5.6</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>6.1</td>
<td>24</td>
</tr>
<tr>
<td>6.2</td>
<td>27</td>
</tr>
<tr>
<td>6.3</td>
<td>28</td>
</tr>
<tr>
<td>6.4</td>
<td>32</td>
</tr>
<tr>
<td>6.5</td>
<td>38</td>
</tr>
<tr>
<td>6.6</td>
<td>41</td>
</tr>
<tr>
<td>6.7</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>49</td>
</tr>
<tr>
<td>7.3</td>
<td>50</td>
</tr>
<tr>
<td>7.4</td>
<td>53</td>
</tr>
<tr>
<td>7.5</td>
<td>55</td>
</tr>
<tr>
<td>7.6</td>
<td>57</td>
</tr>
<tr>
<td>7.7</td>
<td>60</td>
</tr>
<tr>
<td>7.8</td>
<td>62</td>
</tr>
<tr>
<td>7.9</td>
<td>63</td>
</tr>
<tr>
<td>7.10</td>
<td>69</td>
</tr>
<tr>
<td>8</td>
<td>71</td>
</tr>
<tr>
<td>8.1</td>
<td>71</td>
</tr>
<tr>
<td>8.2</td>
<td>88</td>
</tr>
<tr>
<td>8.3</td>
<td>89</td>
</tr>
<tr>
<td>8.4</td>
<td>90</td>
</tr>
<tr>
<td>9</td>
<td>92</td>
</tr>
<tr>
<td>9.1</td>
<td>92</td>
</tr>
<tr>
<td>9.2</td>
<td>98</td>
</tr>
<tr>
<td>9.3</td>
<td>101</td>
</tr>
<tr>
<td>9.4</td>
<td>102</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>References</td>
</tr>
<tr>
<td>11</td>
<td>Appendices</td>
</tr>
<tr>
<td>11.1</td>
<td>Appendix 1: Experimental conditions for mutation testing</td>
</tr>
<tr>
<td>11.2</td>
<td>Appendix 2: Test Conditions for thread testing</td>
</tr>
<tr>
<td>11.3</td>
<td>Appendix 3: Hard Constraints Implementation</td>
</tr>
<tr>
<td>11.4</td>
<td>Appendix 4: Soft Constraints Implementation</td>
</tr>
</tbody>
</table>
2 Acknowledgements

I would like to thank Dr Istvan Maros for his help and advice with this project.

I would also like to thank the following people and organisations for their help and time with testing and specifying this project:

- Pauline Howson
  Nursing Sister, Alexandra Day Unit, Princess Alexandra Hospital, Harlow, Essex

- Catherine Beadle
  Ward Manager, Harold Ward, Princess Alexandra Hospital, Harlow, Essex

- Kathryn Farrow
  Clinical Facilitator, Harold Ward, Princess Alexandra Hospital, Harlow, Essex

- Devi Heath
  Nursing Sister, Bluebell Ward, Lister Hospital, Stevenage, Hertfordshire

- A&E Children’s Centre in the QE2 Hospital, Hertfordshire.
3 Abstract

It seems like we are constantly reading news reports about a National Health Service which is understaffed and overstretched. I wanted to address this with my project, by looking at the problem of rostering. A roster is a list of staff members (e.g. nurses, health care assistants, ward managers) with the times they are scheduled to work during a specific period (e.g. 1 week). Rostering is the task of creating a roster.

Hospital wards require continuous staffing, to ensure the patients always have care. Any ward has a finite number of staff members at their disposal. Hospitals also have access to “agency staff” who do not work full time, or for any particular hospital, but can be called in if required. The cost of agency staff is very high and frequently such staff will not be familiar with the patients or the wards on which they must work, therefore it is desirable to control their use.

As the world’s third biggest employer\(^1\) rostering represents a significant drain on staff time and resources within the NHS. A **ward without a good roster will be unable to provide good care to its patients**. Sub optimal rostering leads to overstretched, understaffed wards or wards staffed with expensive agency nurses. Therefore efficient rostering saves time, money and **potentially lives**.

Despite the existence of rostering software [3,22], nurse duty rostering in the NHS is still performed by hand. In this project I propose to create a piece of rostering software specifically tailored to the needs of the NHS. I also aim to explore some new areas in genetic algorithms [42] and experiment with a different approach to rostering.

The final results of this project are available at my website, [www.nurserostering.com](http://www.nurserostering.com)

---

\(^1\)Source: BBC News
4 Introduction

4.1 Basic rostering concepts

To achieve the continuous staffing needed on hospital wards, a 24 hour period is broken down into a number of shifts. These shifts are assigned to staff members in a manner which ensures the whole 24 hour period is correctly staffed. It is common to see a 3 shifts system on many wards:

- “Early Shift”, which spans over a morning
- “Late shift” which spans afternoon and evening
- “Night shift” which spans a night

There are numerous variations and many other commonly used possibilities. Some wards use half shifts (such as half an early), or overlapping shifts. One commonly occurring overlapping shift is a “long day shift” which overlaps both Early shift and Late shift and spans a 12½ hour period. It is common to use a combination of overlapping shifts and half shifts to create a number of shifts with varying durations. Figure 1 shows an example of a set of shifts in a “Gant” style chart. A staff member can only work one shift in a day, up to some maximum number of hours a week.

![Time Division](image)

Figure 1 Example division of a 24 hour period into shifts

A duty roster is a set of assignments of shifts to staff members across a some time period. It is usually expressed in a tabular form.
An extract from duty roster is pictured in Figure 2 as an example:

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>Grade (Skill level)</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle</td>
<td>G</td>
<td>Early</td>
<td>Early</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farrow</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td>Early</td>
</tr>
<tr>
<td>Bloomfield</td>
<td>F</td>
<td>Night</td>
<td>Night</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hall</td>
<td>F</td>
<td>Late</td>
<td></td>
<td>Late</td>
<td></td>
</tr>
<tr>
<td>Sunner</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>Early</td>
</tr>
<tr>
<td>Geddis</td>
<td>E</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Late</td>
</tr>
<tr>
<td>Lewis</td>
<td>E</td>
<td>Early</td>
<td></td>
<td>Late</td>
<td></td>
</tr>
<tr>
<td>Miller</td>
<td>E</td>
<td></td>
<td>Night</td>
<td>Night</td>
<td></td>
</tr>
<tr>
<td>Wiseman</td>
<td>E</td>
<td></td>
<td>Early</td>
<td>Early</td>
<td></td>
</tr>
<tr>
<td>Castillo</td>
<td>D</td>
<td>Early</td>
<td>Late</td>
<td></td>
<td>Late</td>
</tr>
<tr>
<td>Cells</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>Late</td>
</tr>
</tbody>
</table>

Figure 2 A sample duty roster

These duty rosters are often termed “off duties”. A more accurate name would be “on duties” as they indicate what days and times a staff member is on the ward.

4.2 What is the rostering problem?
Creating a duty roster for a ward boils down to finding a combination of shifts for every staff member which fulfils a number of specific criteria, based on staffing levels and other structural requirements.

This would imply that once a roster has been drawn up for say, one week, it could be applied to all weeks. This is not the case.

On most wards, nurses may request days off, or request shifts. These requests will vary from week to week (around a nurses personal life), so other staff members must have their assigned shifts altered to take account for this absence. Furthermore, nurses may not work the same number of hours each week due for many reasons, including small differences in the duration of shifts. A nurse may work 30 hours one week, 37.5 hours the next week etc. This type of “5 shifts one week, 4 shifts the next week” pattern is typical of some part time staff. The roster must be adjusted for these varying work patterns.

Annual leave or paid holiday represents a further problem, at any one time one or more staff members may take annual leave. This means that other staff members must have their shifts altered to make up for this absence.

Other obligations such as training courses (so called study days) or “administrative” work, also represent a problem. These will not occur at the same point each week, and will not occur in every week. This means that staff members will have their shifts altered to cover for these absences.

Numerous other complicating factors could be listed, however the upshot of all of these things is that each week in a nurse duty roster poses a new and unique challenge.
4.3 Why is rostering so important?
Rostering is a crucial activity on any ward. Without a good roster, patients may find a ward understaffed, overstretched or the staff unable to provide the quality of care they need. Rostering is also safety critical, especially on those wards that deal with acute illnesses. An understaffed ward is potentially dangerous for the patients.

A ward will be unable to provide a good standard of care without a good duty roster.

From the nurse’s point of view, good rosters make the job more satisfying and rewarding. Understaffed wards are nightmarish to work on and rosters which honour staff requests and avoid unpopular combinations are generally favourable and less tiring to work.

Good rosters can save hospitals money by reducing the number of agency staff needed and efficiently using the non-agency staff members.

4.4 Why does this process need automation?
There is no standard methodology within the NHS for rostering. Therefore each ward often ends up ‘inventing’ their own methodology. Thus as an employer the NHS suffers from the inconsistent use of data. Duty data cannot be stored or analysed to work out more efficient methods of using staff because this data does not exist in a single format.

Current commercial nurse rostering packages [3], are tailored to the international market and do not address the unique problem of rostering in the NHS fully, particularly the assignment of agency staff.

Consultation with nursing staff indicated, that drawing up a week’s duty roster for a normal sized ward (30 staff members), accounting for all the various requirements can take up to 3 hours. Rosters are commonly tabulated by hand using pencil and customised duty tables. These rosters must then be copied out if different views / parts of the roster are needed and photocopied etc. This leads to an endless “paper chain” of hand-written rosters and accompanying documents, and a lot of repetition of tasks which could be easily automated.

Figure 3 and Figure 4 show some real duty rosters for Harold Ward, Princess Alexandra Hospital, Essex.
Figure 3 Hand written duty rosters

Figure 4 The "paper chain"
Rostering by hand creates a number of other problems aside from wasted hours:

- Sub-optimal solutions due to the difficulty of considering a large number of possibilities using a paper approach.
- There is often no training given to nurses on rostering.
- Rostering is a task which is assigned to different staff members each month who will have differing levels of skill.
- Rostering by hand is susceptible to personal bias. This can lead to unnecessary tension in the workplace.
- Rostering is seen as a difficult chore. Very few staff members are motivated to spend the time necessary to find an optimal solution.

4.5 What makes the rostering problem difficult?

There are two contributing factors which make a rostering difficult:

1. The number of constraints a roster must conform to.
2. The number of possible rosters

There are many constraints a roster must not violate (hard constraints). For example, having the correct number of staff with the correct skills on each shift. It is also never possible to place a shift occurring during the day on the day after a night shift. This is because the nurse will be sleeping! There are some constraints which are preferable but not necessary for the roster to be usable (soft constraints). For example, day off requests should be honoured, but do not need to be.

When creating a roster for any week, one must find some optimal balance between these various constraints to produce a good roster. Sometimes constraints may even be contradictory, in which case, one must attempt to find an optimal balance (e.g. you should not schedule shifts at the weekend, yet the weekend must be fully staffed).

Any ward only has a finite number of staff at it’s disposal. Usually agency staff can be called in to plug gaps in staffing, but their use is expensive and undesirable as agency staff may not be familiar with the ward. Therefore the person creating a duty roster must try to find the best possible combination with non-agency (permanent) staff members, and only use agency staff as a last resort. If there were an infinite number of cheap agency staff then the problem would become trivial, as any staff member could be assigned to any set of shifts, and the gaps plugged with agency staff.

To find an optimal roster, one must search though the possible rosters to find the best one. A problem arises in the number of possible rosters.

A single week roster will clearly contain 7 days. On an average size ward there will be around 30 staff members. This means that there are $7 \times 30 = 210$ cells in the roster which must be filled. To simplify the problem, let us say that each one of these cells can take at most 4 values, these being: Early shift, Late shift, Night shift and day off. This yields:

$$4^{210} = 2.7 \times 10^{126}$$

different possibilities.

There are too many possibilities to be exhaustively searched in a reasonable period, on any existing computer. Unfortunately rostering, like timetabling is an NP complete
problem, so there are no known algorithms which can find an optimal roster in polynomial time with respect to the size of the roster.

In general, if there are $K$ shifts, $N$ staff members and $D$ days in the roster, then there are:

$$K^{(N \times D)}$$ possible combinations.

Therefore any small increase in the number of staff exponentially increases the problem's complexity. The problem is combinatorially explosive.

![Graph showing the complexity of duty rostering problem.

Figure 5 Complexity of duty rostering problem

Figure 5 shows the number of combinations for various sized rosters on a logarithmic scale.
4.6 Project Objectives

The key objectives of this project are as follows:

- Create a system which can:
  - Totally replace the currently used pen and paper method of rostering in the NHS
  - Yield better results
  - Reduce the time spent to generate a roster

- Create a system which is sufficiently general that it can be applied to:
  - Any ward
  - Any shift structure (typically these vary between wards)
  - Any staff (part-time, full-time, agency)
  - Any requirements / constraints
    - Potentially any similar rostering problem, even beyond the scope of nursing.

- Create a system which can operate in situations where the ward is seriously under-staffed, and can use and assign agency staff to solve staffing difficulties (section 6.7).
  - The problem of agency staff had not been tackled in many of the papers on this subject [2,3,27], so I thought it would be a new and interesting area to explore and of particular relevance to the NHS.

- Create a user interface so that nurses can use the software easily with a minimum amount of training (section 8).

- Attempt to explore new areas of genetic algorithms for rostering, than were explored by previous papers on the subject. Specifically this project will examine:
  - Multi-threaded algorithm with co-operating threads (section 7.9)
  - A Non-classical, constrained, genetic representation (section 6.3)
  - Experiment with twin removal (section 7.4)

- Create a system which can be used in combination with manually creating the roster.
  - In the ill-defined and highly complex world of human resources, this represents a significant advantage, as some staff work fixed rosters while for others the roster may need to be fixed on certain days. Some constraints may be too vague to express to the computer, or too complex to be worth expressing. Therefore the human can deal with these, and allow the machine to finish the roster.
5 Background

5.1 Visualising the rostering problem

We can visualise the rostering problem as a “search space”, which contains all possible rosters (Figure 6). One or more of the rosters in this space will be optimal, and in the above diagram we can see this represented as roster F which has no violated constraints. An algorithm will have to travel through roster space to find roster F, in the above example.

The problem can therefore be seen as a optimisation problem, where we are attempting to find an optimal roster.

This search space has many local optima, however. Local optima are places where there is some sub-optimal roster surrounded by other sub-optimal rosters, such that they obscure a route to the optimal roster for a search algorithm. We could visualise roster B as one such local optimum, as all the rosters around it are worse, or equally bad.

5.2 Ranking rosters

To be able to judge how good a roster is, we need to define some measure of it’s “fitness”. Usually this is in the form of a “fitness function”.

One way of creating a fitness function is to count the number of constraint violations in the roster. This would create some numerical summary of the “fitness”, and as this figure decreased, this would indicate a fitter roster. We can see this as being similar to
a “Utility” function in utility theory [31,32]. This fitness summary, or utility is some measurement of the intrinsic worth of the roster to the user.

In fact, we require a function which can take many different constraints of differing types. Therefore our utility function is more like a “multi-utility” function [32], with different weights for the differing types of constraint. Constraints themselves may be represented programmatically, in terms of which shift combinations are undesirable for example. They can also be represented mathematically (e.g. one might say the mean number of occurrences of a shift needs to equal some number I).

To generate a utility function, one could take some weighted sum of the number of constraint violations, with weightings based upon constraint type. There are many other possibilities for creating such a function, including polynomial based functions.

Another sophisticated alternative would be a function, which given two rosters, using logic, or fuzzy logic to determine whether one was better than the other. Rosters could then be compared and ranked using this function.

5.3 Genetic algorithms

5.3.1 Overview

There are many ways of searching the space of all possible rosters to find a suitable one, and many different algorithms which may be applied, including:

- Tabu search [18]
- Simulated annealing [28]
- Brute force (impractical for all but toy examples)
- Mathematical methods of optimisation [29]
- Variable neighbourhood search [30]

For a detailed treatment of solving the rostering problem using these methods, please refer to [18,19,21,24,26,29,30].

I however, decided to focus mainly upon a class of algorithms known a genetic algorithms. Their use is popular in the field of timetabling and rostering and they have been used successfully in the past [2,3,20]. I also found their flexibility and operation intriguing, and wanted to explore it further in this project.

John Holland is often considered the inventor of Genetic Algorithms [42]. A genetic algorithm is an optimisation algorithm which takes a heuristic approach to optimisation, searching through a search space in a directed manner. A genetic algorithm does not exhaustively consider every possible combination unlike brute force methods, although it is not guaranteed to find an optimal solution.

In a genetic algorithm, we start with an initially random “population” of solutions. An evolutionary process can be applied to this population. To do this we rank the solutions in terms of their “fitness” for use, “mate” the best solutions in some way, and allow the offspring to replace existing members of the population. Possibly some genetic mutations are introduced to simulate the mutation which occurs in biological processes. This process is repeated until some termination conditions are met. Each iteration is known as a “generation”.

14
We assume that by “mating” good solutions, at least some of the offspring will contain the best elements from the parents (Figure 7).

![Figure 7 Selective breeding](image)

Genetic algorithms can be applied to any type of problem which requires searching of a space, and have been successfully applied to problems like travelling salesperson [9], and timetabling [6], where results within 10% of the optimum can be produced. This flexibility makes genetic algorithms particularly suited to rostering, where constraints and roster structure can vary between wards. I also found the complex searching behaviour which results from randomly applying simple operators very intriguing and wanted to explore it with this project.

5.3.2 Essential components

As outlined in [11,42] there are a number of components which make up any genetic algorithm.

5.3.2.1 Genetic representation scheme

A genetic algorithm always requires some form of “genetic” representation of the entity that the genetic algorithm is trying to optimise.

Often this genetic representation is based around biological concepts.

5.3.2.2 Crossover operator

This takes the genetic representation for two (or more) entities (rosters in the case of this project) and produces a child new entity. In other words this operator performs “mating”.

