Reducing The Size Of Driver Scheduling Problems Using Constraint Programming

J.A.Gaw

B.Sc. Computing

Session 2002/2003

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)______________________________
Summary

A review of the driver scheduling problem, and the approaches used to solve it, is presented. A means of improving the computational efficiency is proposed and an algorithm implemented to investigate the feasibility of the proposed approach.

The algorithm is tested using a sample problem, and the results discussed.
Acknowledgements

That I managed to get to this point at all is largely thanks to three people: Tracy Lee, Andy Smith, and Sarah Fores. As Victor Mollo [17] might have said, “They share the blame equally, but Tracy’s share is the most equal of the three”.

Tracy first suggested the possibility of my giving up the day job and studying for a degree in computing. She put up with the late nights, lost weekends, stress and hassle. She gave me all the time and space I needed, and provided unflinching support and encouragement throughout, for which I will be eternally grateful (under pain of death).

Andy evidently thought that exposing the love of my life to all this grief was a good idea. As my former boss, he only had to refuse my application for voluntary redundancy, and none of this would have happened. But he let me go, and to this day continues to encourage my efforts. And that’s got to be worth a few beers.

Finally, I would like to thank Sarah, who had to suffer the consequences of the above recklessness, first as my tutor, and latterly as my project supervisor. Always willing to discuss ideas, offer suggestions, and provide a sympathetic ear, she somehow managed to keep me heading in approximately the right direction, and that is no mean feat.
## Contents

1 Background Research  
1.1 The Driver Scheduling Problem  
  1.1.1 Overview  
  1.1.2 A Sample Problem and A Manual Solution  
1.2 Basic Solving Methods  
  1.2.1 Set Covering vs. Set Partition  
  1.2.2 Integer Linear Programming  
  1.2.3 Constraint Programming  
  1.2.4 Local Search Methods  
    1.2.4.1 Simulated Annealing  
    1.2.4.2 Tabu Search  
1.3 Improvements  
  1.3.1 Overview  
  1.3.2 Column Generation  
  1.3.3 Relief Point Selection  
1.4 TRACS II  

2 Project Management  
2.1 Overall Aims  
2.2 Minimum Requirements and Deliverables  
  2.2.1 Present a Brief Review of Approaches to Driver Scheduling Problems  
  2.2.2 Present a Review of Approaches Used in the School of Computing  
  2.2.3 Implement Small Sample Problem Using Ilog-5.0  
  2.2.4 Implement Small System to Attempt to Reduce the Size of the Shift Subset  
  2.2.5 Present An Evaluation of the New Algorithm  
2.3 Extensions And Enhancements  
2.4 Methodology  
2.5 Project Schedule
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1 The Original Project Schedule</td>
<td>13</td>
</tr>
<tr>
<td>2.5.2 The Original Schedule</td>
<td>14</td>
</tr>
<tr>
<td>2.5.3 The Revised Schedule</td>
<td>15</td>
</tr>
<tr>
<td>2.5.4 Progress vs. the Schedule</td>
<td>15</td>
</tr>
<tr>
<td>3 Algorithm Design</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Data Structures</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Constraints</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1 Legal Spells of Work</td>
<td>17</td>
</tr>
<tr>
<td>3.3.2 Legal Shifts</td>
<td>19</td>
</tr>
<tr>
<td>3.3.3 Work Constraints</td>
<td>19</td>
</tr>
<tr>
<td>3.4 The Objective Function</td>
<td>19</td>
</tr>
<tr>
<td>3.5 The Search Process</td>
<td>20</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Familiarisation With Ilog</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1 Overview</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Ilog Demonstration Code</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1 Demo01-Demo03</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2 Demo04</td>
<td>22</td>
</tr>
<tr>
<td>4.2.3 Demo05-Demo06</td>
<td>23</td>
</tr>
<tr>
<td>4.2.4 Demo07-Demo10</td>
<td>23</td>
</tr>
<tr>
<td>4.3 The First Prototype</td>
<td>23</td>
</tr>
<tr>
<td>4.3.1 Design</td>
<td>23</td>
</tr>
<tr>
<td>4.3.2 Data Structures</td>
<td>24</td>
</tr>
<tr>
<td>4.3.3 Weights</td>
<td>24</td>
</tr>
<tr>
<td>4.3.4 Decision Variables and Constraints</td>
<td>24</td>
</tr>
<tr>
<td>4.3.5 Objective Function</td>
<td>25</td>
</tr>
<tr>
<td>4.3.6 Functional Testing</td>
<td>25</td>
</tr>
<tr>
<td>4.3.7 A Larger Data Set</td>
<td>26</td>
</tr>
<tr>
<td>4.3.8 Evaluation</td>
<td>26</td>
</tr>
<tr>
<td>4.4 The Second Prototype</td>
<td>27</td>
</tr>
<tr>
<td>4.4.1 Design Changes</td>
<td>27</td>
</tr>
<tr>
<td>4.4.2 Generating Complete Shifts</td>
<td>27</td>
</tr>
<tr>
<td>4.4.3 Improving Computational Efficiency</td>
<td>28</td>
</tr>
<tr>
<td>4.4.4 Functional Testing</td>
<td>28</td>
</tr>
<tr>
<td>4.4.5 Calculation of Weights</td>
<td>29</td>
</tr>
<tr>
<td>4.4.6 Evaluation</td>
<td>30</td>
</tr>
</tbody>
</table>
Chapter 1

Background Research

1.1 The Driver Scheduling Problem

1.1.1 Overview

Driver scheduling is one part of the process of designing transport schedules. Transport scheduling is typically broken down into three phases, each tackled separately.

First, a vehicle schedule is drawn up. For bus and train services, a schedule typically covers a period of one working day. The schedule for a single vehicle is referred to as a board, and a vehicle schedule may consist of many boards. Each board is divided into pieces of work, at times known as relief opportunities, because they represent the (only) times at which one driver may be relieved by another. These occur when driver changes can be made conveniently, for example when the vehicle returns to a depot or terminus. The vehicle schedules dealt with in driver scheduling usually cover a single days operations.

Figure 1.1 shows an example, the GMB schedule. This consists of 104 pieces of work, using 12 vehicles. The relief opportunities are marked with the times at which they are scheduled. The relief point locations, normally also marked, are omitted for clarity, since a single relief point location is used in this set. Much of the route data required for vehicle scheduling is omitted. The only data relevant to the problem are the points and times at which a driver may be relieved by another.

The second phase consists of devising a set of driver shifts which cover all pieces of work in the schedule, whilst minimising some cost function. Typically the cost function is dominated by the number of drivers required. In addition, the shifts must meet additional constraints, such as limits on the length of a shift, adequate provision for meal breaks etc. These are generally
imposed by labour agreements and/or legislation. This is the stage with which this project is concerned.

The final stage consists of devising driver rotas such that a driver is assigned to each shift, and each driver works a pattern of shifts which meet contractual and legislative requirements. Driver rotas typically cover periods of several weeks.