Often this is a case of choosing a “crossover point” and copying genes from the parents into children, according to the location of this “crossover point”. A number of classical crossover methods (OX Order Crossover, single point crossover, CX Cyclic crossover etc.) will not be discussed here in detail, however some relevant papers are listed in the references section of this report [6,7,8].
5.3.2.3 Mutation operator

Commonly a mutation operator takes one of the entities under consideration (i.e. rosters), and places a random sequence into it’s genetic representation, in some way as is done in [9] and [2]. Again, this may or may not be based around biological processes.

In fact the mutation operator does not need to act on the genetic representation, and could act on the entity itself (i.e. modify the roster directly), while this is not the classical method by which mutation is performed, we shall see this approach later in the project.

The idea is to introduce some random variation into the population, which may lead to improvements and prevents the algorithm getting stuck.

![Mutation Diagram](image)

**Figure 8 Mutation**

5.3.2.4 Selection & replacement scheme

To selectively breed solutions, we need some method of selecting those that need to be bred.

Many different selection schemes have been conceived, including tournaments between solutions [9] and ranking the solutions by various criteria.

In the context of rostering, the approach to selection that is often used, is to rank rosters by their “fitness” for use, and allow the top $n$ percent to breed. This top $n$ percent is termed the “elite”. The top $n$ percent may also be obtained through a “tournament” method where rosters are compared in a pair wise manner.

There are many possible replacement strategies, one option is to replace all the rosters except for the “elite”, with the new offspring rosters. This approach has been used in this project.

5.3.3 Advantages of genetic algorithms

- Mutli-point search
- Resistant to becoming stuck on a roster which is sub-optimal
- Combines elements of both random search and directed search
- Scaleable with regards to roster size

5.3.4 Potential drawbacks of genetic algorithms

- Sub optimal
- Can still become “stuck”
- Computationally intensive
5.4 Hill climbing

5.4.1 Overview

Hill climbing algorithms [39,40,41] are also heuristic search algorithms. Sometimes their use can be favourable over genetic algorithms [39].

If one is trying to climb a mountain, and cannot see their surroundings, a reasonable method of reaching the summit may be to always move in an uphill direction. In other words, in this situation one may try to move in a direction in space as to increase one’s altitude. We assume that when there is a downhill slope in all directions we have been reached the summit.

A hill climbing algorithm starts at some point in the space we wish to search. In the context of this algorithm we can regard this space as the set of all rosters, therefore the hill climb algorithm will be given a starting roster.

![Figure 9 Hill climbing](image)

A hill climber is a type of “best first search”, it attempts to move around the search space in a manner which will always increase the “fitness” of the roster it is at. When it can no longer move to a roster which has a greater fitness than the current roster, then it is assumed that a optimum is reached. This behaviour is illustrated in Figure 9.

To move around the space of all rosters the hill climber must have some operation, or set of operations, which allows it to move from one roster to another. In other words, there must be some rosters which are defined as being reachable from any given roster.

5.4.2 Potential drawbacks

The biggest drawback of this approach is the risk of becoming stuck in local optima. This occurs when the hill climbing algorithm reaches a roster, from which it can reach
no better rosters in terms of fitness. The hill climber must stop at this point, yet thus point may not be the global optimum.

![Figure 10 Stuck in a local minima](image)

In Figure 10, the lines represent the rosters which the hill climber can reach, by applying it’s operations to the roster. Now, consider the hill climbing algorithm is given roster B. The algorithm will run, but realise that all of the rosters it can reach from roster B are worse than roster B, as they have more constraints violated. It will therefore terminate. However, roster F has no constraint violations, and is therefore clearly better than B, but F could never be reached from B by the hill climbing algorithm.

On the other hand, if the hill climbing algorithm had started from E, F would have been reached almost instantly.

In reality the space of all possible rosters is very much like this. This makes hill climbers largely unsuitable for solving the rostering problem alone.
5.5 **Investigating the ward computing environment**

Before software development could begin, an initial visit to a ward was arranged to gather information on the computing facilities available. This was required so the project could be designed with the specific ward environment in mind.

A visit was arranged with the Ward Managers at Harold Ward, Princess Alexandra Hospital, Essex. Upon questioning it was found that the main computing resources are located in the nurse’s office (Figure 11) on any ward.

![Figure 11 Harold ward nurse's office](image1)

![Figure 12 Most modern machine in nurse's office](image2)

Two computers were present in the nurse’s office. Both were Pentium PCs, with Windows 2000 installed. The newest machine was a modern (at time of writing) Pentium-4 2.0Ghz machine, with 256Mb of RAM (Figure 12).

Since the development machines used for this project were all of a similar specification to this machine, it was decided that any system which ran feasibly on a development machine would run on the computers found in the nurses office. Indeed these machines were comparable (at time of writing) to the machines found in Imperial College’s computing labs.

The office also had a high-speed internet connection.

Conveniently for the purposes of the project, the paper nursing duty rosters were also kept in the nurse’s office. This meant that moving the rosters to an electronic system would not mean they would have to be physically moved.

The everyday office tasks carried out on these computers, only use a fraction of the available CPU power. Therefore there is ample CPU time available for solving the rostering problem.
5.6 Constraints for the nurse duty rostering problem

5.6.1 Overview

To construct a list of constraints upon a nursing roster required comprehensive information gathering exercise involving liaison with the following nurses:
Pauline Howson (Alexandra Day Unit, Princess Alexandra Hospital, Essex)
Catherine Beadle (Harold Ward, Princess Alexandra Hospital, Essex)
Kathryn Farrow (Harold Ward, Princess Alexandra Hospital, Essex)
Devi Heath (Bluebell Ward, Lister Hospital, Stevenage)

I also consulted a report regarding staffing levels and health and safety commissioned by the Royal College of Nursing [1].

It is important to note that not all constraints have the same importance, not even within a group of constraints. Naturally, hard constraints are more important than soft constraints. There also will be some order of importance within the set of hard constraints, and similarly an order or importance within the set of soft constraints.

Another important point, is that some constraints present on a roster will be contradictory, in this case the algorithm will merely have to look for the best balance between the conflicting constraints.

5.6.2 Hard constraints

5.6.2.1 Overview

There are a number of hard constraints which apply to almost any roster and a number which will apply only to the nurse duty rostering problem.

In the nurse rostering problem, many constraints revolve around the skills which staff members posses. This is because for a ward to work correctly, staff members with the correct skills must be present, in the correct numbers. A good example is as follows: “To run a paediatric ward I need at least 4 staff members who are fully qualified with a NVQ or better in nursing. Furthermore I need at least 2 staff members who are qualified to ‘Registered Sick Children’s Nurse’ (RSCN) level. On a large ward I may need one final member of staff who is a play specialist during the day.”

It is these constraints which make this problem especially challenging and interesting.

There are documents produced by the NHS which outline the number of staff members required on each shift for a particular sized ward [1].

The following sections outline my research findings for hard constraints in the nurse duty rostering problem.

5.6.2.2 Staffing levels and skills

Broadly, constraints regarding staffing levels and skills were found to be:
Early shift may require 4 trained staff members and a Night shift may require 2 trained staff members. Therefore the number of staff members required on a shift is a property of the shift itself. A workable nursing roster will always have the correct number of trained staff on any shift.

In the event that a shift needs to be staffed by staff members with certain qualifications (for example qualifications in IV injections, Paediatric nursing), the shifts must have nurses qualified to the correct level, in the correct numbers (you may need 2 qualified RSCN nurses on a day shift in a paediatric ward for example).

The staffing levels of particular shifts on particular days, can vary. For example, an early shift on Monday may need 5 staff, but on Wednesday it may only need 3 staff. This is particularly the case in non-acute day surgery wards, where patients are often admitted on Monday, operated on Tuesday and discharged on Wednesday, for example.

It may also be necessary to specify the maximum number of trained and the number of untrained staff on any particular shift.

Some staff are not counted when considering the number of staff members on the ward.

5.6.2.3 Structural considerations
There are a number of hard constraints which relate to the overall structure of a nurse duty roster, or in fact any roster. These are outlined below:

Some shifts cannot be followed or preceded by other shifts
E.g.
*Do not schedule day duty for any member of staff on the day after night duty.*

Some staff members do not work certain shifts, they should never be scheduled for these shifts.

No member of staff can work more hours in one week than stated in their contract.

All members of staff should work the number of hours stated in their contract,

No member of staff can work more than one shift in a day,

No members of staff should work on days which are designated for taking short courses (so called “study days”).

No staff member must work on any day which has a fixed assignment.

For example:
A nurse performs “administrative” duties on a Wednesday. That staff member
should never be scheduled for anything apart for administrative work on Wednesday, and any roster in which they do is not usable.

- Annual leave
  Nurses may take annual leave. On these days it is not possible for them to work any shifts.

### 5.6.3 Soft constraints

#### 5.6.3.1 Overview

Broadly, soft constraints can be split into two main categories. There are those constraints which relate to staff requests i.e. shifts that staff members have explicitly requested and there are those constraints which relate to the ideal structure of a roster.

#### 5.6.3.2 Staff Requests

Staff will specify requests which ideally should be met, broadly these can be as follows

- Requests for a shift (or day off) on a particular day
- Requests for anything except a particular shift on a particular day e.g. No night duty on Tuesday.
- Requests for certain shift types (i.e. day shifts) to be scheduled for them
- Requests for no shifts of certain types (i.e. no day shifts).

It is worth noting that the management of a hospital may cap the number of requests which any staff member is allowed to make in a certain period. This means that staff members cannot pre determine their part of a duty roster, by requesting shifts on every day of the week.

It is also the case that staff requests will have a varying degree of importance from almost essential, to indifference.

#### 5.6.3.3 Correct Groupings / Clustering of assignments

Certain assignments require grouping together. For example Days off should be grouped together on the roster to avoid so called “split days off”, where the days off are not consecutive. The idea here is that for a normal person, who works a regular desk job in any other industry, weekends are two consecutive days off (Saturday and Sunday). Nurses may not have the luxury of having Saturday and Sunday off, but they should have 2 days off consecutively.

Also Early shifts should occur no more than 3 times in a row (4 Early shifts in a row is frowned upon).

These constraints can be expressed generally in the form:

No more than $Z$ and no less than $X$ occurrences of shift $Y$ should occur consecutively.

In some cases, this constraint may be a hard constraint, for example it should not be possible to schedule 4 “long day shifts” in a row, as these are 12.5 hours long, and working for a 50 hr continuous stretch would breach EU regulations. However in the majority of cases, this constraint is a soft constraint.
To summarise, on any nursing roster, some shifts (or days off) must be clustered, whilst others must avoid clustering. By clustering, I mean the scheduling of the same shift on several consecutive days for any one staff member. Four early shifts in a row for example is undesirable, conversely four night duties in a row is preferred to having so called “split nights”.

5.6.3.4 Consecutive shift scheduling

Certain shifts should ideally not be scheduled on consecutive days.

Early shifts and Late shifts are a prime example of this, as it is considered bad practice to schedule an early shift after a late shift. This can be generalised to:

\[ \text{Shift } X \text{ should not occur on the day after shift } Y \]

5.6.3.5 Consistent numbers of people scheduled on shifts

Ideally the staffing level of each shift, should be as high as possible, and vary as little as possible between days. Thus, the mean number of people working on a shift should be maximised, and the variance should be minimised.

5.6.3.6 Weekend considerations

Some shift combinations are undesirable for any staff member at weekends. For example, it is not desirable to work Late shift on both Saturday and Sunday.

It is generally undesirable to have shifts scheduled at weekends for staff members. Since it is unavoidable that some staff members will have to work weekends, we can then say that “it is undesirable for any staff member to work both Saturday and Sunday”.

Hence there are two constraints here:

a) Avoid scheduling staff to work on both Saturday and Sunday
b) If staff members must work at weekends, avoid certain combinations.
6 Designing the algorithm

6.1 Solving the rostering problem in this project

In this section I will give a brief outline of the method by which this project attempts to solve the rostering problem, and specifically, which algorithms are used and the order in which they are applied.

The approach of attempting to solve the rostering problem using genetic algorithms, runs into some interesting difficulties when considering agency staff.

Unlike non-agency staff members who have fixed working hours, an agency staff member can work anywhere from 0 to 37.5 hours a week. This variation creates difficulties for the genetic algorithm, causing an explosion in the size of the search space, since there are many possible combinations any agency staff member can work. A further point is that agency staff are usually unnecessary to solve the rostering problem, provided a ward is adequately staffed. This would imply, in the vast majority of cases there is simply no point putting them into to genetic algorithm as it will only serve to complicate the problem.

Also it is difficult to assign a “fitness” to the use of agency staff. Use of agency staff generally make a roster less fit for use (due to their higher wages), however, their use is only slightly better than a hard violation. Therefore this creates a problem with assigning fitness’s and weights in a fitness function.

The approach this project uses is to initially ignore agency staff, and simply run a genetic algorithm to attempt to solve the rostering problem using only non-agency staff. The genetic algorithm used in this project itself, takes a different approach to other papers read on the subject, and uses multiple co-operating threads (section 7.9). If the genetic algorithm fails, it then passes the partial solution to a set of hill climbing algorithms.

Since, it is usually the case that very few agency staff members are needed, and typically they work very few shifts, the assignment of agency staff to shifts can be done using a simple hill climbing method (section 6.7). This is an attempt to maximise the performance of the common case (no agency staff needed).

In the rare case where a ward is chronically understaffed the hill climbing algorithm may fail to find a good solution. In this case, the genetic algorithm can reapplied. The hill climber’s result is used to determine the number of hours an agency staff member must work in a week and the agency staff members are changed to non-agency staff members. This simplifies the problem by reducing the number of possible assignments. The genetic algorithm is then re-run.

To measure the fitness of rosters and rank them I decided to use a mutli-attribute utility function [31,32], based upon a weighted average of the number of constraint violations in the roster. I will term this utility value “score”. Lower values for “score” indicate rosters better rosters. Thus I am treating rostering as a minimisation problem,
where “score” must be minimised. This approach to computing the fitness of rosters is similar to the approach used in previous papers [2,3].

To summarise:

- A genetic algorithm considers non-agency staff members
- These are easier to deal with as they have a fixed number of working hours in a week.
- If no solution can be found using only non-agency staff, we pass the partial solution to a hill climber
- The hill climber assigns agency staff until the roster is usable.
- The genetic algorithm does not deal with agency staff due to their varying week length. This creates far more possibilities to be searched.
- Finally, the roster is passed back into the genetic algorithm, with all agency staff being considered as non-agency after the hill climber’s has assigned them shifts.

Figure 13 illustrates this method.
The genetic algorithm is set up to assign agency staff. They are treated exactly the same as non-agency staff. The number of hours worked by these staff is calculated from the shifts assigned by the hill climbers. This helps narrow the search space for the genetic algorithm.

By default the rostering problem is assumed solvable without using agency staff. This considerably simplifies the algorithm, and reduces the search space the algorithm must search.

If the rostering problem was not solvable without using agency staff, i.e. even after a run of the genetic algorithm, there are still serious problems, then agency staff must be assigned.

Hill climbers assign agency staff, one shift at a time, one staff member at a time to try and minimise the number of agency staff used. Usually, they will be successful. Particularly if the ward is correctly staffed.

In the case of a severely understaffed ward, the hill climbers may not find a solution. The genetic algorithm can use the partial solution created by the hill climbers, and attempt to improve it.

Figure 13 Solving the rostering problem
6.2 Motivations behind representation scheme for a roster

The main difficulty with using genetic algorithms on rostering problems, is that the genetic algorithm requires guidance in the massive search space of rosters, as noted by Grobner et al in their paper on nurse rostering using a simple genetic algorithm [2]. The algorithm requires some form of domain specific knowledge so it can guide itself into a feasible area of the search space by eliminating erroneous rosters. A straight genetic search strategy without any domain specific knowledge, will have problems finding a feasible solution.

Previously domain specific knowledge had been encapsulated in:
- “Fix” operators which resolve problems in rosters to steer the genetic algorithm into feasible areas of the search space. [2]
- Complex decoders which take a simple genetic representation and produce a legal roster. [3]

In my view, the problem with these approaches is the simplicity of the representation scheme and genetic operators, which allows erroneous rosters into the population (such as rosters with 2 shifts on a single day for a single person). I decided to take a different approach where domain specific knowledge is incorporated into the genetic representation and genetic operators.

Therefore this project aims to create a genetic representation which enforces some hard constraints. Appropriate crossover and mutation operators are also needed to ensure that these constraints are always maintained. It is intended that the genetic representation will guide the algorithm into a more feasible area of the search space by excluding erroneous rosters.

The below Venn-diagram illustrates these ideas:
6.3 Representing a roster

6.3.1 Overview

In this project a roster may be represented in 2 ways (Figure 14):

1. As a table
2. In a genetic representation

![Diagram showing roster represented as a table and as "Genes".]  

Figure 14 One roster, two representations

This data can be viewed by the user in many different ways, as shown in section 8.1.

The tabular representation of a roster is useful for display purposes and evaluation of the fitness of the roster (section 7.4), while the genetic representation is required for crossover operations (section 6.4).

A roster is defined over a fixed period (1 week) for a fixed set of staff.

6.3.2 Tabular representation

The tabular format of a roster is illustrated in Figure 15.

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle</td>
<td>Early</td>
<td>Early</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farrow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloomfield</td>
<td>Night</td>
<td>Night</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hall</td>
<td>Late</td>
<td></td>
<td>Late</td>
<td></td>
</tr>
<tr>
<td>Sunner</td>
<td></td>
<td></td>
<td>Early</td>
<td></td>
</tr>
<tr>
<td>Geddis</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Late</td>
</tr>
</tbody>
</table>

Figure 15 Tabular representation
As can be seen, each column in the table represents a day, and each row represents a staff member. If a cell contains a shift, then this represents an assignment of a staff member to a shift on a particular day, for example in the above table, “Farrow” must work an “Early shift” on “04/03/2004”.

This representation matches the classical paper-based representation of rosters used on hospital wards and can be implemented programmatically in the form of a 2-dimensional array.

### 6.3.3 Genetic representation

I define a “gene” to be a pair of a day and a shift.

- e.g. (Monday, Early) is a gene
- (Tuesday, Late) is a gene

I define a “chromosome” to be a set of genes.

Each non-agency staff member has an associated chromosome. Each chromosome represents the shifts assigned to that staff member (Figure 17).

Figure 16 illustrates two chromosomes. In this figure, Chromosome 2 contains the gene (Monday, Night) for example, yet Chromosome 1 does not. This indicates Tina works Night shift on Monday, yet Shelly does not. Chromosome 2 has no genes of the form (Tuesday, X) where X is a shift, this represents that Tina has a day off on Tuesday. In the example, Shelly works Early shift on Monday.

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Early</td>
</tr>
<tr>
<td>Mon</td>
<td>Late</td>
</tr>
<tr>
<td>Mon</td>
<td>Night</td>
</tr>
<tr>
<td>Tue</td>
<td>Early</td>
</tr>
<tr>
<td>Tue</td>
<td>Late</td>
</tr>
<tr>
<td>Tue</td>
<td>Night</td>
</tr>
</tbody>
</table>

Chromosome 1 associated with Staff Member “Shelly”

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Early</td>
</tr>
<tr>
<td>Mon</td>
<td>Late</td>
</tr>
<tr>
<td>Mon</td>
<td>Night</td>
</tr>
<tr>
<td>Tue</td>
<td>Early</td>
</tr>
<tr>
<td>Tue</td>
<td>Late</td>
</tr>
<tr>
<td>Tue</td>
<td>Night</td>
</tr>
</tbody>
</table>

Chromosome 2 associated with Staff Member “Tina”

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Night</td>
</tr>
<tr>
<td>Tue</td>
<td>Early</td>
</tr>
<tr>
<td>Tue</td>
<td>Late</td>
</tr>
<tr>
<td>Tue</td>
<td>Night</td>
</tr>
</tbody>
</table>

Figure 16 Example chromosomes

A staff member can only work a certain number of hours per week, so the chromosome can only contain a finite number of genes. The number of hours worked in a week is a property of the staff member (Figure 17). The total duration of all shifts in all genes present must be no more than this value. See section 6.4.3 for further details.

To prevent a staff member working too few hours in one week, a chromosome must contain sufficient genes to ensure the total duration of all shifts in all genes is greater than a lower bound, specific to each staff member. See section 6.4.3 for further details.
Some staff members cannot work certain days of the week, therefore another property of a staff member is the days they may work (the “valid days” block on Figure 17). All genes in a chromosome must have a day in the “valid days” set of the associated staff member. For example, the staff member associated with the chromosome in Figure 18, can work only on Monday and Wednesday.

A final property of a staff member is the shifts that they may work (the valid shifts block on Figure 17). For example in a ward with staff members Catherine and Linda, Catherine may only work day shifts, but Linda may only work night shifts as stated in their contracts. No chromosome may contain a gene with a shift which is not in the “valid shifts” for the associated staff member. For example, the staff member associated with the chromosome in Figure 18, only works “Night” shift.

To prevent staff working more than one shift per day, we need the constraint that genes present in a chromosome cannot share days, this is illustrated in Figure 19.

---

**Figure 17** Conceptual diagram of genetic representation

**Figure 18** Example chromosome

**Figure 19** Allowable chromosomes
To summarise, a chromosome will be **invalid or illegal** if:

- Genes present within a chromosome share days (Figure 19)
- Genes have shifts not in the “valid shifts” for the associated staff member (Figure 17)
- Genes have days not in the “valid days” for the associated staff member (Figure 17)
- There are not enough genes in a chromosome to represent a full week for the associated staff member (section 6.4.3).
- There are too many genes in a chromosome to represent a full week for the associated staff member (section 6.4.3).

Figure 20 illustrates these concepts.

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Early</td>
</tr>
<tr>
<td>Mon</td>
<td>Late</td>
</tr>
<tr>
<td>Mon</td>
<td>Night</td>
</tr>
<tr>
<td>Tue</td>
<td>Early</td>
</tr>
<tr>
<td>Tue</td>
<td>Late</td>
</tr>
<tr>
<td>Tue</td>
<td>Night</td>
</tr>
</tbody>
</table>

Invalid (too few genes)

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Early</td>
</tr>
<tr>
<td>Mon</td>
<td>Late</td>
</tr>
<tr>
<td>Mon</td>
<td>Night</td>
</tr>
<tr>
<td>Tue</td>
<td>Early</td>
</tr>
<tr>
<td>Tue</td>
<td>Late</td>
</tr>
<tr>
<td>Tue</td>
<td>Night</td>
</tr>
</tbody>
</table>

Invalid (too many genes)

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Early</td>
</tr>
<tr>
<td>Mon</td>
<td>Late</td>
</tr>
<tr>
<td>Mon</td>
<td>Night</td>
</tr>
<tr>
<td>Tue</td>
<td>Early</td>
</tr>
<tr>
<td>Tue</td>
<td>Late</td>
</tr>
<tr>
<td>Tue</td>
<td>Night</td>
</tr>
</tbody>
</table>

Valid

Invalid or illegal chromosomes are **not permissible**, and no rosters should ever be created with illegal chromosomes. In this project, exceptions are thrown if illegal chromosomes are created. These exceptions can be caught and the chromosomes corrected. As will be illustrated in the following sections (6.4 and 6.5), care is taken to prevent such exceptions occurring.

This genetic representation now enforces 5 hard constraints from section 5.6:

1. Any staff member cannot work two shifts in one day (Figure 19)
2. Any staff member cannot work more hours than they are contractually obliged to.
3. Any staff member cannot work significantly fewer hours than they are obliged to.
4. Any staff member may not be assigned to shifts which they do not work.
5. Any staff member may not be assigned to shifts on days which they do not work.

For every genetic representation of a roster there is an equivalent tabular representation, but not all tabular representations can be represented genetically (given the above constraints). Those rosters which can be represented in a tabular form, but not genetically violate some hard constraints, and therefore need not be considered.
Translating a genetic representation to a tabular representation is a process of iterating over the chromosomes. Each gene corresponds to a different cell in the table. Likewise to translate a tabular representation into a genetic representation, a chromosome is created for each staff member and a gene is created for each populated cell in the table. An example of this is given in Figure 21.

In paper [2] it is suggested that constraining the genetic algorithm too much leads to it becoming stuck in local minima. The following methods were conceived to attempt to prevent this occurring in this project:

- Use a large population
- Using multiple competing threads (section 7.9)
- Removing twins from the population (section 7.6)

### 6.4 Cross Over operation

#### 6.4.1 Overview

Since I was defining my own genetic representation, I needed to define my own crossover operator. Many of the crossover operators used in previous papers, such as Ordered Crossover (OX) [6,7] and single point crossover are designed for unconstrained genetic representations. In my genetic representation, there are specific constraints which must be enforced (as illustrated in Figure 19) so these operators did not seem appropriate. I created a crossover operator which passes common genes from 2 parent rosters into a new child roster. This seemed like a logical choice as classical crossover operators also copy genes from parents into children.

#### 6.4.2 The crossover process

Conceptually, the crossover process is illustrated in Figure 22.

It should be noted that this operator does not have a crossover point as used in many crossover operators.
Parents share common assignment:
Early shift on Monday

Crossover Operation

Common genes passed on to child. Remainder of genetic representation randomly generated, or randomly chosen from a parent.

Figure 22 Conceptual diagram of the crossover process in this project
### Parents

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon Early</td>
<td>X</td>
</tr>
<tr>
<td>Mon Late</td>
<td></td>
</tr>
<tr>
<td>Tue Early</td>
<td>X</td>
</tr>
<tr>
<td>Tue Late</td>
<td></td>
</tr>
<tr>
<td>Wed Early</td>
<td></td>
</tr>
<tr>
<td>Wed Late</td>
<td></td>
</tr>
<tr>
<td>Thurs Early</td>
<td></td>
</tr>
<tr>
<td>Thurs Late</td>
<td></td>
</tr>
<tr>
<td>Fri Early</td>
<td>X</td>
</tr>
<tr>
<td>Fri Late</td>
<td></td>
</tr>
</tbody>
</table>

### All possible children

- **Gene**
- **Present?**

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

```
| Mon Early | X        |
| Mon Late  |          |
| Tue Early | X        |
| Tue Late  |          |
| Wed Early |          |
| Wed Late  |          |
| Thurs Early |        |
| Thurs Late |         |
| Fri Early | X        |
| Fri Late  |          |
```

**Figure 23 Parents with all children**
The crossover operation acts upon the genetic representation of two rosters X and Y to produce a new roster Z, as illustrated in Figure 22.

For each chromosome the operation takes genes present in both X and Y and places them in the corresponding chromosome in Z (Figure 22). It then randomly fills the chromosomes in Z until they are “full” (section 6.4.3).

Imagine nurse Pauline works 22½ hours in one week. Suppose she works “Late” shift and “Early” shift and is available on weekdays. Say there are 2 rosters for nurse Pauline and we wish to apply the crossover operator to these rosters. Figure 23 illustrates two parent chromosomes and all possible children which could be created by the crossover operator in this situation. Note that all the children fulfill the constraints outlined in section 6.3.3. If “Early” shift and “Late” shift are both 7½ hours long, then every chromosome in Figure 23 represents a 22½ hour working week. Also note that the genes common to both parents are present in all the children.

In pseudo-code the crossover operator is as follows:

```
Operation CrossOver( x, y, z : Roster ) {
    For each chromosome “cx” in x {
        Let cz = the chromosome in z corresponding to cx
        Let cy = the chromosome in y corresponding to cx

        Let cg = All the common genes between cx and cy.
        For (each gene g in cg) {
            Add g into cz with probability “prob_crossover”
        }
        while (cz is not full) {
            Either :
                Randomly generate a new gene and add to cz if it is not on the same day as any existing genes. The new gene generated must consist of a day selected from cz.associated_staff_member.valid_days and a shift selected from cz.associated_staff_member.valid_shifts
            Or
                Randomly choose a gene from cy or cx and add to cz if it is not on the same day as any existing genes.

            If (I am having difficulties creating a legal chromosome) Then {
                /* I may have produced a sequence of genes which can never be made into a legal chromosome by adding genes */
                Clear cz;
                /* Start from scratch.*/
            }
        }
    }
}
```

Since X and Y were both valid rosters, the genes present in both X and Y do not violate the constraints imposed upon the genetic representation. Therefore the common genes must represent part of a legal genetic representation.

Currently the probability “prob_crossover” is set to be 1, although can be altered. Evidence from [9] indicates that this probability should be high, otherwise the search becomes undirected.
The crossover operator works by iteratively adding genes into a chromosome until that chromosome becomes legal or “valid” (section 6.3.3). Sometimes the operator can create a sequence of genes which can never be part of a legal chromosome. For example certain combinations of shifts with unusual durations can prevent the target number of hours in a week being reached. In this case, all the work must be discarded, and the crossover operation starts again. This event is detected if more than a threshold number of loop iterations have occurred. This threshold is directly proportional to the number genes which can exist in the chromosome.

Observations during debugging indicate that this is a rare occurrence. This is probably because the chromosomes which can cause this effect are gradually “bred out”, of the population.

6.4.3 Determining the correct number of genes for a chromosome

A chromosome is defined as being “full” when it contains sufficient genes to represent a working week for its associated staff member. Determining whether a chromosome is full has become an increasingly complex task, over the course of the development of the project.

The main difficulty arises from the fact that the presence of certain shifts in the genes of a chromosome can cause a reduction in the number of genes required to fill the chromosome. A good example of this is Night Shifts. Generally if a person is working one or more night shifts in a week, they work one less shift in total. In other words the presence of night shifts reduces the number of allowed working days by 1.

E.g. Staff nurse Linda works 5 days a week when working only day duties. If she works 1 or more night duties in a week then she may only ever work 4 shifts in that week regardless of whether they are night or day duties.

Figure 24 gives another example. Say the staff member associated with the chromosomes illustrated usually works 22½ hours in a week and all shifts are 7½ hours long. If she is working day shifts only, they she must work 3 shifts in a week. However, if she works night shifts, she only needs to work 2 shifts in a week.

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon Early</td>
<td>X</td>
</tr>
<tr>
<td>Mon Late</td>
<td></td>
</tr>
<tr>
<td>Mon Night</td>
<td></td>
</tr>
<tr>
<td>Tue Early</td>
<td></td>
</tr>
<tr>
<td>Tue Late</td>
<td>X</td>
</tr>
<tr>
<td>Tue Night</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon Early</td>
<td></td>
</tr>
<tr>
<td>Mon Late</td>
<td></td>
</tr>
<tr>
<td>Mon Night</td>
<td>X</td>
</tr>
<tr>
<td>Tue Early</td>
<td></td>
</tr>
<tr>
<td>Tue Late</td>
<td></td>
</tr>
<tr>
<td>Tue Night</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gene</th>
<th>Present?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon Early</td>
<td></td>
</tr>
<tr>
<td>Mon Late</td>
<td></td>
</tr>
<tr>
<td>Mon Night</td>
<td></td>
</tr>
<tr>
<td>Tue Early</td>
<td></td>
</tr>
<tr>
<td>Tue Late</td>
<td></td>
</tr>
<tr>
<td>Tue Night</td>
<td></td>
</tr>
</tbody>
</table>

Figure 24 Example full and under filled chromosomes

After trialing a number of different approaches, I opted for an “hours reduction” parameter in chromosomes and shifts. If a night duty gene is present, this “hours reduction” parameter is set to the value of the “hours reduction” property of a night
shift. This is length of one shift. The hours reduction parameter is then added to the total hours in the chromosome.

To generalise further:

- “Hours Reduction” is a property of a **shift and a chromosome**.
- Upon addition of a gene with a shift which has a non-zero “hours reduction” property, this hours reduction value is placed in the chromosome’s “hours reduction” property.
- When checking if the chromosome is full, the “hours reduction” property is added to the total number of hours in the chromosome.
- This is compared with the maximum number of hours that staff member can work in a week.
- If it is smaller, the more genes can be added to the chromosome.

There is one further complicating factor: even without night shifts, the number of hours worked in a week by any staff member may never exactly equal the number of hours stated on their contract. There is scope for a 1 or 2 hour leeway per week. For example if nurse Shelly usually works 37.5 hours as stated in her contract, but an unusual combination of shifts means she only works 36 hours in week 12, then this is perfectly acceptable. In fact, when working with fractions, the inaccuracies in floating point arithmetic make including this “leeway” favourable.

It was simple to introduce a “maximum hours owing” parameter, which is part of a staff member. The number of hours in a week is deemed to be equal to the number of hours stated in a staff member’s contract, if the difference between the two is within the “maximum hours owing” for that staff member.

In pseudo-code:

```pseudo
Boolean chromosomeIsFull(Chromosome C) {
    Hours_reduction = 0
    Total_hours = 0
    For each gene “g” in chromosome C {
        If g.shift.hours_reduction > Hours_reduction then {
            Hours_reduction = g.shift.hours_reduction
        }
        Total_hours = Total_hours + g.shift.duration_of_shift
    }
    If (Total_hours + Hours_reduction <
        C.AssociatedStaffMember.hours_worked_in_a_week -
        C.AssociatedStaffMember.maximum_hours_owing ) then
        This chromosome is not full (return true)
    Else
        This chromosome is full (return false)
    }
}
```

By using this design, any shift can be given “hours reduction” allowing it to exhibit the same behaviour as these night shifts. This allows for far greater flexibility.

In conclusion, we can determine if a chromosome is full by examination of it’s contents. However, in the general case, we can never say with certainty exactly how many genes a chromosome will have in it when it is full.
6.5 Mutation operation

6.5.1 Overview
In this project the mutation operators act upon the tabular representation of a roster, applying random modifications, which in turn cause the genetic representation to be altered. This is because it is more intuitive to apply this project’s mutation operations to a table, as will be seen later.

To create a mutation operator which acts effectively upon rosters, I decided to examine how rostering occurs in practice. I wanted to encapsulate domain specific knowledge about solving the rostering problem in the mutation operators as well as the genetic representation.

To do this I spoke to Pauline Howson (Ward Sister, Alexandra Day Unit, Princess Alexandra Hospital, Essex) and inquired as to the methods she used for resolving problems in a roster, and generally generating a roster. I decided to build my mutation operators around these methods. I hypothesised that applying these techniques to a roster would help solve problems, even when applied randomly. This was in contrast to using a more classical set of mutation operators, as defined in previous papers such as [2,3]. Furthermore these mutation operators are defined so they never break the constraints imposed upon the genetic representation (section 6.3.3).

Therefore the following set of operators was proposed:

**Method 1:** Swap two shifts around for an existing staff member (Figure 25).

**Method 2:** Replace all the shift assignments for an existing staff member, with new, randomly generated assignments.

**Method 3:** Choose a random day, and two staff members. Swap the assignments for the staff members around (Figure 26). The two staff members must be working on shifts which are allowable for both staff members. If this results in an illegal roster, randomly drop or create assignments until it is legal. It is acceptable to swap a day off for one staff member, with a shift assignment for another staff member.

**Method 4:** Replace an existing assignment, with a new, randomly generated assignment (Figure 27). If this results in an illegal roster, randomly drop or create assignments until it is legal.