It should be noted that some approaches, e.g. Haase et al. [10], and Valouxis and Housos [2], combine driver and vehicle scheduling in a single step.

### 1.1.2 A Sample Problem and A Manual Solution

As a familiarisation exercise, the Greater Manchester Buses (GMB) problem was examined. The GMB schedule consists of 12 boards, containing a total of 104 pieces of work depicted in figure 1.1. The labour agreement specifies a minimum spell duration of 2 hours, and a maximum spell duration of 5 hours. The maximum shift length is 9 hours 45 minutes including a meal break (known as a Personal Needs Break, or PNB) of at least 1 hour, or 12 hours, including a meal break of at least 3 hours (known as a split shift). Split shifts are generally avoided where possible because they incur extra costs. The problem is to design a set of driver shifts which cover all the pieces of work whilst minimising some cost function. The cost function for this example is simply the total number of drivers used. Certain undesirable features, such as split
shifts, and shifts which contained more than two spells, were avoided where possible.

It should be noted that this problem is greatly simplified in comparison with real-world problems, which have many additional restrictions, e.g. relief points are not all at the same physical location; travelling between them takes time, and in some cases may be completely impractical. There may be constraints on the number of drivers based at certain locations. Drivers may be restricted in the types of vehicles (or routes) which they may drive. Even with these simplifications of the true problem, it is still not a simple task to draw up an efficient schedule. In fact, the driver scheduling problem is known to be NP-hard, e.g. Fores et al. [8], meaning that optimal solutions cannot be guaranteed to be obtained in polynomial time. For non-trivial problems, this means that brute-force approaches cannot be used effectively, and other methods must be applied.

In attempting a solution, several points were noted. The total number of working hours is 120. With 8:45 working hours available on a regular shift, or 9:00 hours on a split shift, a solution with fewer than 14 shifts is impossible. There are two periods of peak demand, between 08:10 and 09:00, and between 15:15 and 18:15, each with 10 routes in service. Because the timespan between 08:10 and 18:15 is greater than the 09:45 hours maximum for a regular shift, a driver cannot cover both peak periods on a regular shift, so without split shifts, a minimum of 20 drivers will be needed. This strongly suggests that split shifts will be a necessity in an optimal solution, despite their higher cost. Ideally, PNBs should be staggered, so that a minimum number of drivers are on a PNB at any given moment. This was subsequently found to be both well known, e.g. Wren [24], and exploited in some approaches to the problem e.g Layfield et al. [16].

The best solution produced used 16 shifts, matching the result obtained solving the problem using TRACS II, though the shifts produced were different.

The approach taken was to target the hardest parts of the schedule first, such as pieces with the least flexibility. Shifts were constructed, as far as possible, to cover times of peak demand or other pieces deemed difficult to deal with, and covered as much work as possible. For example the first pieces assigned to shifts were the late evening pieces on boards 5, 6 and 9. It was clear that assigning shifts became harder as the task progressed and the number of possible combinations was reduced. This is the stage where the advantages of an automated system, capable of checking many combinations, is most apparent.

The work raised the question of whether there is a performance benefit to be gained from a preprocessing step which constructs shifts to cover such pieces of work, leaving a reduced size problem that can be processed by other methods.
1.2 Basic Solving Methods

Wren [24] states that computer-aided scheduling dates back to the 1960s, and that early research focused on heuristics rather than mathematical programming, as available hardware was incapable of solving real-world problems expressed as ILPs. The heuristics relied heavily on specific features of the problems they were intended to solve and were not easily generalised to other problems. He notes that since the late 1970s, the focus of research has been on hybrid systems which combine mathematical programming and heuristics.

TRACS II, a driver scheduling system developed at the University of Leeds, takes this approach.

1.2.1 Set Covering vs. Set Partition

Assigning pieces of work to shifts can be treated as a set partitioning or a set covering problem. In a set partitioning problem, all pieces must be covered exactly once - partitioning the set amongst a number of shifts. In a set covering problem, the position is relaxed slightly, in that each piece must be covered at least once. A piece which is covered by more than one shift is said to be overcovered. Obviously this is not desirable as it is inherently inefficient, but it is more flexible. In practise, the set covering approach is more common, though set partitioning has been used in some cases, e.g. Curtis et al. [1], Muller [18].

1.2.2 Integer Linear Programming

This approach is the basis for several commercial systems, including TRACS II, and details are given in many papers, e.g Fores et al. [5, 6, 8]. It takes a set of pieces of work, a set of separately generated shifts, and a set of constraints derived from the problem, and looks for a solution as a subset of the shifts which covers the entire schedule whilst minimising a cost function. The constraints can be split into those which ensure a schedule is complete (i.e. every piece of work is covered) and side constraints which may restrict, for example, the types or numbers of shifts used in a solution, numbers of drivers operating from particular locations, times allowed for drivers to move between relief points, etc.

The cost function can be as simple as the total number of shifts used, or very complex, including factors which take account of lengths of shifts, undesirable (but acceptable) features such as very long shifts or shifts with long idle times, overtime, etc. A typical cost function would include very heavy penalties for pieces of work which were left uncovered, penalties for overcover on pieces of work (i.e. two or more selected shifts covering one piece), and would be strongly dependent on the total number of drivers used.

A solution is found using branch and bound technique, a standard method for problems of this type, which according to Winston [23] “find the optimal solution to an IP by efficiently enumerating the points in a subproblem’s feasible region”.

4
1.2.3 Constraint Programming

Constraint programming is a technique which defines a problem in terms of variables, domains on those variables, and constraints which restrict relationships between variables. A solution consists of finding values for all variables which satisfy all the constraints.

A pure CP model would have to represent all possible shifts and all pieces of work, and specify constraints to represent legal shifts and other considerations, and suffer from the NP-hardness of the problem - an impossibly large solution space to be searched. Curtis et al. [1] note that there are two ways of representing the driver scheduling problem, using either pieces of work or shifts as variables, and that the choice is critical to performance. They describe an implementation of a CP model, employing knowledge of the structure of the problem to improve performance.

1.2.4 Local Search Methods

Local search methods have 3 key components: an initial solution, which may be good, bad, or even infeasible; a method of evaluating a solution; and a means of modifying one solution to obtain another, called a move. In the context of a driver scheduling problem a move may mean adding or removing a piece of work. The basic method involves generating a number of neighbouring solutions from the initial solution - this set of candidate solutions is known as a neighbourhood. The new solutions are evaluated, and the highest scoring candidate, if it is better than the current best solution, is accepted as the new best solution. This process repeats until either no neighbouring solution offers any improvement, or some limit on time or iterations is reached, at which point the best solution found is returned.

The initial solution can be obtained using any means available, but in general, the better the initial solution, the more effective the search.

The evaluation step is obviously crucial. It is possible to incorporate hard and soft constraints by placing extremely high penalty weights on forbidden solutions, and much smaller penalties on soft constraint terms, so that the hard constraints always dominate the evaluation if violated. The moves might involve swaps, additions, or deletions, depending on the model and the nature of the problem.

The serious flaw in the model is that the search can only progress if a neighbouring solution can be reached which improves on the current solution, so it can become trapped in a locally optimal solution, even though better solutions may exist nearby. However, a solution is always produced, though the quality cannot be guaranteed, and there are many improvements on the basic method.

1.2.4.1 Simulated Annealing

This is a widely described improvement on the basic local search model, which allows escape from a local minimum. Dowsland [3] describes the model, in which neighbouring solutions are
generated from an initial solution, selected at random, and evaluated. An improved solution is always accepted, whilst a worse solution is accepted if a randomly generated number exceeds a threshold value. The threshold value is a function of a parameter known as the temperature, and this function is designed so that the probability of acceptance is initially very high, but gradually declines to zero with increasing iterations. This allows the search to explore large areas of the search space during the early stages of a search, but forces the search to become localised later. Wilson et al. [22] report the use of a simulated annealing approach in PTG, a system developed by AEA Technology Rail, used in the production of automated rail timetables. They state that the system “...rapidly produces valid solutions for an hourly pattern service over a complex network”, which was not previously feasible.

1.2.4.2 Tabu Search

The tabu search method is another variation of a local search, which involves maintaining a history of recent moves, and their effectiveness, and using the information to improve the search process. An account by Glover and Laguna [9, Chapter 3] describes various ways in which this is achieved: the simplest involves maintaining a list of the most recent moves, which are then forbidden (tabu) for a certain number of iterations, unless they improve on the current solution. Moves which are not tabu may be accepted, even if the move worsens the solution. At each iteration a number of possible moves are evaluated, and the best non-tabu move is accepted. The neighbourhood (i.e. the set of solutions resulting from all possible moves from a given solution) is often too large to be exhaustively examined, and the selection of good subsets at each iteration is a crucial design issue, as is the number of iterations for which a move remains on the tabu list. Shen and Kwan [20] describe a tabu search method (HACS) for solving driver scheduling problems which models shifts as a series of links between spells of work (each comprising several pieces of work. Moves may consist of swapping pairs of links, swapping two spells between a pair of shifts, adding spells to shifts, or moving individual pieces of work from a spell in one shift to a spell in another. Move attributes represent the change in wage costs or penalty costs due to the move. The system is claimed to be fast, and while not quite capable of matching the performance of the TRACS system, may have the potential to do so with further development.

1.2.5 Genetic Algorithms

Derived from ideas in in biology, Reeves [19, Chapter 4] describes genetic algorithms as “the intelligent exploitation of a random search”, in which a solution to a problem is represented by a data structure called a chromosome. Sets of chromosomes, called populations, are generated, and each has its quality evaluated by a fitness function. The idea is to select the best chromosomes from a generation, and use them as the basis for generating a new population. The new population may be generated by a variety of methods, including simple random changes to
selected attributes, or more complex cross-breeding schemes where pairs of parent chromosomes are combined to produce an offspring which inherits some attributes from each of the parents. The process is an iterative one, with successive generations being created, evaluated, selected and used to create the successor generation. By giving preference to the best candidates in each generation when choosing parents for the next, the method seeks to preserve and combine traits which offer the prospect of a good solution. The process may continue for a set number of generations, or until a chromosome meets some target fitness value.

This approach may be applied to many types of problem, including scheduling. Kwan et al. [15] believe that “it would be very difficult for a pure GA to yield near optimal solutions for driver scheduling.”, but describe a system for the generation of driver schedules in which a genetic algorithm is used to generate candidate shifts, which are passed to a separate scheduling algorithm. This algorithm generates schedules from the shifts supplied, and passes them back to the GA algorithm, which determines the fitness of the schedule (based on the cost of its constituent shifts), and hence the fitness of the chromosomes.

Like the local search methods, genetic algorithms are capable of producing solutions quickly, but the quality of the solutions cannot be guaranteed.

1.3 Improvements

1.3.1 Overview

All these methods suffer from the same problem. In order to guarantee that a solution is optimal, every possibility must be examined, but there are so many possibilities that exhaustive examination is not practicable.

The basic methods described are not very effective in isolation, and the most successful approaches usually involve combinations of techniques. The aim is to reduce the size of the search space in order to make the problem small enough to solve within an acceptable time frame, but without eliminating any solutions which are likely to be optimal. Many techniques have been employed.

1.3.2 Column Generation

Fores et al. [4] describe this technique, used to improve a solution to an ILP iteratively. It takes an optimised problem, and tries to find new shifts which offer a chance of improving the solution. These shifts are added to the set currently available (possibly with others being removed) and the problem is re-solved. This process continues until no shifts offer any further improvement (in practice, there is a limit on processing time too). Various methods may be used to generate the new columns.
1.3.3 Relief Point Selection

The selection could be done in various ways, but the aim is to remove from the initial problem relief points which are unlikely to be used in a solution e.g. in the GMB problem, relief points less than two hours from the beginning or end of a board are less likely to be used because there is a requirement for spells of at least two hours duration. Layfield, Smith & Wren [16] describe a method based on the construction of mealbreak chains, using constraint programming. A mealbreak chain is described as “the case where at least two driver shifts are linked such that one driver finishes a stretch to start a mealbreak at a relief opportunity where another driver finishes a mealbreak (and starts work again)”. It is assumed that the relief points in these chains will be more likely to be used in efficient solutions to the original problem. The generation process is repeated many times using a randomised process. The set of all the relief points thus generated, a subset of the original set, represents a smaller problem which may be solved by existing methods, in this case TRACS II.

1.4 TRACS II

TRACS II is a driver scheduling system developed at the University of Leeds, which has been described in many papers, e.g. [4, 5]. At the heart of the TRACS II system is an ILP solver, but many enhancements have been incorporated to improve performance. It uses the set covering model rather than set partitioning. This is more flexible, but allows the possibility of a piece of work being covered by more than one driver (called overcover).

In [6] Fores et al. state:

There are three stages to the solution process of TRACS II:

1. Generate a set of valid shifts
2. Reduce the shift set (if required)
3. Select a subset of shifts which cover all the vehicle work and minimises an objective function

A number of shift generation methods are available, and sets generated by different methods may be combined in order to obtain a good set - critical to overall performance. Depending on the number of shifts generated, the size of the set may be reduced. The final set must cover all pieces of work in the schedule, or no solution can be obtained, and have enough flexibility for the solver to operate. In [4], Fores et al. state that research showed a minimum of 10 shifts in the set should be capable of covering each piece of work.