<table>
<thead>
<tr>
<th></th>
<th>Monday</th>
<th>Tuesday</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Staff Member 1</strong></td>
<td>Early</td>
<td>Late</td>
</tr>
<tr>
<td><strong>Staff Member 2</strong></td>
<td>Early</td>
<td>Early</td>
</tr>
</tbody>
</table>

*Figure 25 Method 1 example*
6.5.2 Testing
Since I was defining my own mutation operators to fit my own genetic representation, I felt that it was important to find which of the mutation operators would produce the greatest improvement. I was not, initially certain which operations were likely to help the search, and which were likely to hinder it.

Furthermore, I was interested to see whether a number of mutation operators would be required, or whether a single mutation operator would be sufficient to solve the problem satisfactorily.

Finally, in previous papers such as [2,3], the authors spent time optimising their mutation operators and the other parameters of their algorithm to obtain the best results, it seemed logical that I should go through the same process.

6.5.3 Experimental method
It was decided to test each of the 4 methods outlined in section 6.5.1, individually. The behaviour of the genetic algorithm with no mutation operations was also observed and used as a benchmark.

It was decided that the best way to test the operators was to compare the final score of a run of the algorithm after a fixed number of generations. To minimise experimental error, each experiment was run 15 times.

It was decided to take the best result created by the genetic algorithm after 300 generations. This was because, after much observation of the algorithm during debugging, it was noted that the algorithm had “settled down” very well by about 300 generations, and little extra change was likely.

Finally a mutation operator which randomly applies each of the 4 methods above was tested. The probabilities were as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1</td>
<td>0.15</td>
</tr>
<tr>
<td>Method 2</td>
<td>0.02</td>
</tr>
<tr>
<td>Method 3</td>
<td>0.50</td>
</tr>
<tr>
<td>Method 4</td>
<td>0.33</td>
</tr>
</tbody>
</table>
Experimental conditions were kept constant for all tests, see appendix 1 for full details.

6.5.4 Results
The following graph shows the experimental results. The stalks represent the 95% confidence interval. The Y-Axis represents “score” which is a measure of the utility of the roster (see section 7.5). Higher values indicate poorer rosters.

![Figure 28 Overall Mutation Results](image)

6.5.5 Conclusions
As can be seen from the graphs, there is relatively little difference between mutation methods 1, 3 and 4, with mutation method 2 being the only method which was significantly worse, when error bounds were taken into account.

It can also be noted that any mutation operator yields a significantly better result than the algorithm with no mutation. This result was not unexpected, as outlined by Holland [42], the mutation operator is an essential component of a genetic algorithm, which enables it to reach all areas of the solution space.

Finally, it can be seen that the combination of operators was slightly better than any of the mutation operators alone, however, when the error bounds are taken into account, it was not significantly better. This implies that the mutation operators act approximately the same when they are allowed to interact, as when they are executed in separation.

Therefore it would appear that since methods 1, 3 and 4 yield approximately the same results, a mutation operator which applies 1, 3 and 4 with approximately the same probabilities would be applicable. It was decided to therefore use this as a mutation operator in this project.
6.6 Formal specification of genetic representation and operations

Having defined the genetic representation and operations informally, I will now outline them in a formal manner. I have formulated a formal specification in Object-Z notation [13,14]. Object-Z is based upon Oxford University’s Z specification language [12,16]. The formal specification is intended as a guide to clarify the exact constraints in the genetic representation, rather than a rigorous specification.

Let DAYS = { Monday, Tuesday, Wednesday …..}

\[
\text{Shift} \quad (\text{duration, name})
\]
\[
\text{name: NAME} \\
\text{duration: } \mathbb{R} \\
\text{Hoursreduction : } \mathbb{R}
\]

\[
\text{Gene} \quad (\text{day, shift})
\]
\[
\text{day : DAYS} \\
\text{shift : Shift}
\]

\[
\text{StaffMember} \quad (\text{name, validdays, hoursworked, validshifts, INIT})
\]
\[
\text{name: NAME} \\
\text{validdays : } \mathbb{P} \text{ DAYS} \\
\text{hoursworked: } \mathbb{R} \\
\text{validshifts: } \mathbb{P} \text{ Shift}
\]

\[
\text{INIT}
\]
\[
\text{validdays } \neq \emptyset \\
\text{validshifts } \neq \emptyset
\]
Chromosome

\[ \{ (\text{associatedStaffMember}, \text{genes}, \text{INIT}, \text{Crossover}) \} \]

\text{associatedStaffMember} : \text{StaffMember}
\text{genes} : \mathcal{P} \text{ Gene}
\epsilon, \text{AllowedOwing} : \mathbb{R}

\forall x, y : \text{genes}.
\quad (x.\text{day} = y.\text{day} \Rightarrow x = y)
\forall x : \text{genes}.
\quad (x.\text{day} \in \text{associatedStaffMember.validdays} \land
x.\text{shift} \in \text{associatedStaffMember.validshifts})
\epsilon = \left( \text{MAX}_{x : \text{genes}} (x.\text{shift.hoursreduction}) \right) + \text{AllowedOwing}

\left( \sum_{x : \text{genes}} x.\text{shift.duration} \right) \in
\left( \text{associatedStaffMember.Maxhours} - \epsilon, \text{associatedStaffMember.MaxHours} \right)

Crossover

\text{A?}, \text{B?} : \text{Chromosome}
\text{C!} : \text{Chromosome}
\text{A?.associatedStaffMember} = \text{B?.associatedStaffMember} = \text{C!.associatedStaffMember}
\text{A?.genes} \cap \text{B?.genes} \subseteq \text{C!.genes}

INIT

\begin{align*}
\text{genes} &\neq \emptyset \\
\text{associatedStaffMember.INIT} &
\end{align*}
Roster

\[
\{(table, chromosomes, INIT, Crossover)\}
\]

\[
table : DAYS \times StaffMember \rightarrow Shift
\]

\[
chromosomes : \bigoplus \text{Chromosome}
\]

\[\forall x : DAYS, y: StaffMember \bullet (\ \text{table}(x,y) \iff (\ \exists c: \text{chromosomes} \bullet (\ c.\text{associatedStaffMember} = y \wedge \exists g: c.\text{genes} \bullet (\ g.\text{day} = x \wedge g.\text{shift} = y ) ))\]

Mate

\[
X?, Y?, Z!: \text{Roster}
\]

\[\forall a: X?.\text{chromosomes} \bullet (\exists b: Y?.\text{chromosomes} \bullet (\ a.\text{associatedStaffMember} = b.\text{associatedStaffMember})\)

\[|X?.\text{chromosomes}| = |Y?.\text{chromosomes}| = |Z!.\text{chromosomes}|\]

\[\forall a: X?.\text{chromosomes} \bullet (\exists b: Y?.\text{chromosomes}, c: Z!.\text{chromosomes} \bullet (\ a.\text{associatedStaffMember} = b.\text{associatedStaffMember} = c.\text{associatedStaffMember} \wedge [ A?: \text{Chromosome}, B?: \text{Chromosome}, C!: \text{Chromosome}] A? = a \wedge B? = b \wedge C! = c) \bullet a.\text{Crossover}\)

\]

Mutate

\[
\Delta(table)
\]

\[
\Delta(chromosomes)
\]

\[\forall a: \text{chromosomes} \bullet (\exists b: \text{chromosomes'} \bullet (\ a.\text{associatedStaffMember} = b.\text{associatedStaffMember})\)

\[|\text{chromosomes}| = |\text{chromosomes'}|\]

INIT

\[
\text{chromosomes} \neq \emptyset
\]

\[
\text{table} \neq \emptyset
\]

\[\forall c: \text{chromosomes} \bullet (\ c.\text{INIT})\]
6.7 Hill climbing problem solvers

6.7.1 Overview
As outlined in section 6.1, it was decided that the genetic algorithm should be concerned with working on shift assignments for non-agency staff. Therefore it was decided to code two hill climbers to deal with the assignment of agency staff. These hill climbers mimic the process a human goes through in altering the roster. Another hill climber was made to “tweak” shift assignments for non-agency staff. These hill climbing approaches were partially inspired by the variable neighbourhood search used by Berke et al [30]. In this paper, rostering is performed by moving shifts around a roster using various operators. They call this “shaking”, and it appears to be an effective method of rostering. I therefore wanted to investigate this method with this project.

The hill climbers operate upon the tabular representation of a roster, rather than it’s genetic representation as this is more intuitive.

6.7.2 Agency staff assignment
The first hill climber I defined was one to assign shifts to agency staff to attempt to “plug gaps” in the roster.

In any single iteration the hill climber tries assigning each shift, in each available agency staff slot, and will pick the best shift assignment out of the possible assignments. Note that the hill climber in any iteration only tries one assignment at a time.

The operation of the hill climber is explained in Figure 29 and pseudo-code:

Let current_roster be the roster which is being “hill climbed”
Let best_roster = An exact clone of current_roster

For each agency staff member B {
    For each legal day of the week D {
        For each shift S {
            Assign B to work S on day D in current_roster
            Evaluate current_roster
            If (current_roster is better than best_roster)
                best_roster = current_roster
            Undo any changes made.
        }
    }
}

return (best_roster)

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tues</th>
<th>Wed</th>
<th>Thurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tina</td>
<td></td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
</tr>
<tr>
<td>Shelly</td>
<td>Late</td>
<td>Late</td>
<td>Late</td>
<td>Early</td>
</tr>
<tr>
<td>Agency Staff 1</td>
<td></td>
<td></td>
<td>Early</td>
<td></td>
</tr>
</tbody>
</table>

Figure 29 Agency staff shift assignment
The above process is repeated until it can yield no further improvement. One iteration has the following cost:

\[ C = bds \]

Where:
- \( C \) = the maximum number of evaluation operations per iteration, i.e. the maximum number of combinations considered per iteration.
- \( b \) = the number of agency staff in a roster
- \( d \) = the number of days in a week (7 in most cases)
- \( s \) = the number of different shifts any agency staff member can work, at maximum.

The time complexity of this hill climber is linear with regards to the size of the ward (assuming that the week length is usually constant), thus making it scaleable and practically viable.

This hill climber assumes that agency staff can be assigned in a globally optimal way through individually optimal assignments of agency staff. This seems like a reasonable assumption because we can see this hill climber as “plugging holes” in the roster at each iteration. Once the hill climber has reached a plateau we can assume it has either plugged all holes, or run out of agency staff.

### 6.7.3 Reassignment hill climbers

It was decided, that there should be a standardised method of taking any roster where a staff member was working an insufficient number of hours, and turning it into a legal roster.

The algorithm designed to do this takes a roster with a single staff member whose week is incomplete. It tries all possible assignments for the staff member to rectify the problem, returning the roster containing the best assignment it could find. The algorithm is outlined in Figure 30 and below in pseudo-code:

```
Let current_roster = the current, incomplete, roster
Let best_roster = the best roster found so far by this algorithm.
Let staff_member = a staff member working an insufficient number of hours in current_roster

While (staff_member is not working enough hours in current_roster) {
    For each day “d” {
        If day “d” is empty in current_roster {
            For each shift “s” {
                Assign staff_member to shift “s” on day “d”
                Evaluate the fitness of current_roster
                If
                    (current_roster is better than best_roster)
                    AND
                    (current_roster is now a complete roster)
                    then
                    {  
                        best_roster = current_roster
                    }
            }
        }
    }
}

Undo any changes made
```
“best_roster” is now a complete roster, which has been completed by the assignment of a shift which yielded the greatest “utility”.

One iteration has the following cost:

\[ C = ds \]

Where:
\[ C \] = the maximum number of evaluation operations per iteration
\[ d \] = the number of days in a week (usually 7)
\[ s \] = the number of shifts which can be worked by the staff member (usually about 3).

Again, this algorithm scales very well, and will usually have the same complexity regardless of the size of the roster (since “d” and “s” are usually constants).

6.7.4 Agency staff swapping

Whilst effective, the agency assignment hill climber (section 6.7.2) was not able to solve problems of consecutive shifts - such as a night duty followed by a day duty - that can sometimes remain after the genetic algorithm has terminated.

To solve this problem would involve moving one of the offending shifts to an agency staff member, then reassigning the staff member to a different shift, possibly on a different day.

The algorithm to achieve this is outlined in Figure 31 and the pseudo-code below:

Let \( current\_roster = \) the current roster
Let \( best\_roster = \) the best roster found so far. Initialise to \( current\_roster \)

For each agency staff member “b” {
    For each non-agency staff member “s” {
        For each day “d” {
            Assign “b” to the same shift as “s” on day “d”
            Remove any assignment for “s” on day “d” in roster “current_roster”
            Complete the roster using the algorithm described in Section 6.7.3
            Evaluate the fitness of \( current\_roster \)
            If (\( current\_roster \) is better than \( best\_roster \))
                \( best\_roster = current\_roster \)
            Undo any changes made
        }
    }
}
To summarise, this hill climber tries swapping every assignment for every staff member with an agency staff member. It records the best resultant roster.

This should be run until it yields no more improvement.

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tues</th>
<th>Wed</th>
<th>Thurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tina</td>
<td>Night</td>
<td>Late</td>
<td>Early</td>
<td>Late</td>
</tr>
<tr>
<td>Shelly</td>
<td>Late</td>
<td>Late</td>
<td>Late</td>
<td>Early</td>
</tr>
<tr>
<td>Agency Staff 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Swap the offending shift into an agency staff member

Figure 31 Action of the agency swapping hill climber

One iteration has the following cost:

\[ C = b s x d \]

Where:
- \( C \) = the maximum number of evaluation operations per iteration
- \( b \) = the number of agency staff
- \( x \) = the number of non-agency staff
- \( s \) = the number of different shifts which may be worked by any staff member
- \( d \) = the number of days in a week.

Although there is a \( d^2 \) term present in the above formula, since \( d \) is usually 7, we can see that the algorithm will still scale linearly with regards to the size of the roster.

6.7.5 Shift swapping

This hill climber was designed to give some general optimisation to the roster, for all non-agency staff.

This hill climber tries to improve the roster by re-assignment of shifts, it finds an assignment for a non-agency staff member and replaces it with a different assignment for the same staff member.

The algorithm to achieve this is outlined in Figure 32 and the pseudo-code below:

Let \( \text{current}_\text{roster} \) = the current roster
Let \( \text{best}_\text{roster} \) = the best roster found so far. Initialise to \( \text{current}_\text{roster} \)
For each non-agency staff member “s” {
    For each day “d” {
        Remove any assignment for “s” on day “d” in roster “current_roster”
        Complete the roster, use algorithm in Section 6.7.3
        Evaluate the fitness of current_roster
        If current_roster is better than best_roster
            best_roster = current_roster
        Undo any changes made
    }
}
This should be repeated until it yields no more improvement.

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tues</th>
<th>Wed</th>
<th>Thurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tina</td>
<td>Night</td>
<td>Late</td>
<td>Early</td>
<td></td>
</tr>
<tr>
<td>Shelly</td>
<td>Late</td>
<td>Late</td>
<td>Late</td>
<td>Early</td>
</tr>
</tbody>
</table>

Delete a shift, and create a new one somewhere else

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tues</th>
<th>Wed</th>
<th>Thurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tina</td>
<td>Night</td>
<td>Late</td>
<td>Early</td>
<td>Early</td>
</tr>
<tr>
<td>Shelly</td>
<td>Late</td>
<td>Late</td>
<td>Late</td>
<td>Early</td>
</tr>
</tbody>
</table>

Figure 32 Action of the shift swapping hill climber

One iteration has the following cost:

$$C = sxd^2$$

Where:
- $C =$ the maximum number of evaluation operations per iteration
- $x =$ the number of non-agency staff
- $s =$ the number of different shifts which may be worked by any staff member
- $d =$ the number of days in a week.

So this again, will scale linearly.

This hill climber is very susceptible to becoming stuck in local minima so is totally inappropriate for solving the rostering problem alone.

6.7.6 Experimenting with shift swapping

If applied to a random roster, the “shift swapping” hill climber can run for many iterations, until eventually it reaches a plateau and halts, often giving a significant improvement in score.

If applied to a roster which has already been passed through the genetic algorithm, the hill climber can do little to improve the roster.

Also, the genetic algorithm typically yields a far better final score than is achievable with the hill climber alone.

So, the genetic algorithm, in some sense, produces rosters which are optimal with regards to the operations this hill climber performs. Similar results have been observed by Mitchell et al [39].
7 Implementation of the algorithm

7.1 Implementation language
The chosen language of implementation for this project was Java. This was chosen for several reasons:

- Cross-platform compatibility

- Java Applets can be created so the project can be made accessible from the World Wide Web (see www.nurserostering.com).

- The speed of execution of Java has increased in recent years to an acceptable level. Tests indicate as fast as C++ in some cases [33].

- Object-oriented language allows for modularity, extensibility and clear design of programs.

7.2 Implementation roadmap

Figure 33 is a UML class diagram [17] of the overall class structure used to implement the algorithm. The following sections of this report will outline the structure and functionality of the classes illustrated.
7.3 Roster representation and roster class

7.3.1 Overview
The roster class is one of the most important classes in the whole project. The roster class encapsulates the concept of a nurse duty roster, and common operations which may be performed upon rosters (e.g. crossover, mutation, randomisation, error checking etc.).

The roster class holds both the genetic and tabular representations of a week’s duty roster (as outlined in section 6.3) and references to all classes associated with the duty roster, including references to staff members, shifts and shift types.

7.3.2 Historic days
The roster class holds a number of days from the previous week’s roster. These days have been named “historic days” and solely exist to enable the detection of problems between weekly rosters. For example if a nurse works night duty at the end of one week, they cannot work day duty at the beginning of the next. The number of historic days required to detect all possible problems between weekly rosters will vary depending on the constraints, and must be externally assigned to a roster class.

7.3.3 Class contents
The below diagram illustrates the main components that compose the roster’s class. The lines on the diagram represent dependencies between the various components.