The solution method uses the branch and bound method, which starts by solving a relaxed form of the main problem, to obtain a starting point for the main process. In a relaxed solution, non-integer values are allowed; instead of a piece of work being covered (1) or not covered (0)
by a particular shift, it may be partly covered by one shift, and partly by another. The aim is to find a lower bound on the number of shifts which will be needed to solve the main problem. This subproblem is solved by either dual steepest edge simplex, or primal simplex with column generation [7].

The branch and bound phase then uses the results of the solution to relaxation of the problem, to search for an integer solution (i.e. one in which each shift is either used or is not).
Chapter 2

Project Management

2.1 Overall Aims

The overall aim of the project was to investigate ways of simplifying driver scheduling prob-
lems by reducing their size. This was to be achieved by a preprocessing algorithm which
assigned shifts to unusually long or isolated pieces of work, resulting in a partial initial solution
and a smaller driver scheduling problem, which could be solved by existing methods (such as
TRACS II).

This approach was suggested by the attempts at a manual solution to the GMB problem. The
background research indicated that even the best current approaches have difficulties when
faced with large problems (though the definition of ‘large’ is constantly increasing) and the
more successful approaches use combinations of techniques. Both the manual attempts to solve
the GMB problem, and the solution given by TRACS, showed that some pieces have relatively
few viable assignments. If these areas of the problem can be assigned to shifts quickly and
efficiently, and the remaining work assigned by existing methods with no loss of efficiency, then
an overall increase in efficiency may result, particularly for larger problems.

Given the time constraints on the project, this was likely to be limited to a proof of concept,
in which the effect of the division of the problem on the efficiency of the resultant solution was
assessed.
2.2 Minimum Requirements and Deliverables

2.2.1 Present a Brief Review of Approaches to Driver Scheduling Problems

This was necessary both to set the project in context, and to ensure that the work undertaken had not been done previously. This deliverable forms part of the background research chapter of the final report.

2.2.2 Present a Review of Approaches Used in the School of Computing

An aim of the project was to produce software which could be deployed within or alongside the TRACS II system. It was therefore necessary to understand the TRACS II system, and the data file formats which underpin it. This also forms part of the background research chapter of the final report.

2.2.3 Implement Small Sample Problem Using Ilog-5.0

This was primarily a software familiarisation exercise. Conversion of Colin Layfield’s Bus Relief Point Selection code [16] was one possibility, since this had a similar underlying philosophy, but took a different approach. Investigation of this work suggested that the conversion would not be feasible within the available time. Another option considered was full documentation of existing Ilog 5.0 example problems, in a similar manner to those by Smith [21], which would be useful as a tutorial. The deliverable consisted of documented code, and a brief description, which forms part of Chapter 4.

2.2.4 Implement Small System to Attempt to Reduce the Size of the Shift Subset

This was the main deliverable other than the final report, consisting of well-documented source code, implementing an algorithm to reduce scheduling problems in size, together with functional test plans and the results of the tests.

2.2.5 Present An Evaluation of the New Algorithm

The evaluation of the algorithm naturally fell into two parts. The first was a qualitative evaluation of the shifts generated on sample problems. In short, does the algorithm produce shifts which might be produced by hand, and cover similar parts of the schedule to those used in the manual solution? The second part was to determine the effect of using the algorithm on the quality of solutions to scheduling problems. This required a comparison of solutions to problems solved with and without the use of the algorithm. In the first instance, it must be demonstrated that use of the algorithm does not adversely affect the quality of solutions obtained.
2.3 Extensions And Enhancements

It was almost certain that the first version of the software would not perform particularly well, and that a considerable amount of effort could be expended in simply tuning and refining the algorithm and its parameters. Whilst a pure constraint programming approach was intended, other solvers may be used to attack the problem, and these too could be investigated.

Conceptually, the proposed algorithm works as a preprocessor, reducing the size of a scheduling problem before submitting it to existing scheduling software. Assuming the approach proves to be feasible, there are issues concerning the degree to which a given problem can or should be reduced in such a manner, and how this can be determined in practise. Optimising the amount of reduction was therefore an area to be investigated.

Performance testing of the software is an interesting problem in its own right. Driver scheduling problems are known to be NP-hard, so exhaustive testing on non-trivial cases is impossible. Acceptable testing standards seem to consist of running a small set of existing problems, on systems with and without the feature under test, and comparing performance. A possible extension of the project was to examine the nature of ”easy” and ”difficult” scheduling problems, by reverse engineering a problem from a known solution. Such a problem could then be perturbed slightly, and the consequences on difficulty of solving the problem investigated.

2.4 Methodology

Given the experimental nature of the project, an iterative development model was adopted. This called for an initial design and implementation, followed by review of performance, and further enhancements. The development cycle would continue with evaluation primarily on a qualitative basis, until the software produced promising results. At that point, the results could be used for a comparative study of results obtained from TRACS II with and without the use of the partial solution generated from the project software. Further development cycles would build on the results from both stages of testing.

The choice of software was effectively fixed by the option to convert the Layfield code to version 5.0 of Ilog, necessitating the use of Ilog 5.0 and the Linux platform on which it runs. Whilst this limited the possibilities for integration with TRACS II, which runs on the Windows platform, it was considered unlikely that the project would progress to a point where this would be a serious problem. Data files can be easily transferred between the platforms for testing purposes.

Ilog also recommend a 4-stage incremental application development model [14, Ch.16], which is incremental and was used throughout the software development.
2.5  Project Schedule

2.5.1  The Original Project Schedule

The project was backloaded: 10 credits allocated to the first semester and 30 credits to the second. The initial aims for semester 1 were as follows:

1. To investigate a sample problem (GMB).

2. Review Layfield’s work on relief point selection using constraint programming.

3. Gain sufficient familiarity with Ilog 5.0 to decide whether revising the Ilog v3.2 code to v5.0 was a feasible means of gaining fluency in Ilog 5.0, should Ilog be used in the project proper.

4. Background reading.
In the original schedule, 4 weeks were allocated for writing up the work. This was deliberately excessive, and intended to allow a week of slack in the event of delays or problems. It soon became apparent that the decision on whether to convert the Layfield code was a major one, and the familiarisation work on Ilog and the code was brought forward to allow the decision to be made at the start of the second semester in January. At that time, the idea was abandoned, and a revised plan was devised. The small Ilog programs to be produced were to be demonstration or proving examples of key features of the Ilog libraries, to be used in the main program. This should provide a more focused introduction to Ilog, and, by running concurrently with the design of the main algorithm, allow checks on the feasibility of ideas.
2.5.3 The Revised Schedule

<table>
<thead>
<tr>
<th>Week Ending</th>
<th>Tasks</th>
</tr>
</thead>
</table>
| 31-Jan-2003  | Familiarisation With Ilog  
small problem coding |
| 14-Feb-2003  | Coding & Testing Of Sample Code  
Design of New Algorithm |
| 28-Feb-2003  | Writeup  
Design & Coding |
| 14-Mar-2003  | Coding, Testing  
DEADLINE ** Table of Contents  
Draft Chapter (14-Mar-2003) |
MILESTONE ** Demo Software Complete  
MILESTONE ** First Prototype Complete |
| 28-Mar-2003  | Functional Testing, Tuning, & Development |
| 4-Apr-2003   | Performance Testing, Development, Enhancements  
MILESTONE ** final version of code |
| 11-Apr-2003  | Writeup |
| 25-Apr-2003  | Writeup |

Background research was planned to continue throughout February and March, and has been omitted for clarity.

2.5.4 Progress vs. the Schedule

All the specified project deadlines were met.

Several problems occurred during the software development in March, causing some delays, and consequently not all sample programs were complete by the Milestone date set, and the first prototype was produced on 22-March. This resulted in the Milestone for the final version of the code being put back, but the delay was offset by the fact that the development work was being written up as it was in progress, effectively moving that work forward in the schedule. This continued, with testing and tuning on the code still in progress until 25-April.
3.1 Overview

The algorithm can be broken down into the following stages:

Read vehicle and labour data

Assign weights to individual pieces of work

Generate a number of legal shifts, maximising the weights of the pieces of work covered

Output shifts generated and parts of schedule left uncovered

This stage of the development process was mainly concerned with the allocation of weights and the generation of shifts. During this stage a simplified model was used, and the data hard-coded into the program. The iLog software is very flexible, and many approaches to both modelling and searching for solutions are possible. This chapter describes the model as originally conceived.

3.2 Data Structures

The fundamental units of the model are pieces of work and shifts. In the model a shift consists of two spells of work separated by a personal needs break (PNB). Each spell comprises a number of successive pieces of work, though there may be short breaks between these pieces.

Important attributes of a piece of work, crucial for the initial development of the algorithm, are its start and end times, start and end relief points, and its weight, discussed below. These
data are held in a 2-dimensional array, in which each row corresponds to a single piece of work, and each column an attribute.

The data will ultimately be read in from data files, together with additional data, in particular the joinup times between relief points. For the prototype a small hard-coded data set was used, and a single relief point was assumed, with a default joinup time of zero. Later versions will accept multiple relief point locations, which will require data on joinup times to be stored.

Spells of work are represented as arrays of binary decision variables, of length equal to the total number of pieces of work in the schedule. Each shift is modelled as a pair of these arrays. A value of one indicates that the corresponding piece of work is covered by the spell, a value of zero that it is not. In addition, each spell has its own data structure which holds summary details, including its start and end times. By implementing these as constrained variables they are kept up to date throughout the search process. Meta-constraints (constraints on these constraints) enforce labour regulations on shift lengths and meal breaks.

### 3.3 Constraints

At the heart of the model is the basic constraint satisfaction program, which seeks to maximise the sum of weights of pieces of work covered by the shifts generated. There are many constraints, covering valid combinations of pieces of work, maximum and minimum spell and shift durations, lengths of personal needs breaks, adequate joinup time, etc.

#### 3.3.1 Legal Spells of Work

The first group of constraints are concerned with the generation of legal spells of work. In particular, a shift can cover at most one piece of work at any given time, must have sufficient time to get from the end of one piece of work to the start of the next, and must not exceed the maximum length specified in the labour agreement. For efficient shifts to be produced, long waiting periods during a spell (between changes of vehicle) should be avoided. A limit on the maximum waiting time between two successive pieces is also imposed, with combinations which exceed this limit being forbidden. Data for these constraints are recorded in a binary compatibility matrix (cover) which identifies, for each piece of work, those pieces with which it cannot be combined in a single spell. In the matrix, a variable \((i,j)\) is set to 1 if pieces \(i\) and \(j\) cannot be legally combined in a single spell, and 0 otherwise. These values are determined when the data are loaded, and remain constant thereafter.

It is used as follows:

A piece of work is allocated to a spell by binding the corresponding decision variable to 1. The set of available pieces of work can then be determined by examining the row of the combination matrix corresponding to the index of the piece of work. This is illustrated in the figure 3.1. When piece 1 is assigned to spell A, we see from row 1 that pieces 2, 6, and 7
are permitted combinations. The rest are forbidden, and their variables can be bound to zero. Subsequently the search selects piece 2, and by repeating the process we can now also eliminate piece 7 as it is a forbidden combination with piece 2.

The constraint can be expressed on the individual elements of each spell array, by using the scalar product of the spell array and the piece row (or column - the matrix is symmetrical), multiplied by the value of the element itself as a parameter. By constraining this value to zero, we force it to zero whenever there is an excluded combination, only allowing it to take the value 1 when no conflicting pieces are selected. This is illustrated in figure 3.2, where, having selected piece 1, we then attempt to add piece 3, a forbidden combination.

Examining the constraint with respect to piece 1, we first take the scalar product of row 1 and spell A, which is 1. Multiplying this by the value of element 1 gives a total of 1, violating the constraint and preventing the combination from occurring.

Conversely, in the example above where pieces 1 and 2 are selected, the legal combination results in constraint values of zero for both elements. Indeed, piece 6 can be added to the shift without violating the constraint on any element.

This rapidly reduces the search space when building a shift, since, in a realistic problem, the majority of combinations will be forbidden.

The above operations are equivalent to bitwise OR operations on the rows (or columns) of the combination matrix corresponding to the pieces selected. In a scratch-built system, this would be a much faster operation than a series of scalar products.
A variation on the compatibility matrix was to restrict the definition to those pieces of work which can immediately follow (or precede) a given piece of work. This has the considerable advantage of further reducing the search space, but the subsequent control of the search process becomes more complicated. For example, when two pieces are linked in a particular shift during the search, we need to consider only pieces preceding the earlier of the two, or succeeding the latter. Successors to the earlier piece of work are almost certain to overlap with the later one. The idea was shelved for the initial design.

3.3.2 Legal Shifts

In the model a legal shift consists of two spells separated by a PNB. It is necessary to constrain the pairs of spells in each shift so that the total length of the shift does not exceed the permitted maximum, and the time between the two spells meets the minimum length of a PNB. These requirements are enforced by meta-constraints.

Constrained variables hold the values of the earliest start time and latest end time of each piece of work selected in each spell. Meta-constraints on these variables enforce the minimum PNB duration and the maximum shift duration. By using logical OR expressions, any number of permitted combinations can be included. In this model, two legal combinations, representing regular and split shifts, are included.

The imposition of a minimum PNB between spells also forbids the two spells from overlapping, so there is no need for further constraints to prevent this.

3.3.3 Work Constraints

Since a complete set of shifts is not the aim, there is no need for a constraint that all pieces are covered. There is a requirement that each piece should be covered by at most one spell, easily implemented as a constraint on the totals of each column of the spell array.