![Figure 34 Roster class internal structure](image-url)
The roster class contains the following simple data structures:

- **The “shift matrix”**
  - This is simple a 2D array of shifts, and represents a tabular form of the roster (section 6.3.2). The columns of this array represent days, while the rows represent staff members. Therefore any element in this array represents an assignment to a particular shift on a particular day for a particular staff member.
  - If the value is null, this represents a day off.

- **Historic day count**
  - This counts how many days are included in the shift matrix are from the week before the week represented by the roster class.

- **The “assignable matrix”**
  - This is a 2D array of Booleans. It’s purpose is to determine which slots in the roster have been fixed by the user and may not be altered. If an element in the “assignable matrix” is true, then this indicates that the corresponding element in the roster matrix can be freely changed by the program.
  - Note that “historic days” always have a false value in the assignable matrix. Since they come from previous weeks, they cannot be changed.

- **The “staff collection”**
  - This is a collection of all the staff member objects associated with the duty roster i.e. all the staff members to which the roster refers.

- **The “hours worked by…” array**
  - This is an array containing the maximum number of hours worked in a week by each staff member.
  - It serves 2 purposes:
    1. Avoids method calls to find the number of hours worked by each staff member.
    2. Allows independent adjustment of the number of hours worked in a week.

- **The “maximum assignable slots” array**
  - This is a 1D array, as large as the staff collection. It’s purpose is to store the number of slots for each staff member which are not marked as “false” in the “assignable matrix”. It represents how many days may be modified by the algorithm for each staff member. This data is calculable from the “assignable matrix”, but is cached in this array for maximum efficiency.

- **The “open days” array**
  - This is a 1D array of Booleans with one element for each day of the week. It indicates which days the ward is open. If the open days array contains false for any particular day, it can be said that the ward is not open that day, so no staff members can be assigned any shifts on that day. All cells for that day in the “assignable matrix” will be false.
This array also caches data calculable from the assignable matrix, but is present for maximum efficiency.

- The valid shifts collection
  This is compiled from the valid shifts for each staff member. It’s main use is for setting up of classes to evaluate the fitness of the roster.

- The chromosomes array.
  Holds the genetic representation associated with the roster.

### 7.3.4 UML Diagram

The information in the previous section is summarised here in a UML [17] diagram (Figure 35).

![UML Diagram](image)

**Figure 35 Roster class structure**
7.4 Constraints class suite

7.4.1 Overview
The constraints class suite contains a number of classes for counting the number of violations of constraints (section 5.6) in the roster.

In essence, each class in this suite encapsulates a single constraint (or group of constraints). This is to enable maximum modularity within the code, and prevent a large monolithic block of code for checking for violations. Furthermore, the class structure used makes it very simple to add new constraints. Some constraint classes are parameterised, so multiple instances of each constraint can be created, to encapsulate different constraints of the same type.

7.4.2 Software design
Figure 36 illustrates the class structure used in UML [17]. To maximise code re-usability and totally decouple other classes from the representation of constraints, a number of design patterns [4] have been used. All constraints implement a constraint interface, and there is a sub-interface for both soft and hard constraints.

Constraints can instantiate a “ViolationDescriptor” class. This class encapsulates the concept of a constraint violation and can be used to display the individual violations to the user. These violation descriptions may be obtained by calling the “getLastViolations” method in any constraint class.

The abstract class “RequestSoft” implements the “Template” design pattern [4], holding common code which all constraints regarding requests use.

The above classes are all in the “constraint” package. Soft constraints are kept in a soft constraint package and hard constraints in a hard constraint package, which are both sub packages of the constraint package.

Full information on the implementation of all these classes can be found in Appendix 4: Soft Constraints Implementation and Appendix 3: Hard Constraints Implementation.
Figure 36 Constraints class overview
7.5 **Evaluation Structure**

### 7.5.1 Overview

![Evaluation Structure Diagram]

Figure 37 Evaluation structure overview

Figure 37 illustrates the basic scheme for evaluating the “fitness” of a roster.

As can be seen, the evaluator evaluates a roster by computing some “score” or “utility” value [31,32] which represents fitness of the roster. This is done by using the constraints class suite (section 7.4). The “score” or “utility” may then be used to compare and rank rosters in terms of their intrinsic worth or usefulness to the ward.

A configurator object sets up an evaluator object so a computation of fitness can be made. The configurator is given a list of the required properties of the roster, which it uses to configure the evaluator. Each of these requirements is itself associated with a constraint.

To maintain encapsulation, visibility of the methods in “evaluator” is such that only the configurator can configure the evaluator.

### 7.5.2 Configurators

The main purpose of the “configurator” class is to assign appropriate weightings to constraints. In some sense, a weighting is a measure of the importance of a constraint with regards to the fitness of the roster. It is then up to the evaluator classes to compute an overall score for the roster using these weights and the constraint classes (section 7.4).

It can be said that the “configurator” class configures the evaluator class. The “configurator” is passed constraints and will then pass these to the evaluator class with the appropriate weightings.

The “configurator” sees constraints in terms of 3 distinct sets:
1. Hard constraints.
2. Constraints relating to requests from staff members
   These are soft constraints relating to a specific staff member.
3. “General” constraints
   These are soft constraints, not relating to any staff member in particular

Each individual constraint may have a weighting. All three sets of constraints
additionally have a “weighting quota”. This is the maximum weighting that can be
assigned to all the constraints in that set put together.

Adjustment of the weighting quotas will adjust the relative importance of each of the
sets of constraints. The quotas allow the user to specify that one set of constraints can
never be more important that another set. For example, if I set the weighting quota of
staff member requests to half that of hard constraints, then regardless of the number of
staff requests, or the relative importance of those staff requests, they will only ever
contribute half as much as hard constraints.

The reason for dividing the constraints into 3 categories, was to make the process
more intuitive for the user, and to differentiate between those constraints which are
universally favourable, and those which relate specifically to the requests of a specific
staff member.

7.5.3 Evaluators

The evaluator classes represent a function, which can evaluate the overall fitness of a
roster, i.e. a utility function [31].

Evaluators are separate classes in the program to allow for total customisation, and
easy interchanging between different evaluation functions. All inherit an abstract
superclass containing common methods.

A simple evaluation function is a weighted, linear function, as follows:

\[
F(T) = W_1 C_1(T) + W_2 C_2(T) + W_3 C_3(T) + W_4 C_4(T) + \ldots + W_n C_n(T)
\]

i.e.

\[
F(T) = \sum_{i=1}^{n} W_i \times C_i(T)
\]

Where

- \( T \) = a roster
- \( W_i \) = Some numeric weight
- \( C_i(T) \) = The number of violations of constraint “i” in roster \( T \)

The “score” methods of the constraint classes are used to compute \( C_i \) in the above
equation. The weightings \( (W_i) \) have been decided by the configurator. Thus,
evaluation of a roster is simply a case of iterating through the constraints passed by
the configurator and calling the “score” method in the appropriate constraint class,
then multiplying the result by the associated weight.
7.6 Twin capture and removal

7.6.1 Overview
I define “twin” as being two identical rosters.

The main difficulty with allowing twins in the population is that their inclusion results in a narrowing of the gene pool, causing the algorithm to eventually stagnate. When twins are mated, it will produce identical offspring (section 6.4), eventually leading to a population with little or no genetic variation.

In tests, after around 100 generations, it was found that most of the “elite” of the population were twins, therefore twins represent a significant problem.

At worst, twin detection is very computationally intensive. In a simple implementation, it requires comparing every cell of roster with every other roster.

The quantity of computation can be significantly reduced by using a hash function. If the information in a single roster can be compressed into a single integer number, then comparison of two rosters boils down to a comparison of two numbers, rather than the whole roster.

7.6.2 Calculating hash value
I have mathematically defined my hash function below:

Let:

\[ h(s, d, i) = (g(s)P(d))^{i+1} \]

Where:
\( P(n) \) = the \( n^{th} \) prime number.
\( s \) = a shift object (section 7.8)
\( g(s) \) = the internal Java language hashing function, which may give a hash value for any object based upon it’s type.
If \( s = \text{null} \) we can assume \( g(s) = 0 \)

\( d \in \mathbb{Z} \)
\( i \in \mathbb{Z} \)

\[ H(R) = \sum_{i} \sum_{d} h(t_R(d, i), d, i) \]

Where:
\( H(R) \) is the hash value of roster \( R \)
\( t_R(d, i) \) is the shift assigned on day \( d \) for staff member \( i \) in roster \( R \), if there is no shift assignment for this staff member on this day, then let \( t_R(d, i) = \text{null} \)
7.6.3 Optimisation

I realised that the hash function being used was computationally intensive to calculate. Therefore I replaced the power calculations in $h$ with binary shifts which may be executed more rapidly, to give the following function:

$$h(s,d,i) = (g(s) \times P(i)) \ll (2d)$$

where “$\ll$” represents a binary shift left, in this case of 2d bits.

This optimises further still to:

$$h(s,d,i) = (g(s) \times P(i)) \ll (d\ll1)$$

The hash value is defined to be 64 bits in size. The roster requires more than 64 bits to store all the information in it, therefore the hash function is imperfect.

The hash value is computed when translating between the tabular and genetic representations for a roster (section 6.3).

7.6.4 Hash function testing

The hash function was run extensively, checking rosters of varying sizes. It was expected that the hash function would detect more twins than were actually present, due to it’s imperfect nature. Hash collisions will cause non-identical rosters to take the same hash value.

<table>
<thead>
<tr>
<th>Actual number of twins</th>
<th>Twins found by hash</th>
<th>Checks performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4225072</td>
<td>4235833</td>
<td>35697986</td>
</tr>
</tbody>
</table>

Rosters found to be twins by hash function, which were not twins: 10761

As a percentage of twins found: 0.25%

Coming to the conclusion that two rosters are identical when they are in fact different in only 0.25% of cases is an acceptable proportion to use this hash function in the algorithm. Collisions will only result in 0.25% of the population being incorrectly removed in any run of the algorithm.

7.6.5 Twin removal experiment

To find whether twin removal makes a significant difference, I needed to run a number of scientific experiments.

I decided to run 4 tests under identical conditions to evaluate the effectiveness of twin removal. I examined the fitness or “score” (section 7.5) of the final roster obtained by the genetic algorithm alone. I also decided the variance in results would also be a useful for comparison. A lower variance indicates that the genetic algorithm is producing more consistent results.

The following tests were run:

1. Entirely deleting twins and replacing with a new randomised roster (*Full twin deletion*)
2. Mutating any twins found until no twins exist in the population (*Full twin removal*)
3. Removing any twins found during crossover (*Simple twin removal*).
4. Not removing twins at all (used as a benchmark).

Experimental conditions were identical to mutation testing (appendix 1), except the test was run to 1000 generations, it was thought that the algorithm would reach a plateau by this point. Each test was run 15 times so error bounds could be generated.

### 7.6.6 Experimental results

![Figure 38 Twin removal results](image)

Figure 38 illustrates the experimental results. The Y-Axis represents the final “score” of the roster created by the genetic algorithm. Recall that lower score implies a better roster. The stalks are 95% confidence intervals.

It was also observed that with full twin removal, changes in the best solution occur more often, especially towards the end of a test run.

### 7.6.7 Conclusions

It can be seen that using twin removal makes a significant difference to the final score achieved by the genetic algorithm. Full twin removal / deletion gives better results and a lower variance (i.e. more consistency) than simple twin removal. Therefore it would appear that more twins we eliminate, the better the results.

Full twin removal offers about a 25% improvement on no twin removal in the experiment, and about 10% improvement on simple twin removal. The computation required for full twin removal is not significantly more than simple twin removal, which in itself, is not significantly more computationally intensive than no twin removal, therefore it is feasible to implement this in the project.

Full twin deletion, produces good results, but not as good as full twin removal. This is not entirely unexpected. By completely deleting twins and replacing them with entirely new rosters, the search of the genetic algorithm can become undirected as parts of the population will be replaced by completely unrelated rosters. Thus full twin removal was used in this project.
7.7 **Staff member representation**

7.7.1 **Overview**

For the purposes of the project, it was decided that there should be exactly 2 types of staff member:

- Non-agency staff members
- Agency staff members

The following information must be held about any staff member:

- The name of the staff member
- The skill level of the staff member.
- Any qualifications or specific skills the staff member has.
- The maximum number of hours they can work in a week.
- The shifts they may work
- The days they are not available to work on
- Whether they actually count towards staffing levels (some administrative staff and ward managers need to be included in a roster but do not count towards staffing levels on a ward).

It is assumed that any staff member with a skill level above a specified “trained staff level” is trained.

Each staff member is represented by an instance of a “Staff Member” class.
### 7.7.2 Class structure

#### Figure 39 Staff member class diagram

As can be seen from Figure 39, the two types of staff member are both modelled by classes. These derive from an abstract super-type “StaffMember”, this means that every staff member must either be a “regular” (i.e. non-agency) staff member, or an “agency” staff member.

Any special skill that a staff member has (for example trained in IV injections, Royal Sick Children’s Nurse, trained in endoscopies etc.) are encapsulated within a “SkillsDescriptor” class, which simply holds a textual description of that skill and a numeric identifier (m_uid), for rapid comparison.

Finally, the StaffMemberComparator class provides a simple method for comparing staff members for sorting, so they may be sorted by skill level, then by name.
7.8  Shift structure and customisation

7.8.1  Overview

To represent a shift, it was decided that each shift, should be represented by it’s own single instance of a “shift” class. Furthermore, it was decided to treat paid, or unpaid leave as a type of shift. This was an abstraction to allow the algorithms to view a roster in a consistent manner. We can think of unpaid leave, as a shift with zero duration, and paid leave as a shift with some duration, which requires no staffing.

Any shift carries references to the shifts which it overlaps, and those shifts which cannot be placed before or after it (m_illegalBefore and m_illegalAfter on Figure 40).

After some initial analysis of the problem, it was concluded that shifts can have different types:
- Day shifts
- Night shifts
- Leave shifts

There are many more types possible, and it was decided that for maximum flexibility, it should be easy for a user to define new shift types, and assign shifts to certain types. Therefore a shift carries a reference to a class containing a description of it’s type. This can be altered to change the type of the shift, at run time. To check the type of any shift, the reference to the shift type description can be compared. Leave is a special shift type, as it carries special significance with the algorithm. Figure 40 shows the classes used to implement this scheme.

Initially I implemented shift typing using Java’s built in type checking by sub-classing of the Shift class. However, this could not be dynamically changed at run time and limited the flexibility of the system.
7.9 Genetic Problem Solvers

7.9.1 Overview

The genetic algorithm used in this project, uses a co-operating, multi-threaded approach. This approach was tried in part by Konstantin Boukreev in his freeware sample program [9], to solve the travelling salesperson problem. This program used multiple highly co-operating threads, and was very successful in solving the problem. This inspired me to trial a similar approach on rostering.

The basic behaviour and structure of the algorithm built for this project is outlined in Figure 41.

![Figure 41 Design of multi-threaded genetic algorithm](image)

The algorithm uses multiple “worker” threads, each of which is capable of solving the rostering problem itself. Each thread runs the code for a genetic algorithm, on it’s own population of rosters.

This is partially where this project differs from Konstantin Boukreev’s work on a multi-threaded genetic algorithm [9]. His approach moved solutions between threads, thus meaning that the space searched by each thread was similar. In this project, I will keep the threads as separate as possible. There are several reasons for this:

1. Each thread should operate in a different region of the search space and have a totally different population. This helps the algorithm escape if it becomes stuck in a local minima, and increases the portion of the search space that is covered by the algorithm.

If all threads had similar populations, it would be analogous to having a single thread with a larger population, thus negating the purpose of having >1 thread.
2. Interference between threads is prevented, thus race conditions and other concurrency errors are avoided.

3. If all the threads are totally independent, then the Java VM can choose how to execute them, and all ordering of instructions between threads will be irrelevant. Furthermore synchronising threads around critical sections is a difficult, error prone, and time consuming activity, which can lead to bottlenecks.

I wanted to apply evolutionary processes to the threads, as well as the rosters, in the hope of finding a better solution. **In other words my idea was to have evolutionary processes acting at the thread level, as well as at the roster level.**

Therefore the algorithm starts with a population of threads. These threads are run in competition with each other, the best solution being selected from all the threads and presented to the user. If a thread does not produce a good solution, for a certain number of generations, it is killed and restarted. The rationale behind this is that the thread must be stuck in a sub-optimal part of the search space. By restarting it, with a fresh population, the thread will jump to different part of the search space.

Multi-threading alone was not found to produce much of an improvement. This was a very disappointing result. I did not want to scrap the multi-threaded part of the algorithm, so after some careful consideration, the idea of **“thread culling”** was added to the algorithm. The concept is “survival of the fittest thread” and works as follows:

- Start with a large population of threads (>4)
- After each thread has run to a number of generations \(G\), the threads are ranked, by the fitness of the best roster they have produced.
- The top \(N\) threads are preserved and the rest are deleted.

An example of this process is illustrated in Figure 43.

I am assuming that if a thread is producing the best solution after \(G\) generations, it is more likely to produce a better final solution. This seems reasonable as the majority of improvement in a run of a genetic algorithm occurs at the beginning of a run [2], thus \(G\) should be small.

![Figure 42 Thread culling and scope](image-url)

For the first \(G\) generations before thread culling, the scope of the genetic algorithm’s search is significantly larger due to the large number of threads. Then after thread culling, the scope is narrowed, focussing the algorithm on the best rosters. This is illustrated in Figure 42.
7.9.2 The core of the genetic algorithm

At its most basic, each thread performs the following sequence of operations:

1. Evaluate all of the rosters in the thread’s population using the evaluation class suite.
2. Order the population by fitness. Preserve the top N% of the population and discard the rest. A normal value for N, is around 15% as used in [9,10].
3. Mate the remaining rosters, using the crossover operation (section 6.4) until the population is re-built to its previous level.
4. Mutate a random selection of the population (section 6.5).
5. Repeat the process from step 1. Each iteration is termed a “generation”.