3.4 The Objective Function

The objective function determines which of many possible solutions is the best in the context of the problem. In Ilog the objective function is declared as an object, and a goal, to maximise or minimise the value of that function, is added to a model. Here we want shifts which cover pieces of work that have particular features, which were highlighted by the original work on the GMB data:

- unusually long duration
- running at times of peak demand
- few efficient links to other pieces of work.
Each piece has a weight assigned to it, which reflects the presence of these attributes by higher values and their absence by lower values. The value of the objective function for any given shift is the sum of the weights of all pieces of work covered by that shift, and the search process returns the shift which maximises the value of the objective function.

The function can be relatively complex, since it will be used only once at the start of the algorithm. Since the function is intended to match a subjective set of priorities, it can only be determined by experiment, and there is some danger of overfitting to a particular set of data. Since one of the aims is to generate efficient shifts, the function will be based on the total driving hours, thus maximising the work covered, with bonuses for targeted pieces of work.

This requires further data, describing the links available to each piece of work and the number of simultaneously active boards. The number of pieces of work which overlap with a given piece was used as a crude measure of peak time period, and the number of incompatible pieces of work, the sum of the appropriate row of the cover matrix, was used as a measure of lack of links.

The objective function used was a weighted sum of these terms, with the weights used to be determined by experiment.

### 3.5 The Search Process

Ilog allows considerable control over the search process. The order in which variables are explored, the order in which elements of a particular variables’ domain are explored, and the times at which these decisions are made, can all be specified, but the best method often depends on the model [13, p126] so this must be determined experimentally. The default search parameters will be used initially, with the search returning a specified number of shifts.
Chapter 4

Implementation

4.1 Familiarisation With Ilog

The Ilog software used in this project is part of a suite of optimisation libraries. This section briefly describes the main features of the Ilog Concert Technology and Solver packages which are used in the project. Further details can be found in the relevant user [12, 14] and reference manuals [11, 13] which are also available online as part of the standard installation. Some of the features described are demonstrated in sample programs described in section 4.2.

4.1.1 Overview

A key part of the Ilog approach is the separation of the processes of defining and solving problems. Ilog Concert Technology provides facilities for modelling problems. Modelling components include platform-independent data types, and extensible arrays, constrained variables (which have a domain as well as a type), and a wide range of constraint types, including user-defined constraints, expressions based on constrained variables, and goals and objectives. A model may then be solved by extracting the relevant data from the model, and applying one of a range of solution methods, e.g. Solver or CPLEX. The Concert Technology User Manual[12] states:

ILOG CPLEX supports mathematical programming with ILOG Concert Technology, including primal and dual simplex algorithms, barrier algorithms, a network simplex solver, and a branch & cut solver for mixed integer programming.

ILOG Solver provides the facilities for constraint programming with ILOG Concert...
Technology. It serves as the basis for the other constraint programming products from ILOG, such as ILOG Scheduler, Dispatcher, and Configurator. It also offers the means for extending constraint programming facilities yourself.

4.2 Ilog Demonstration Code

This work consisted of the production of a series of small test programs, each designed to demonstrate or prove a particular feature required in the main program. The work was done in parallel with the design of the main algorithm, and the coding of the prototype, so that ideas for the main code could be tested in a small-scale environment. This approach was adopted after a number of problems with the Ilog documentation and sample code including:

- undocumented syntax for variable declarations used without explanation or clarification, in Ilog example programs.
- examples quoted in the manual purporting to show the right and wrong way to use a particular function, but which were identical.
- example code containing spelling errors in function calls.
- example code containing alternative, commented-out versions to demonstrate different approaches, which fail on activation.

The programs described provide a set of examples of the most basic elements required to solve problems using Ilog, in very small increments. Demo03, Demo04, Demo06, and Demo10 show important features in the process, with the remainder being intermediate stages.

4.2.1 Demo01-Demo03

Demo01 starts at a trivial level, finding an assignment of different values to two decision variables, each with a domain of \( \{0,1,2\} \). Demo02 finds all possible solutions, and Demo03 introduces an objective function, the sum of the variables, which is to be maximised. The search is set to produce all solutions. When an objective function is present, a new solution must not only meet all the constraints, but must improve on the best value of the objective function previously found. By extending the domain of the variables to \( \{0..5\} \) the search order and backtracking is demonstrated.

4.2.2 Demo04

This code demonstrates how the search order can be controlled. By using the IloGenerate() or IloBestGenerate() functions it is possible to specify the events which trigger re-selection of the variable domain to be searched - respectively, when the the domain of the current variable
is exhausted, or when it is simply reduced. A range of options are available to control which variable is selected. This code uses the built-in function IlcChooseMaxSizeInt() to specify that the variable with the the largest domain should be tried first. This is not usually an efficient method in practice, but is used here to demonstrate how the search proceeds, since, as the domain of one variable is reduced by the search, the search switches to the other variable. This can be seen in the program output.

4.2.3 Demo05-Demo06

This code demonstrates repeated solving of a problem, with the addition of constraints between each search. Demo06 shows a key feature of the main algorithm, that of modifications to a model between successive searches. In this example an objective function was added between two searches; constraints can also be added.

4.2.4 Demo07-Demo10

Demo07 was an attempt to determine how an objective function could be modified. It demonstrates that IloInt variables are not extractable, and are therefore constant within an instance of a model. In Demo10 the problem is to select two items from a set of six, so as to maximise the sum of their values. After each search is completed, the variables selected have their decision variables constrained to zero, to prevent them being selected in a subsequent search. This was verified by repeated searches, each producing the maximum sum available from the unused values.

This mechanism is the basis for the greedy heuristic used in the shift generation phase of the main algorithm.

4.3 The First Prototype

4.3.1 Design

The aim in producing a prototype was to demonstrate the principle of shift generation from a set of pieces of work, and to provide a test platform to be used to determine approximate values for parameters used in the weight function. The design is described in Chapter 3. In order to do this, a simplified model was used. Relief point locations were not used, and no time was required for a driver to change vehicles. Spells were of not more than 5 hours duration, and shifts comprised two such spells with either a maximum duration of 10 hours including a PNB of at least 1 hour, or a maximum duration of 12 hours including a PNB of at least 3 hours. The data set used for development testing was the GMB data set, or a subset of the data from which several complete boards were removed.
The prototype was produced using an incremental development process, which allowed testing to be carried out at each stage.

4.3.2 Data Structures

Data about the pieces of work - start and end times and locations, identifiers and weights - are stored in a 2-dimensional array. At this stage the data were hard coded and assigned directly. Compatibility data was generated from the raw data, and variables (which may be passed as command line arguments) representing maximum spell duration and joinup time, and minimum PNB duration. The original design included a further constraint on the maximum idle time permitted between pieces of work in a single spell, in order to eliminate inefficient shifts. This idea conflicts with the concept of pieces of work which could be covered in a single half-shift - a spell consisting of three consecutive pieces of work on a single vehicle is both legal and efficient, but the idle time constraint considers the first and third to be an unacceptably inefficient and therefore forbidden combination. This was revealed during testing, and this part of the compatibility requirement was removed.