This is the classical way in which a genetic algorithm functions, as used in [9,10] and described in [2].

7.9.3 Genetic solver implementation and class structure

The classes and possible method invocations which constitute the core of the genetic algorithm are illustrated in Figure 44. A full UML diagram is given in Figure 49. To start the genetic algorithm, an application programmer calls a “solve” method on a GeneticSolver class. Each thread (WorkerThread class) is set off with its own initial population, and runs as normal, in the manner described in the previous section.

When the best roster in a thread’s population changes, the thread calls “SolutionImproved” on a co-ordinating monitor class “GeneticSolver”. This method
takes the roster and compares it with the “global best” roster. If it is better, the “global best” roster is replaced.

Each thread counts the number of generations since it last changed the “global best” roster. If this number exceeds a certain threshold value, the thread is terminated. This will kill off threads producing poor solutions. The threads may then be restarted with a new population.

The algorithm terminates when the thread producing the best rosters terminates. At this point, all remaining threads are terminated, and “solve” method returns the best roster found.

7.9.4 Performance testing

7.9.4.1 Experimental conditions
See Appendix 2: Test Conditions for thread testing.

7.9.4.2 Multi-thread performance
For initial testing of the effect of multiple threads, I decided to examine the effect of multiple threads on system performance, without any “thread culling”. A measure of performance had to be chosen. It was decided that the time taken for N threads to run the genetic algorithm to 200 generations (each), would be the performance measure.

Several tests were conducted on the computing department’s machines. The most important results are shown in Figure 45 and Figure 46. The stalks represent 95% confidence intervals. The Y-axis is measured in milliseconds.

![Figure 45 Run time on Windows, high thread priority](image-url)
The “expected time” to execute the test is calculated as follows:

\[
(\text{Time for 1 thread to run test}) \times (\text{Number of threads in test})
\]

This disregards any overhead for thread management, so one might expect it is a slight underestimate.

### 7.9.4.3 Thread culling

To test “thread culling”, I started the algorithm with a population of 5 threads. After 25 generations, all the threads except the one producing the best roster were killed. The remaining thread was then allowed to run to normal termination, and the final fitness of the roster was noted. Each test was repeated 30 times. Figure 47 shows the experimental results. The stalks on the graph represent 95% confidence intervals. Recall that a low final score indicates a better roster. For testing without culling I used a single thread.
In Figure 48 and Figure 47, we see a 14% increase in run time for the tests with culling, but a 1% improvement in the final solution fitness. The error bounds indicate this is a significant change, at the 95% confidence level. This final 1% change in score will probably represent the fulfilling of staff requests and soft constraints.

7.9.4.4 Explanations and Conclusion

On a single processor machine, it can be seen that the multi-threaded version of the algorithm is no better, in terms of run time, than running the algorithm several times (Figure 45). In fact at above 3 threads, the performance is notably worse. Therefore more than 3 threads should ideally not be used on a single CPU computer.

The results on a multi-CPU server were somewhat disappointing (Figure 46). Unsurprisingly, the run time for multiple threads on a multi-CPU server is faster than multiple runs with a single thread, however the difference is not as great as expected. This is almost certainly due to the overheads of thread switching. Still, the multi-threaded implementation with 2 or 3 threads runs significantly faster than on a comparable single processor computer.

It would appear from experimental results the algorithm can produce a better solution, with “thread culling” (Figure 47). By eliminating threads producing poor solutions close to the beginning of a run, we can eliminate poor populations of rosters from our search. Note this approach does increase the run-time of the algorithm, but not dramatically. Therefore I conclude that thread culling is a worthwhile addition to the algorithm. However, the results of multi-threading the algorithm where on the whole disappointing, not producing dramatically better solutions.
7.10 Problem solving architecture

7.10.1 Overview
It was decided that all rostering algorithms used in the project should be encapsulated in classes exposing a standardised interface. A single method “solve” is exposed which takes a roster as it’s arguments. A call to “solve” applies the algorithm encapsulated by the class to the given roster, and returns an improved roster. Thus, the user interface and evaluator are entirely decoupled from the problem solver itself. This makes it easy to add other types of solver if required in future e.g. a Tabu search algorithm [18].

7.10.2 Roster improvements and handlers
Any call to the “solve” method may take a long time to return. During this time we may want to show the user the best result generated so far, so they can make a decision on when to terminate the algorithm.

After consulting some designs pattern literature, I decided the best course of action here would be to implement the “chain of responsibility” design pattern [4].

I regarded an algorithm improving a roster as an event. This event causes various handlers to be notified. The chain of responsibility design pattern allows multiple handlers to be notified and ordered if required (for example we may want to write the result to disk before displaying them to the user). The chain also adds flexibility, allowing event handlers to be added, removed and altered without affecting the algorithm classes at all.

Figure 49 illustrates the implementation of the solving structure. The chain of responsibility [4] is implemented by the “ImprovementNotificationHandler” class.
7.10.3 Full class structure

Figure 49 Solver full class structure
8 The User interface

8.1 Building the GUI

8.1.1 Overview and aims
As outlined in the introduction to this project, it was essential that the application developed had a user-friendly graphical user interface.

The main aim was to create an interface with the following properties:
• Easily understandable to a Nurse, so no training is required to interpret the output.
• Easy to use the basic, common, functionality, such that any user can instantly use the software.
• The interface should not be daunting, and all the complexity of the algorithm should be totally hidden by the user.

To achieve this, it was decided to take a task-oriented approach [34] to user centred systems design [35,36,37]. This would require:
• Analysis of the task of rostering.
• Analysis of the way the task is currently performed (i.e. analysis of the pen and paper method).
• Prototyping and frequent evaluation sessions with nursing staff.

To use the user interface, please visit http://www.nurserostering.com, further details of the user interface are also available in the user’s manual attached to this report.

8.1.2 The current method of rostering
I decided to examine the paper grid used for rostering on Harold Ward. A number of tables were also used, which hold information on the staff members of a ward, and other ward-related information. I was keen to examine these documents, and see if they could be incorporated into the system.
A full list of available staff members

Dates

Allocated shifts, written in an abbreviated form

Trained staff sheet

Untrained staff sheet

Holidays marked with special notation

Requests marked with an "R" written on to sheet before rostering process

Figure 50 Harold ward sample roster
Figure 50 shows a real nursing duty roster for Harold Ward, Princess Alexandra Hospital, Harlow, Essex, UK. The duty roster was for the month of February 2004.

As can be seen from the annotations, the roster is in tabular form (section 6.3.2), with columns representing days, and rows representing staff members. Interestingly, both the requests for shifts and the shift assignments themselves are written upon the roster.

Although it is not pictured, it is common for the nurse to write the number of staff members scheduled for each shift at the bottom of a day column. It is also common to write the number of hours worked in the roster by each staff member in the rightmost column.

Figure 51 illustrates the document which accompanied this roster.

![Figure 51 Document accompanying roster](image)

As can be seen, the above document mainly contains information regarding the number of hours worked by each staff member.

WTE is a measure of the number of hours worked by a staff member in a week and 1.0 WTE = 37.5 hours (i.e. one full working week).
Finally note the comments section at the end which contains a number of requirements for the roster (i.e. constraints). It also contains some indications for soft constraints such as “two D grades are very new”. A human could infer some soft constraints regarding these D grades, for example.

One final sample roster is shown in Figure 52.

![Figure 52 Sample duty roster](image)

This roster is very similar to the other rosters pictured, except it has been produced using Microsoft Excel. This implies that at least some of the user base will be familiar with spreadsheet tools, and spreadsheet style interfaces.

This duty roster is taken from Bluebell Ward at the Lister Hospital, Stevenage, UK.

### 8.1.3 Task analysis

Some formal task analysis of the current method of rostering was needed, so the task could be performed in a similar manner in the project, therefore making the system much more intuitive for the user.

A hierarchical task diagram (Figure 53) was created after liaison with the following nurses:

- Pauline Howson  
  Nursing Sister, Alexandra Day Unit Princess Alexandra Hospital, Harlow, Essex;
- Catherine Beadle  
  Ward Manager, Harold Ward, Princess Alexandra Hospital, Harlow, Essex;
- Devi Heath  
  Nursing Sister, Bluebell Ward, Lister Hospital, Stevenage

74
Rostering

1. Write documentation
   1.1 Write number of hours worked by each staff member
   1.2 Describe requirements of roster, and other information to be included (i.e. hard and soft constraints)

2. Nurses write requests on draft roster
   2.1 Assign staff members to shifts
   3.1 Count number of staff members on each shift. Write this at the bottom of the roster
   3.2 Count number of hours worked by each staff member, write this at the side of the roster
   3.3 Check roster against documents, and for problems and undesirable shift combinations

3. Create roster
   3.1.1 Choose a staff member + a day
   3.1.2 Choose a shift

Figure 53 Current process task diagram
The task diagram is read left to right. The numbers represent the sequence of actions. Further down the hierarchy represents more specific actions whereas the higher levels of the hierarchy represent more general actions.

As can be seen the main process consists of
1. Layout requirements
2. Putting requests onto a roster
3. Editing the roster, and checking this against the requirements.

Note that this third step may need to be repeated a very large number of times to achieve a satisfactory roster. Additionally the first step is not required every time a roster is created. Once the accompanying documentation is written, it may be used for every roster for that ward, until situations change.

It can also be noted that a significant expenditure of time in the rostering process, is counting numbers of hours and numbers of people on shifts and writing these at the bottom of the roster for error checking purposes.

For 4 weeks this process typically takes around 12 hours.

**8.1.4 Task design**

To begin with, a task diagram for the system was drawn (Figure 54) based around the current method of rostering (Figure 53).
Rostering

1. Supply requirements to software
   - 1.1 Input number of hours worked by each staff member
   - 1.2 Formally describe exact requirements of timetable to the software package

2. Input requirements into system
   - 2.1 Choose a staff member + a day
   - 2.2 Select a shift
   - 2.3 Select a priority for the request

3. Create roster
   - 3.1 (Optional) Assign staff members to shifts
   - 3.2 Use automatic generation
   - 3.3 Check roster against documents, and for problems and undesirable shift combinations
     - 3.1.1 Choose a staff member + a day
     - 3.1.2 Choose a shift

Figure 54 System task diagram
In Figure 54, several tasks from pen and paper rostering have been replaced by data input. Nurses no longer have to write supporting documentation for a roster, instead they enter the requirements into the system. Likewise, instead of writing requests on a roster, nurses must enter them into the system. Requirements and constraints upon the roster, must be simplified, and specified to the system in a formal manner.

Once the data has been input into the system, it may be used for every roster for that ward, until situations change.

These new tasks may be unfamiliar to users, but they are essential for the system to be practical. Therefore, I decided effort should be made in these areas to create a user interface which is as user-friendly as possible. Luckily, these tasks do not need to performed for every roster, so they are not the “common case”.

The final task of creating a roster, initially looks more complex than the existing pen and paper system. However, manual assignment of shifts is optional. Furthermore, the counting tasks have been eliminated, as the computer can easily perform these and display the results on screen. The automatic generation task, will perform the rostering with no user intervention.

A trade off had to be made between the exactness and functionality of the new system, and it’s usability. Increasing the functionality, would lead to more complex tasks for the user and hence make the system less usable with a steeper learning curve.

8.1.5 Initial user interface design

It was decided the main interface for inputting requests, and shift assignments (tasks 2+3), should mimic as closely as possible the current roster sheets, thus providing the user with an instantly familiar interface, so no re-training would be necessary. Also this would make the system less daunting for an very inexperienced user.

It was noted, that some hospitals are starting to use Microsoft Excel for rostering (as in Figure 52). Therefore it was decided that at least some of the user base must be familiar with Microsoft Excel, so an interface similar to Excel’s would be easier to use.

A critical part of the user interface and a basic requirement of the project, is the ability to write shift assignments directly on to the roster. This allows the user to solve the entire rostering problem manually, should they so desire. It also allows the user to make specific assignments for those staff members who work set patterns.

Through user consultation the decision was taken to provide 3 different views of the roster.

- The week view
  This is useful for editing a roster as it easily fitted on screen.

- The month view
  This is not be useful for editing, but good for giving an overview.

- A transposed view
  As illustrated in Figure 55
  This enables the user to see who is working any given shift.
For deciding priorities of constraints it was proposed that the user should choose from a list of phrases, which could then be represented by numerical weights (Figure 56). This was judged easier to understand for a user with no concept of the workings of the algorithm. It is also less daunting than entering a large number of numerical weightings.

Finally for prioritising between Hard and Soft Constraints, it was decided that more continuous scale was required. Sliders were therefore proposed as a easy method of entering the relative priorities (Figure 57). Sliding the slider to the left increases the priority of a set of constraints and sliding it to the right decreases the priority.

### 8.1.6 Implementation technologies

It was decided that Java’s Swing [38] would be a reasonable technology for use in the project. This was decided because of the following reasons:

- Part of the usual Java Runtime Environment shipped by Sun Microsystems and used on the majority of Java capable computers.
- Has a large, well designed class hierarchy which can easily be expanded and modified.
- Does not use the “graphical widgets” from the user’s operating system, so it can be guaranteed that the interface will look the same on all platforms.
- Mature, well tested, technology with many applications already using Swing (Sun’s NetBeans IDE; IBM’s Eclipse; Opera web browser).

### 8.1.7 Initial prototypes

It was decided that a number of successive prototype interfaces should be created and shown to the user group, to determine their needs and any further requirements. This
was also in line with usual design practices [35,36,37]. The first prototype interface created is shown in Figure 58.

The main screen was based around a one-week tabular view of a roster. After showing this to the same user group as mentioned previously, the following points arose:

- The staff level indicators at the base of the display were confusing and difficult to read.
- More information on the staff members should be included on the main display, such as their skills.
- The display lacked general user friendliness. It was suggested by Ward Manager Catherine Beadle, that an improved colour scheme and inclusion of other graphical mnemonics such as icons would be useful.

![Figure 58 First prototype](image)

The main screen was based around a one-week tabular view of a roster. After showing this to the same user group as mentioned previously, the following points arose:

- The staff level indicators at the base of the display were confusing and difficult to read.
- More information on the staff members should be included on the main display, such as their skills.
- The display lacked general user friendliness. It was suggested by Ward Manager Catherine Beadle, that an improved colour scheme and inclusion of other graphical mnemonics such as icons would be useful.

![Figure 59 Field testing with the users](image)

### 8.1.8 Second prototype

After some redesign work, a second prototype interface was proposed. A number of tabs at the top of the interface allow the user to switch between a number of different views.
Figure 60 The week view, second prototype

Figure 60 illustrates the week view, this is analogous to one week of the paper roster. At the bottom of the view, staff counts are present, this time presented in a more logical, tabular form, showing minimum staff levels and current staff levels on each day. Consultation with Staff Nurse Devi Heath indicated that alone, this table is a useful timesaving tool, especially when assessing or manually generating a roster.

Additional information on staff members, and their working hours is shown in the left hand column. Controls at the bottom allow navigation between weeks. Users can use this interface to view the roster, generate rosters automatically and edit the rosters.

Clicking on a cell brings up a menu which allows the user to select a shift assignment or requested shift. This menu is pictured in Figure 61.
A “requirements view” was designed to enter constraints and is shown in Figure 62. The “requirements” entered in this screen directly represent constraints for the genetic algorithm. Each “requirement” on the display actually is an instance of a constraint class (section 7.4). The user can add and remove constraints and, assign them priorities. Upon adding constraints, there are supplementary screens for setting up parameters of each constraint, such as selecting which shifts are undesirable when placed together.

For convenience and maximum clarity, the constraints are split into three categories. These categories are:

**“Essential” requirements**
These represent hard constraints.

**General requirements**
These represent soft constraints, not specific to any particular staff member.

**Staff requirements**
These are soft constraints related to staff requests.

There is a full discussion of these categories of constraints in section 7.5.2.

The user can also set the relative importance of the various categories of constraint using the sliders at the base. For example, by setting the slider for “Essential requirements” to half the value of “General requirements”, in total “essential requirements” will only ever contribute half as much as “general requirements” to the fitness of a roster.
Figure 63 The staff view

Figure 63 shows the staff list screen, listing all staff and their details. Staff can be added and edited and there is a table for entering staff skills / qualifications.

Figure 64 The ward sheet view

The ward sheet view (Figure 64) represents the transposed view outlined in section 8.1.5
The month view (Figure 65), is a view which shows the shift assignments for all staff across an entire month.

### 8.1.9 Final prototype

During further user consultations, it was decided that the interface for entering constraints was too complex for the normal user to use.

Constraints were renamed “rules”, as this is closer to the true meaning of the data entered in this screen. The constraints were adjusted to appear in the display in decreasing order of importance, so the user can instantly see the order of importance upon their constraints. This new screen is shown in Figure 66. Extra information on the constraints was made apparent as well as help text, to guide the novice.

To aid in the rapid addition of new constraints a “Rule wizard” was introduced (Figure 67), which offers a number of English sentences, which the user can select.
Each of these sentences makes a statement about constraints on the roster, and once the user has selected one, an appropriate constraint is constructed.

Figure 67 The Rule Wizard

To help the user comprehend the true meaning of each constraint, a feature was added to explain the constraints in "plain English" (Figure 68). This uses a set list of sentences, which are combined with the information in each constraint to produce a sentence which approximately explains the constraint in words that should be clear to the novice user.

Figure 68 Explaining rules in plain English

While generating a roster, the user is presented with a "thinking" dialog (Figure 69) which indicates the progress of the algorithm and the fitness of the best roster found. Indications of the algorithm’s expected run time are also displayed.

Figure 69 The thinking dialog

The prioritisation sliders shown in Figure 62, were moved to their own screen, shown in Figure 70.

Figure 70 Sliders view
By clicking on examine in the week view, or when automatic generation is occurring, a problems window will pop up.