4.3.3 Weights

Each piece of work must have a weight, a measure of priority for assignment which is used in the objective function. This was originally conceived as a function of the duration of the piece of work, the number of pieces with which it could share a half-shift, and the number of other vehicles running simultaneously (vehicle demand).

The prototype weight function determines these values as follows:
- duration: end time - start time
- flexibility: the number of pieces of work with which it could be combined
- demand: number of other pieces overlapping in time with the specified piece

They are combined into an overall piece weight by a weighted sum. The start and end times are directly available, and the number of pieces with which a piece can share a half-shift can be easily calculated from the compatibility matrix. The number of pieces overlapping with a given piece of work is determined from the start and end times of pieces.

A set of default weighting factors were hard-coded, with optional command line parameters available to modify them.

4.3.4 Decision Variables and Constraints

The next stage in the process was to add the decision variables which represent solutions to the problem. An array of binary (0,1) variables, equal in size to the number of pieces of work in the problem, is used for each half-shift to be generated (the target number having been specified
by the default value or a command-line parameter). Initially the target number of half-shifts was limited to one.

Constraints were added to restrict the combinations of pieces selected to those permitted according to the cover matrix, as described in the design chapter. This was implemented using the IloScalProd() function. Constraints were added to prevent a piece from being covered by more than one piece of work.

4.3.5 Objective Function

The objective function was also added, at this stage in the form of a constraint on each half-shift, forcing the sum of weights of pieces covered by that half-shift, to exceed a specified value. This was useful for testing purposes, since it allowed all possible solutions to be examined, rather than only the optimal one. After testing, the objective function was then added to the model as a maximise goal, and tested on the 11 piece data set.

4.3.6 Functional Testing

For the prototype, a data set consisting of 3 vehicles and 11 pieces of work were hard-coded into the source file. The raw data arrays were checked by loading and then reading back data from them. The compatibility matrix (cover) was tested using variations on the data set, joinup time, and spell duration parameters. In each case the resulting cover matrix was output and checked by hand against the original data. The idle time constraint was found to conflict with the concept of pieces of work which can be covered by a single shift. For example, given 3 consecutive pieces of work on a single board, each of 1 hour duration, with a 3 hour maximum spell length. The three pieces are a legal spell, but the idle time condition makes the first and last pieces a forbidden combination because there is an excessive gap between them. If the weight function is based on duration, then the optimal shifts will tend to be those with least idle time, so in practice this should make little difference to the shifts generated, though it does mean the solution space is larger, slowing the search. The idea was shelved.

The weight generation function was tested using a variety of factors for the 3 component terms. By using factors of zero for some terms, each component was checked both individually and in pair combinations. By using small data sets it was easy to check the results by hand.

By adding a constraint, the decision variable for a piece can be set to 1 for one half-shift, forcing that piece to be included. The effectiveness of the compatibility constraints can then be checked easily by making modifications to the data in the cover matrix so as to forbid or permit specific combinations, including allowing all or forbidding any combinations with a particular piece. Checks were also made without this additional constraint. By generating multiple half-shifts on small data sets, the constraints on repeat selection were also verified.

The objective function (not the GOAL) was tested by first implementing it as a constraint on
Table 4.1: Shift Generation Times for Prototype

<table>
<thead>
<tr>
<th>Set Size</th>
<th>Number Of Shifts</th>
<th>Time To Generate</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>1</td>
<td>0.03</td>
</tr>
<tr>
<td>36</td>
<td>2</td>
<td>3.9</td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>timed out</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>3.44</td>
</tr>
<tr>
<td>104</td>
<td>2</td>
<td>timed out</td>
</tr>
</tbody>
</table>

4.3.7 A Larger Data Set

After testing of the model using the initial small data set, the full GMB data set, consisting of 12 vehicles, and 104 pieces of work, was hard coded, and the earlier tests repeated using a series of subsets of increasing size. The results are given in figure 4.1. This confirmed the severe performance degradation with large data sets, in particular when more than one half-shift was to be produced.

Whilst the search process was clearly very inefficient, the tests did confirm that the piece data and weights, and the compatibility matrix were correctly initialised, and, by setting selected piece weights manually, that the objective function was working correctly.

Some experimentation with the factors in the weighting function was done at this stage, but the performance on large data sets was so slow that it was not practical to do a thorough evaluation.

4.3.8 Evaluation

The prototype generates a specified number of half-shifts (spells of work) which contain no overlaps and comply with limits on maximum spell duration. When more than one spell is generated, each piece of work is covered by not more than one of them. The spells are chosen so as to maximise the sum of weights of pieces of work covered by all the spells. By varying the means by which the weights are calculated, different features of pieces of work can be given priority.

With the introduction of the larger GMB data set, the need for improving the computational efficiency was made very clear. Performance was very poor, even on relatively small subsets of the data, and particularly when more than one shift was to be generated.

The performance deterioration noted with larger data sets was due to the far greater number of
permutations to be examined in the search. A further problem with the generation of multiple half-shifts was symmetry. This is a known issue in constraint programming [14]. It arises because when multiple component solutions are generated (in this case half-shifts), the order of the components can be changed to produce other solutions. For all practical purposes these are identical, but the labelling of the components (by assigning them to variables) imposes an ordering even though the ordering is not meaningful in the context of the problem.

The inclusion of restrictions on idle time permitted between consecutive pieces of work during a spell cannot be implemented as part of a selection constraint on pieces of work which can be covered by a single spell.

4.4 The Second Prototype

4.4.1 Design Changes

The primary aim at this stage was to move the search results from individual spells to complete shifts and to improve the computational efficiency by eliminating symmetry effects. The secondary aim was to investigate suitable weighting functions for pieces of work, so that the target areas of a schedule were given priority.

A complete shift was modelled as two spells of work, with constraints between the spells to enforce a minimum PNB duration and a maximum total shift duration.

Informal tests confirmed that by forcing the inclusion of one piece of work, the efficiency of the search was greatly improved. Prior to the search phase, the piece of work with the highest weight is determined, and the corresponding decision variable in the first spell bound (by an additional constraint) to 1, to force inclusion of the piece in the shift. This eliminates symmetry problems and greatly reduces the search space, giving a substantial improvement in performance. A consequence of this is that the generated shift may have the later of its two spells in the first array, depending on the timing of the bound piece.

The need to identify and bind the highest weight piece of work makes it impossible to generate more than one shift at once (since it may be possible to cover the two highest weighted pieces in one shift). A single shift is therefore generated, after which the model must be updated to reflect the assignment of work to a shift, before generating the next shift.