The problems window lists any problems (i.e. violations) in the roster, and clicking on one will highlight the location of the problem (as pictured in Figure 71). A dialog was added so the user could add, edit and customise shifts through the user interface. This is illustrated in Figure 72.
Finally undo and redo facilities were added to the interface (Figure 73) to allow easy recovery from mistakes, the colour scheme was changed to a more businesslike grey and a full HTML driven online help system (Figure 74) was added.

The application was also made available on the web using a Java Applet (Figure 75). This enables the user to access the software and run it in their Java-compatible web browser without having to install anything on their computer. It also makes conducting a wide scale beta/user acceptance test easier. The applet and this report are available at the project website http://www.nurserostering.com. Also the user guide at the back of this report gives a full explanation of user interface.
8.2 Basic Client Class structure

This section gives a brief overview of the software design layout used in the GUI.

It was decided to base the design of the software around the “Model View Controller” (MVC) paradigm [4]. This would allow the document code, the views and the control code to be completely decoupled, particularly useful in this case due to the number of different views.

The packages which make up the GUI are broadly as depicted in Figure 76. The lines attempt to show dependencies between the packages.

The frame package contains all classes regarding the overall content of the main frame of the GUI, including the main frame class itself.

The view package, as described in the following section holds all classes required for viewing roster documents. The classes correspond to the view part of the MVC paradigm [4].

The “viewcontroller” package provides a number of classes which control the views in the view package, and allow editing and automatic generation. The classes correspond to the control part of the MVC paradigm [4].

The dialogs package holds all classes which represent dialog boxes. The sub packages, categorise these dialogs for maximum clarity. Each class in this package, is itself a dialog box.

Finally, the web package contains a single class which is a java applet, which may be used to launch the main frame of the GUI from within a browser (www.nurserostering.com).

Throughout all the packages, standard Java Swing programming techniques [38] have been used, and much of the code deals with layout of the components and their behaviour. I will therefore not go into detail about these classes.
8.3 View class structure

8.3.1 Overview
Since several different types of view are required in the application, it was decided that there should be a view abstract class which implemented common methods and exposed a common interface. This cut down on duplication and maximised flexibility. The class structure is shown in Figure 77:

8.3.2 CalendarView class
The calendar view class holds a number of attributes and methods associated with all views. The most critical is the attachDocument method, which allows a view to be attached to a document, and therefore display the contents of that document.

8.3.3 GenericGridView
This class contains basic code for drawing a grid. It has a number of unimplemented abstract methods for displaying text in that grid and gets the grid dimensions from a number of attributes in the class (template design pattern [4])

8.3.4 StaffGridCalendarView
This class extends GenericGridView, and represents a grid which always has staff members down the right hand side.

8.3.5 TransposedView, MonthCalendarView and WeekCalendarView
These three classes are concrete view classes which display information to the user, either for a whole week (Figure 73), a whole month (Figure 65), or for a week in a transposed format (Figure 64).
8.4 Document representation

Ideally a “document” in the project should represent and arbitrary number of weeks, and should be clear and easy to comprehend for the application programmer. A roster for any week can then be generated from this document.

The document package represents the “Model” part of the MVC paradigm [4].

It was decided to keep the algorithmic representation of a duty roster (sections 6.3 and 7.2) and the document representation completely separate. Mainly this was because the algorithm’s representation for a roster is:

(a) highly optimised and
(b) Only represents a single week

The basic structure of a “document” is shown in Figure 78:

![Figure 78 Document structure overview](image)

I decided to model the document as a collection of days. Each day contains 0 or more assignments. These assignments or “slots” occur on a certain day, are for a certain staff member, and hold a certain shift. Therefore the document holds the exact same information, as the tabular representation of a roster (section 6.3.2). Attached to a document, are all the shifts, types of shift, staff members and constraints referenced in the roster. In this way a document contains literally everything to do with the rostering problem.

By modelling the document as a collection of days, there can be an arbitrary number of days (storage space permitting). We only need to store days which have 1 or more assignments or “slots”.

90
All document classes and referenced classes are “serializable”. This mechanism in Java allows instances of the classes to be saved to disk, and re-loaded, without any need to define a custom file type and structure.

The classes for the document are held in their own separate “rostercalendar” package, as the document representation is seen to transcend all possible interfaces, and all algorithms.

Figure 79 Document structure UML Diagram

The Figure 79 illustrates the classes used to implement the document.

CalendarDocument acts like a “façade” [4], abstracting away the structure and the complexity of the classes it references. The programmer can perform nearly all document actions they need through CalendarDocument. Furthermore the algorithm or GUI programmer need not be concerned with the internal structure of a document.
9 Evaluation and testing

9.1 Applying the project to real ward situations
It was decided to apply the project to a number of different rosters, from different wards, with wildly differing shift structures. In this section I will present the results of testing on the rosters from 2 wards in particular.

9.1.1 Bluebell ward
For my initial testing, I took a week’s duty roster from Bluebell ward in Lister Hospital Stevenage. This particular ward had 2 main shifts, a “Long day” shift, covering the majority of the day and a “Night” shift which was longer, covering the night. A long day could be split down into an “Early” (6am – 1pm) and “Late” shift (1pm – 7pm) if required.

Bluebell ward is a well staffed ward, with a sufficient trained and untrained staff to cover every hour of every day with the correct staffing level. I was therefore interested to see how quickly the project could arrive at a solution in an uncomplicated situation such as this.

On the following two pages are both the human-generated roster (in the form of an Excel spreadsheet) along with the computer generated equivalent. Both were created from the same set of requirements and staff requests.

I have concentrated on the period 4 January to 11 January inclusive, this region is bounded by the black box in Figure 80. The roster automatically generated by this project is shown in Figure 81.

The constraints present on the computer generated version were agreed with Devi Heath, a Nursing Sister on Bluebell ward as being an accurate representation of the constraints usually taken under consideration when rostering.

Unfortunately with this dataset, the names of the staff members were not available. However, their grading was, so it was still possible to construct an accurate roster.
### Figure 80 Roster for bluebell ward

**Key:**

- D = Day shift
- N = Night shift
- L = Late Shift
- E = Early shift
- C = Clinic (not on the ward)
- X or empty cell = Day off
- r = Request (for a day off)
### Roster generated by this project

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>04-Jan</th>
<th>05-Jan</th>
<th>06-Jan</th>
<th>07-Jan</th>
<th>08-Jan</th>
<th>09-Jan</th>
<th>10-Jan</th>
<th>11-Jan</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>E10</td>
<td>D</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>D</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>E11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E12</td>
<td>R</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>E6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>E7</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>E8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>E9</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>D13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>D14</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td>R</td>
<td>R</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>D15</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>D16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>D17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AL</td>
</tr>
<tr>
<td>D18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>D19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>D20</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>HCA1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>HCA2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>HCA3</td>
<td>R</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HCA4</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>Do</td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>HCA5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
</tbody>
</table>

**Figure 81 Roster for Bluebell ward generated by this project**

**Key:**
- D = Day shift
- N = Night shift
- L = Late shift
- E = Early shift
- Do or empty cell = Day off
- R = Request (for a day off)

**Note:** The D-grade staff member who was on placement in Figure 80 Roster for bluebell ward was excluded from this roster.
Generating the roster took the computer only about 2-3 minutes, however to perfect the constraints and the weighting of the constraints took somewhat longer, and several rosters were generated. The computerised process took about 40 minutes - 1 hour including the time taken to specify and weight the constraints. Subsequent weeks were faster to generate as the constraints did not need re-specifying.

It is worth noting that a nurse can perform other tasks while the roster is being automatically generated by the computer. Therefore the run time of the algorithm is not a very important factor.

Devi Heath estimated that an entire month could take as long as 12 hours to roster, from start to finish. This would be about 3 hours per week, so we can the software yields a dramatic timesaving.

In consultation, it appeared that the computer had created a satisfactory roster, resulting in good staffing levels throughout the week without violating any important hard or soft constraints. However, it is also important to note that this particular roster was for a well staffed ward, so did not test the application’s performance on a difficult, understaffed NHS ward, where automatic generation is most useful.

9.1.2 Harold ward
Harold Ward in the Princess Alexandra Hospital (Essex), has a slightly different shift structure to Bluebell ward. Although Harold ward has around 30 staff members, this is still not sufficient for it’s size. Harold ward’s roster is also more constrained than Bluebell ward’s. Staff retention has become a problem on Harold Ward, so in an effort to keep staff happy, any requests are always granted.

The names of the shifts on Harold Ward are the same as on Bluebell ward, however the durations are different, and “Long Day” shifts are uncommon. Furthermore Harold ward requires a different mix of skills, and one RSCN trained staff member on at all times.

Currently on Harold ward, due to the large number of staff, the rostering process is carried out by 2 members of staff, each spending a significant period to create the roster (several hours per week).

I wanted to see how my algorithm would perform across multiple weeks, so I chose a 1 month period.

I also took the opportunity for another ward visit so the roster could be shown to the staff members concerned.
The constraints entered into the algorithm for this roster were agreed by Ward Manager Catherine Beadle (Figure 82) and were made as close as possible to the constraints present on the roster. The staff requests for the period 1st March to 31st Match 2004 were also entered. Results of the algorithm on the first 2 weeks of March are shown in Figure 83.

It took the computer longer to generate a good roster for Harold ward due to low staffing and many constraints. However, the time taken to generate each week was **below 5 minutes**. Determining the constraints and weightings needed took several hours, but once these were determined, they did not need to be changed for the whole period. A few re-tries were needed to get a good assignment for agency staff, due to the sub-optimal nature of the algorithms used, however this only took a few extra minutes. The roster automatically generated by this project is shown in Figure 83.

Overall all that were questioned on Harold ward were generally pleased and impressed by the generated roster. Some of the constraints present on this roster were difficult or impossible to represent programmatically, especially those regarding intricacies of particular staff member’s preferences. It would appear that there is an inherent semantic gap between human perception of roster fitness and simple mathematical definitions of utility functions as used in this project.

As not every constraint could be exactly represented, a number of swaps and alterations were suggested, but these were easily carried out by hand through the user interface of the project. Although the software was unable to completely solve the problem autonomously, it was regarded by those questioned as an effective tool in aiding the process. The user interface also made adjusting the roster significantly easier than on paper. Some of the roster could also be assigned by hand through the user interface, leaving the algorithm to solve the remainder of the roster. This was also found to be a fast and effective way of producing a good roster.

Catherine Beadle was quoted as saying **“In 2 minutes it’s produced a better roster than you often get after spending 2 hours with a pencil and paper”**.
<table>
<thead>
<tr>
<th>Staff Member</th>
<th>Grade</th>
<th>01-Mar</th>
<th>02-Mar</th>
<th>03-Mar</th>
<th>04-Mar</th>
<th>05-Mar</th>
<th>06-Mar</th>
<th>07-Mar</th>
<th>08-Mar</th>
<th>09-Mar</th>
<th>10-Mar</th>
<th>11-Mar</th>
<th>12-Mar</th>
<th>13-Mar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle</td>
<td></td>
<td>E</td>
<td>E</td>
<td></td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farrow</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloomfield</td>
<td></td>
<td>N</td>
<td>N</td>
<td>Do</td>
<td>R</td>
<td>Ld</td>
<td>Ld</td>
<td>Ld</td>
<td>Do</td>
<td>Ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hall</td>
<td></td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>R</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>Sunner</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ld</td>
</tr>
<tr>
<td>Geddis</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>Lewis</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>Miller</td>
<td></td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wiseman</td>
<td></td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Do</td>
</tr>
<tr>
<td>Castilho</td>
<td></td>
<td>D</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>Cells</td>
<td></td>
<td>D</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>Mathew</td>
<td></td>
<td>D</td>
<td>R N</td>
<td>N</td>
<td>R N</td>
<td>R N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mayenbe</td>
<td></td>
<td>D</td>
<td>N</td>
<td>L</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Ojo</td>
<td></td>
<td>D</td>
<td>Ld</td>
<td>Ld</td>
<td>Ld</td>
<td>Ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reamboanza</td>
<td></td>
<td>D</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simmons</td>
<td></td>
<td>D</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>Do</td>
<td>Do</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>HCA</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>AL</td>
<td>AL</td>
<td>R</td>
<td>R</td>
<td>AL</td>
<td>AL</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Alison</td>
<td>HCA</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>AL</td>
<td>AL</td>
<td>R</td>
<td>R</td>
<td>AL</td>
<td>AL</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Carol C</td>
<td>HCA</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Carol D</td>
<td>HCA</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Caroline</td>
<td>HCA</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Chantelle</td>
<td>HCA</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Chris</td>
<td>HCA</td>
<td>N</td>
<td>L</td>
<td>L</td>
<td>Do</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>N</td>
<td>Ld</td>
<td></td>
</tr>
<tr>
<td>Donna</td>
<td>HCA</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacqui</td>
<td>HCA</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katie</td>
<td>HCA</td>
<td>E</td>
<td>L</td>
<td>OL</td>
<td>OL</td>
<td>OL</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lubke</td>
<td>HCA</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>Do</td>
<td>Do</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>Do</td>
</tr>
<tr>
<td>Rachel</td>
<td>HCA</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>Do</td>
<td>Do</td>
<td>L</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>Do</td>
</tr>
<tr>
<td>Ruth</td>
<td>HCA</td>
<td>N</td>
<td>Ld</td>
<td>L</td>
<td>E</td>
<td>L</td>
<td>Ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treena</td>
<td>HCA</td>
<td>OL</td>
<td>OL</td>
<td>OL</td>
<td>OL</td>
<td>OL</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank1</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank2</td>
<td>E</td>
<td>Ld</td>
<td>Ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank3</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>N</td>
<td>E</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank4</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank5</td>
<td>E</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 83 Roster for Harold Ward generated by this project
9.2 Comparison of project and previous papers

9.2.1 Overview

It was decided that a good test of the project would be to compare it’s performance to that of the algorithms mentioned in previous papers on the subject.

As has been illustrated in section 9.1, the algorithm implemented in this project has no difficulty generating feasible rosters, with many constraints for 30 or more staff members. This is in line with previous papers on the subject, which also have achieved similar results. Therefore, it was decided that the main criteria for comparison should be time taken to create a roster.

For a fair comparison, the agency staff assignment will be ignored as other papers have not performed a thorough treatment of this. Therefore the project will be compared in situations where there are adequate staffing levels to find a feasible solution.

9.2.2 Project performance

Figure 84 illustrates a run of the genetic algorithm for a standard rostering problem for a ward with 30 nurses. Score (section 7.5) is a weighted average of constraint violations.

![Figure 84 Performance on a 30 staff member roster](image-url)
As can be seen, the genetic algorithm yields no significant improvement after about 40,000 ms or 40 seconds.

The algorithm does not reach its termination condition however (300 generations without improvement) until around 100,000 ms or 1 minute and 40 seconds. A feasible solution (i.e. no hard violations) is reached after 23 seconds.

At termination, the algorithm had considered slightly more than 728 generations with 1 thread.

These results were recorded on a single processor Pentium 4 1.7Ghz machine, using Sun's JRE version 1.4.1.

9.2.3 Performance as quoted by Grobner and Wilke

In their 2001 paper entitled “Rostering with a Hybrid Genetic algorithm”[2], Dr Matthias Grobner and Dr Peter Wilke attempted to create a program to solve the nurse duty rostering problem using a completely different genetic algorithm to the one used here. In the paper, the algorithm is described as using “fix operators” which encapsulate domain specific knowledge on the problem and “fix” problems in the rosters. These were required because their genetic algorithm used a totally unrestrained genetic representation which would create rosters with, for example, more than one shift on a day for a single person.

Figure 85, shows a graph of a sample run of Grobner and Wilke’s program, showing how violations varied with successive generations. The algorithm was run on a roster for approximately 30 staff members, making it comparable to the run shown in the previous section.

Figure 85 Grobner and Wilke's Algorithm performance

The first notable difference between this graph, and the comparable graph shown for my algorithm (Figure 84) is the number of hard violations. The unconstrained genetic representation used here leads to many more hard violations than are possible with this project’s more constrained genetic representation.
The second most notable difference is the number of generations before no more change occurred in the solution. In this project, this was around 800 generations, as previously quoted, we can see Grobner and Wilke’s program ran to around 40,000 generations, before a feasible solution was found.

This run is quoted to have taken “10 minutes on a Pentium II 600MHz machine”. Therefore, most notably, Grobner and Wilke’s program goes through generations far faster than the algorithm used in this project, yet requires many more generations to reach a solution.

9.2.4 Conclusions

A direct comparison of speed between this project and Wilke’s will be difficult due to the differing test conditions and differences between the programs.

If the computer used for testing this project is assumed to be around 8-10 times faster than the one Grobner and Wilke used (a large overestimate), then we can see that on a comparable machine these two programs would run in about the same time.

This is interesting as this project is written using Java, which running on a VM should make less efficient use of the CPU than the implementation provided by Grobner and Wilke. This may imply the algorithm used in this project is fundamentally faster at reaching a solution, but test conditions are too different to draw any concrete conclusions.

We may say that the algorithm used in this project takes a reasonable period to produce a roster, if Grobner and Wilke’s 10 minutes for a solution is used as a benchmark.
9.3 Conclusion

9.3.1 Overview
Overall the feedback on the rosters generated by this project was very pleasing and encouraging, several of the nurses who helped in testing expressed interest in the system, the project and the commercial viability of the project.

This project highlights the inherent difficulty in representing human perception of roster fitness in terms of a multi-attribute utility function [31,32]. The rosters generated using this approach are not likely to be perfect, however, it is often simpler for the user to enforce more complex and enigmatic constraints by assigning shifts manually through the user interface.

The nurses on Harold Ward perceived the automatic generation of rosters as more of a time saving aid than a complete solution, which given the nature of their problems is reasonable. The number of constraints and complexity of some requests means that a human element will always be necessary in the rostering process, and is in fact desirable.

Imperfections can be manually corrected through the user interface easily and quickly. In human generated rosters such fine tuning often occurs between staff members on wards, with nurses swapping shifts between themselves.

The project also illustrates the effectiveness of genetic algorithms at solving the rostering problem, when augmented with domain specific knowledge. It also illustrates that it is not essential to use “fix operators” [2] or complex decoders [3] to encapsulate domain specific knowledge.

9.3.2 Effectiveness of user interface
The main aim of the interface was to provide a user-friendly experience which would make the common case quick and easy. In the project, the common case was the generation of rosters, as opposed to the specification of constraints or setting up the roster, as this must be done only once per ward, with infrequent adjustments.

My mimicking the interface of popular applications such as Excel, the project produced an interface instantly understandable to the average nurse, familiar with a nursing roster. Furthermore, the addition of various views on the roster data was very much welcomed by nursing staff. Some staff members were almost as impressed by the multiple views as by the automatic generation of rosters, this alone represents a significant timesaving in the rostering process.

Detailed consideration was given to producing a user interface for inputting constraints which was both user friendly and contained the necessary functionality for cross ward usage. User consultation indicated that the interface was functional, and understandable, once acquainted with. Since this interface is seldom used in daily operations, this seemed satisfactory for the scope of the project.
9.3.3 Use of multiple threads
Alone, the run time of the multi-threaded algorithm without “thread culling”, indicated that the use of multiple threads was not any better than running one thread multiple times. This was a disappointing result. However, with thread culling, better solutions were produced, although the increase in fitness was only marginal, this marginal increase probably represents the fulfilment of soft constraints such as staff requests. Therefore it was decided to accept an increase in run time for a small, yet significant increase in roster fitness.

9.3.4 Use of hill climb
The hill climbing algorithms seemed effective in operating in the simple case of efficiently assigning agency staff. While they could not be used for completely solving the problem, they executed rapidly. The hill climbers varied in the quality of solution they produced, being susceptible to becoming stuck in local minima. This meant that on difficult rosters, several attempts were needed to find a good solution.

9.3.5 Effectiveness of twin removal
The effectiveness of twin removal in improving the solutions surpassed expectations. It was expected to yield some small differences, after observations from Konstantin Boukreev’s freeware travelling salesman program [9] revealed that it’s inclusion improved the solutions found. Across a wide range of rosters, twin removal increased resistance to getting stuck in local minima with negligible increase in run time.

9.4 Possible extensions
There are a number of possible extensions to this project which, given more time and resources would be interesting to implement.

- Full length user acceptance test of the software, on a number of wards, across a number of health authorities.
- Direct comparison of performance with other algorithms, such as Tabu search [18], by adding extra classes into the application.
- Adding the ability for users to store rosters on a central server
- Add a “manager” interface to allow analysis and comparisons of rosters between wards.
- A semi-natural language processing system for specifying constraints would be an interesting piece of research work, and would perhaps help overcome some of the limitations with the project’s representation of constraints, particularly complex constraints.
- Allowing the user to specify complex and arbitrary orderings upon constraints and using a more sophisticated method of ranking to evaluate rosters would also be an interesting piece of research. Such extensions would more accurately represent the human perception of roster fitness.
- Replace the hill climbing algorithms with a different approach such as simulated annealing [28] or Tabu search [18].
- Add “long term fairness” constraints to the fitness function.
10 References


Second Asia-Pacific Conference on Simulated Evolution and Learning (SEAL ’98), Canberra


[41] JOHNSON, A W. JACOBSON, SH. 1996 Generalized hill-climbing algorithms for discrete optimization problems

11 Appendices

11.1 Appendix 1: Experimental conditions for mutation testing

Test conditions were as follows:

- Genetic algorithm run for 300 generations, with no extra features such as hill climbers. This was to ensure that the only factor being tested was the effectiveness of the pure genetic algorithm.
- Three threads were used.
- A hypothetical ward with 15 staff members: 7 trained, 8 untrained, all full time.
- A request for each of the 15 staff members, spread across the week and kept constant.
- Four possible shifts that could be worked:
  - Early Shift (7.5 hours)
  - Late Shift (7.5 hours)
  - Night Shift (7.5 hours)
  - Long Day shift (15 hours)
- Long day shift can only be worked by a single trained staff member. All other shifts may be worked by all other staff members.
- On a day shift 4 staff members are required: 2 Trained, 2 Untrained
- On a night shift 2 staff members are required: 1 Trained, 1 Untrained
- The usual set of general constraints:
  - Poor assignment soft constraint: Avoid scheduling the same shift for a staff member all week.
  - Clustering constraint: Days off should be clustered, Early and Late shifts should not exceed a run of 3 consecutive shifts.
- A request not to work night duties for one, trained staff member.

The weightings for these constraints were as follows:

<table>
<thead>
<tr>
<th>Hard Constraints</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect number of staff members a shift</td>
<td>25.0</td>
</tr>
<tr>
<td>Illegal consecutive shifts</td>
<td>50.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Soft Constraints</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>All day off requests</td>
<td>6.25</td>
</tr>
<tr>
<td>Scheduling an Early Shift after an Late Shift</td>
<td>2.222222222222223</td>
</tr>
<tr>
<td>Penalises for unequal distribution of shifts</td>
<td>0.8888888888888888</td>
</tr>
<tr>
<td>Prevent scheduling any one shift too many times for any member</td>
<td>2.6666666666666665</td>
</tr>
<tr>
<td>Ensures that blocks shifts are grouped together correctly</td>
<td>6.222222222222221</td>
</tr>
</tbody>
</table>

MAX_POPULATION=300; ELITE_PROPORTION = 0.15; MUTATION_PROB = 0.23;
11.2 Appendix 2: Test Conditions for thread testing

For testing, it was decided to use a standard laptop PC, equipped with a single Intel Pentium-4 processor (with speed-step) and 256Mb of RAM. For the purposes of the test, the speed-step technology was disabled on the test machine, so it always ran at a clock speed of 1.7Ghz. The operating system was Windows XP Professional and the Java VM used was Sun Microsystems Java VM version 1.4.1. During test runs, the VM was the only user process running.

The roster used was a simple test roster, with 15 staff and all the types of constraint present. The exact details are fairly irrelevant, however, as similar results can be obtained using any test roster. It should be noted that the test roster was never changed or altered in any way.

The worker threads were instructed to run the genetic algorithm to exactly 200 generations and terminate. Once all threads had terminated the time for the whole run was saved to a log file.

All experiments were repeated 15 times to avoid variations in CPU utilisation from system processes. The variance and mean of the recorded times were computed, and from this confidence intervals for the “true population mean” were computed.
11.3 Appendix 3: Hard Constraints Implementation

11.3.1 Class StaffLevelsConstraint

When the score method of this class is called, the class generates a fixed penalty score for each violation of the following constraints (outlined in section 5.6.2.2):

- Always have at least a certain number of trained staff on any one shift,
- Always keep staffing levels above a certain minimum defined for each shift,

This is achieved by passing over the roster, and counting the number of assignments for each shift on each day, and the number of trained staff assigned to each shift on each day. These figures can be compared with the predefined minimum and maximum values associated with the shift (section 7.8).

For example:

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>Grade (Skill level)</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle G</td>
<td>Night</td>
<td>Early</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farrow G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloomfield F</td>
<td>Night</td>
<td>Night</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hall F</td>
<td>Late</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sunner F</td>
<td></td>
<td>Early</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geddis E</td>
<td></td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Late</td>
</tr>
</tbody>
</table>

Given the above roster, the class would generate the following table by iterating over the cells in the roster:

<table>
<thead>
<tr>
<th></th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of staff on Night</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of staff on Late</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Number of staff on Early</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Number of trained staff on Night</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of trained staff on Late</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Number of trained staff on Early</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

The figures in this table can be compared with the minimum and maximum staffing levels for each shift.

In the majority of cases, the number of staff required on any shift is the same, regardless of the day of the week. In some special circumstances, staffing levels may differ on certain days of the week. For example it may be required that 3 staff members work early shift, except on Wednesdays when only 2 are required.

Therefore this class has a hash table, indexed by shift and containing the normal maximum and minimum staffing levels. It has a second hash table, indexed by day and by shift containing any special cases, which differ from the normal maximum and minimum staffing levels.
Example hash maps:

**General Case hash map:**

<table>
<thead>
<tr>
<th>Key</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Late Shift</td>
<td></td>
</tr>
<tr>
<td>Night Shift</td>
<td></td>
</tr>
</tbody>
</table>

*Key Details*
- Late Shift Minimum level = 4
- Minimum trained staff = 2
- Maximum staff = 6
- Night Shift Minimum level = 2
- Minimum trained staff = 2
- Maximum staff = 6

**Special case hash map:**

<table>
<thead>
<tr>
<th>Key</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Late Shift, Monday)</td>
<td></td>
</tr>
</tbody>
</table>

*Key Details*
- Minimum level = 3
- Minimum trained staff = 1
- Maximum staff = 5

This design was used because the number of special cases is low. Therefore the special case hash map is hardly referenced. This seemed like a more efficient method of storage than storing all the staffing levels in an array, or storing all the staffing levels in a hash map indexed by day and shift. For all days which are not special cases, it uses the general case hash map.

It was later realised that one may need to specify a maximum number of untrained staff. This was implemented the same way as the other staffing levels.

### 11.3.2 Class ConsecutiveConstraint

This generates a fixed penalty score for each violation of the constraint outlined in section 5.6.3.4.

Every shift has a set of shifts which are illegal if placed before it and a set of shifts which are illegal if placed after it (section 7.8). The roster is traversed and each time one of the prohibited shifts is seen, a fixed penalty is generated.

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>Grade</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle</td>
<td>G</td>
<td>Night</td>
<td>Early</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farrow</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloomfield</td>
<td>F</td>
<td>Night</td>
<td>Night</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hall</td>
<td>F</td>
<td>Late</td>
<td></td>
<td>Late</td>
<td></td>
</tr>
<tr>
<td>Sunner</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>Early</td>
</tr>
<tr>
<td>Geddis</td>
<td>E</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Late</td>
</tr>
</tbody>
</table>

In the above roster, the class would start by examining the cell “01/03/04” for Beadle. When it got to “02/03/04” for Beadle, it would realise that a Night duty cannot precede an Early shift, as Night shift will be in the set of shifts illegal before an Early shift. Therefore a penalty score will be generated.

### 11.3.3 Class SkillsHardConstraint

As mentioned in section 7.7, staff members are tagged with their skills.
The class passes across a roster. For each day it counts the skills possessed by the staff working on each shift. These figures are compared to the minimum levels required for each skill, which are held in the shift classes (section 7.8). If the figures fall below the specified minimum level, a penalty score is generated.

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>Skills</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle</td>
<td>RSCN</td>
<td>Night</td>
<td>Early</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farrow</td>
<td>RSCN</td>
<td></td>
<td></td>
<td>Early</td>
<td></td>
</tr>
<tr>
<td>Bloomfield</td>
<td>RSCN</td>
<td>Night</td>
<td>Night</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hall</td>
<td>RSCN</td>
<td>Late</td>
<td></td>
<td>Late</td>
<td></td>
</tr>
<tr>
<td>Sunner</td>
<td>RSCN</td>
<td></td>
<td></td>
<td>Early</td>
<td></td>
</tr>
<tr>
<td>Geddis</td>
<td>RSCN</td>
<td>Early</td>
<td>Early</td>
<td>Late</td>
<td>Late</td>
</tr>
</tbody>
</table>

Given the above roster, the class would pass across the roster and generate the below table:

<table>
<thead>
<tr>
<th>RSCN staff present on</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Late</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Night</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This table would then be used to check whether there are enough RSCN staff members on each shift.
11.4 Appendix 4: Soft Constraints Implementation

11.4.1 Requested days / shifts (Class ShiftRequested)

One of these classes is created for all requests of the form:
“Schedule / Do not schedule Staff Member A for shift X on day Y”

The requests can be either positive (please schedule me a shift) or negative (schedule anything except this shift).

Every time a violation of the constraint encapsulated by this class is seen, a fixed penalty score is generated.

11.4.2 Requested shifts types (Class ShiftStaffRequested)

An object of this type is created for each request for a type of shift (see ShiftType class in section 7.8). The requests can be either positive (please schedule me a day shift) or negative (schedule anything except a day shift), and can be for any shift type in the system. For example we could express:
“Please schedule staff member A for a day shift on day Y”
“Please schedule staff member A for a night shift on day Y”

Every time a violation of the constraint encapsulated by this class is seen, a fixed penalty score is generated.

11.4.3 Shift preference (Class ShiftPref)

An object of this type is created for each request of the following form:
Schedule / Do not schedule staff member A for shift B throughout the week

Every time a violation of the constraint encapsulated by this class is seen, a fixed penalty score is generated.

11.4.4 Shift Type preference (Class ShiftTypePref)

One of these classes is created for each request of the following form:
Schedule / Do not schedule staff member A for shift B type throughout the week

Every time a violation of the constraint encapsulated by this class is seen, a fixed penalty score is generated.

11.4.5 Shift staffing (Class ShiftDistributionSoft)

This class attempts to generate a fitness for the roster with respect to the constraint outlined in section 5.6.3.5 of this report.

If we try to find the variance of the staffing level of a specific shift over the week (the variance of the number of staff working that shift each day), minimising this variance will lead to better spread of staff.

This is because by narrowing the variance of staffing level, we ensure that the staffing level of that shift over the whole week is closer to the mean staffing level. The ideal
situation would be zero standard deviation, so the same number of people were working the shift on each day.

The problem with this approach is that it neglects the mean, thus having a low mean staffing level for a shift throughout the week would actually be ranked better than an arrangement with good staffing, but high variance. Ideally, mean should be maximised as well.

To encapsulate the fact that both minimising the variance and maximising the mean are desirable, a “score” for this constraint is calculated as follows:

\[
s = \frac{1}{N} \sum_i \sigma_i^2 + \frac{1}{1 + (\frac{\mu - \mu_s}{m})}
\]

Where:
\( \mu_i \) = The mean staffing level of shift "i" in the roster
\( \sigma_i^2 \) = The variance of the staffing level of shift "i" in the roster
\( m_i \) = The minimum required staffing level of shift "i"
\( \mu_s \) = The "normalised mean staffing level" of shift "i" in the roster
\( N \) = The number of different shifts in the roster, which have a minimum staffing requirement.
\( s \) = “Score”

This score will decrease if the variance is decreased or if the mean is decreased. Furthermore, if the normalised mean staffing level of a shift drops to zero, we are not left with an undefined result, due to the “1+” term.

The score is passed straight to the genetic algorithm and used in the fitness function.

11.4.6 Consecutive shifts soft constraint (Class ConsecutiveShiftsSoft)

This class checks for violations of the constraint described in section 5.6.3.4. The constructor of this class is given 2 shifts, call these S1 and S2. The class penalises each time S1 is seen the day before S2.

11.4.7 Poor assignment soft constraint (Class PoorAssignmentSoft)

This class counts the number of hours worked on each shift for each staff member. If the number of hours worked on any one shift in a week exceeds 75% of the total time worked, then a penalty is generated.

This penalises the algorithm for assigning the same shift in a row many times, which generally frowned upon.

11.4.8 Clustering penalties (Class ClusteringSoft)

This class checks for violations of the constraint outline in section 5.6.3.3.
The class is passed a set of “clustering instructions”. Each “clustering instruction” describes a shift (or a day off), the maximum cluster size and the minimum cluster size. If either the max or min size is not appropriate for a specific shift, these values can be set to -1 to indicate that they should not be checked.

The class runs across the roster, counting the number of identical consecutive shifts for each staff member. A penalty score of 1 is generated for every shift assignment which exceeds the specified maximum. A penalty score of 1 is also generated for every shift that the roster is below the specified minimum.

**Example roster:**

<table>
<thead>
<tr>
<th>Staff Member</th>
<th>Skills</th>
<th>01/03/2004</th>
<th>02/03/2004</th>
<th>03/03/2004</th>
<th>04/03/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beadle</td>
<td>RSCN</td>
<td>Night</td>
<td>Night</td>
<td>Night</td>
<td>Night</td>
</tr>
<tr>
<td>Farrow</td>
<td>RSCN</td>
<td>Early</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloomfield</td>
<td>RSCN</td>
<td>Night</td>
<td>Night</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sunner</td>
<td>RSCN</td>
<td>Night</td>
<td>Early</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example clustering instruction:**

<table>
<thead>
<tr>
<th>Shift</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Night</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Given the above data, this class would traverse the given roster from left to right, top to bottom. On “4/3/04” for “Beadle”, the class would detect that 4 night shifts had occurred in a row and would generate a penalty score of 1. On “3/3/04” for “Bloomfield”, no penalty score would be generated because 2 night duties in a row is within acceptable limits.

On “2/3/04” for “Sunner”, the class would detect that 1 night shift had occurred. This would cause the class to generate a penalty score of 1.

**11.4.9 Weekend considerations (NoScheduleWeekend and InvalidWeekendCombination)**

As outlined in 5.6.3.6, there are 2 constraints relating to the weekend:

1. Do not schedule undesirable shift combinations at the weekend
2. Try not to schedule many shifts at the weekend for any staff member.

To encapsulate these two constraints, there are two classes. Both of these derive from the same abstract superclass “WeekendShiftSoft”, reducing replicated code. These classes must be informed which days in the roster (section 7.3) represent the weekend. For example it might be day number 5 and day number 6 which represent Saturday and Sunday in the algorithm’s representation of a roster.

The NoScheduleWeekend class generates a fixed penalty for each person it sees with more than 2 shifts scheduled at the weekend.

The class “InvalidWeekendCombination” holds two shifts. If both of these shifts are seen at the weekend for any staff then a fixed penalty is generated for each staff member with these shifts scheduled.