4.4.2 Generating Complete Shifts

Decision variables for two spells were created, with constraints between them to enforce a minimum PNB duration and a maximum total shift duration (as specified by control parameters). These constraints can be compounded (using logical AND and OR) so that many options can be combined. In this case the options coded were a split shift, requiring a minimum PNB of 3 hours and a maximum shift length of 12 hours, or a regular shift, requiring a minimum PNB of 1 hour and a maximum shift length of 10 hours. Individual spells retained all the constraints
Shifts Generated | 1st Piece Bound? | Time To Generate
---|---|---
1 | yes | 29
2 | yes | 53
3 | yes | 119
4 | yes | 123
1 | no | timed out

Figure 4.2: Shift Generation Times - Second prototype

introduced in the prototype. The objective function was modified to maximise the sum the weights of the pieces of work covered by all the generated shifts.

4.4.3 Improving Computational Efficiency

Modifications were made to the search component of the software. A simple search of the weights data was used to determine the piece with the highest weight. The corresponding decision variable in the first spell was bound to the value 1 (to force selection) by a named constraint. Naming constraints is generally optional in Ilog, but it is essential if the constraint is to be removed later, as is the case here. Additional code was added after the search and results output. The constraint forcing selection of the first piece must be removed. The weights of all selected pieces must be set to a large negative value, and their selection variables constrained to zero, to prevent them being considered for inclusion in the next shift. The whole search process was contained within a loop, to allow searches for any specified number of shifts.

4.4.4 Functional Testing

Generation of complete shifts was first tested with single shifts being formed, using small subsets of the GMB data and variations on shift and PNB lengths. The tests were repeated for regular and split shifts separately, and with both types available. By setting artificial weight values, it was possible to steer the algorithm towards particular shifts to check that both types can be formed correctly. An error in the constraint on the duration of split shifts was identified, traced to a logic error, and corrected.

Correct binding of the highest weight piece was tested by assigning a very high weight to one piece after the normal weight determination was completed, and checking for its inclusion in the subsequent shift. In addition, the domains of decision variables were output during testing, to confirm that the reason for inclusion was indeed that the appropriate variable was bound. Binding one piece of work to the shift to be generated had the marked effect on performance that was expected, as the results in figure 4.2 show.
4.4.5 Calculation of Weights

A series of tests were carried out to determine a suitable weight function. The tests started with runs using the individual terms from the original objective function, in order to check the values generated; length, number of overlapping pieces of work, and number of compatible pieces (i.e. pieces which could be covered by a spell containing a given piece).

These values were then used as the basis for a series of trials, each aimed at developing a weight function which targeted what were seen as the crucial pieces of work. As it was expected that the algorithm would cover at most 25% of the available work, and a 16 shift solution was obtained by TRACS II on the GMB data, the tests were set to generate 5 shifts (25% of the work plus a one shift margin).

Using fewest overlapping pieces of work did target the expected areas of late evening and early morning, but produced a badly fragmented set of shifts and left uncovered pieces of work which would be difficult to cover. The shifts had many changes of vehicle, which is inherently inefficient.

Using normalised overlaps (ratio of duration to number of overlapping pieces), in an attempt to reduce the effect of longer pieces naturally having more overlapping pieces of work than short ones, also gave poor shifts.

Weighting by greatest number of prohibited combinations gave promising results, covering all three late running boards with the first three shifts, but again resulted in individual pieces being selected from several boards because of slight differences in weights. The remaining work would have been impossible to cover efficiently.

An attempt to prioritise both very high and very low demand, by determining the maximum number of overlaps for any single piece, and weighting each piece as the maximum of the number of overlaps, and the number of overlaps subtracted from the global maximum, was tested. It is possible to change the balance in emphasis between high and low demand simply by adding a constant to the appropriate term. Eg

\[ \text{weight} = \max(\text{overlaps} + A, B + \text{MaxOverlaps} - \text{overlaps}) \]

By increasing A the weighting gives peak period work a higher priority, and by increasing B off-peak work gets a higher priority. Several variations on this model were tried but all seemed too sensitive to minor variations in timings, particularly with respect to the selection of the first piece of work. This is perhaps due to insufficient variation in the weights themselves.

In summary, two major problems were noted.

Isolation is caused by the use of weights to select both the bound piece and subsequently the pieces covered by a shift. The greedy heuristic approach takes no account of the pieces of work not selected, which can easily be isolated, leading to a subproblem which cannot be efficiently solved.
Diagram 4.3 shows part of the GMB data. With a greedy heuristic, and any reasonable set of weights which favour isolated and long pieces of work, a spell including pieces ACD is likely to be favoured over one including xAB, especially if the maximum length of a spell is such that yzCD exceeds the maximum legal spell length, but wxACD does not. If the spell xACD was generated, and these were the only active vehicles, then piece B would be left isolated, with no means of including it in an efficient shift.

Fragmentation is a similar effect with similar causes - choices between pieces with similar weights and running times are made by highest weight rather than neatness, and consequently the generated shift comprises pieces taken from several boards. Whilst the shift has a slightly higher objective function value than one comprising pieces from a single board, the cost is the breaking up of other boards into small spells which cannot be efficiently covered.

Of the results obtained, the fewest compatible pieces model was marginally the best, but there are two critical considerations. First, the weights were obtained from tests on a single data set and may be well suited only to that model. Second, the tests demonstrate that using the same weight function to determine the first piece to be bound, and to weight the resulting shift, is almost certainly a flawed strategy. The features of the critical pieces for first piece selection are very different from those which build the desired pattern of shift.

### 4.4.6 Evaluation

The model uses only one pair of decision variable arrays, and generates one shift at a time using a greedy heuristic, in which the highest weight piece of work available is bound to the shift, forcing its inclusion. The shift generated then maximises the sum of weights of selected pieces. Once a shift has been generated, the weights of all selected pieces are set to a negative value, and constraints are added to force the corresponding decision variables to zero, thus preventing their re-selection.

A user-specified number of shifts is generated, each produced as two half-shifts, constrained so that there is an adequate PNB between them, and the total shift length complies with the
labour regulations for split or regular shifts.
The vehicle schedule data are still hard-coded at this stage.
The use of piece weights for both selecting an initial piece of work to be included in a shift, and for evaluating the shift subsequently generated, is not effective. There are two conflicting issues in using the greedy heuristic; we want to maximise the assignment of awkward pieces, but not if it results in the creation of even more difficult assignment problems for the remaining work. On this basis it seems logical to separate the selection process for the first piece from the selection process for the others (i.e. the weights).

4.5 The First Working Version

4.5.1 Design Changes

The essential development was the reading of data from TRACS files. The data used was limited to the vehicle relief opportunities and joinup times. The vehicle data were to be stored in the existing structures, and a 2-dimensional array used for the relief point travelling time data.

The algorithm was modified to use different functions for the selection of the first piece of work to be included in each shift, and for the function used to generate weights used in the objective function. From a review of earlier tests, the most promising model was the selection of the first piece based on fewest overlapping pieces of work, so as to focus on early and late running vehicles, which have the least flexibility. The length of the piece was used directly as a weight. This was done partly to promote the generation of shifts which cover the greatest amount of work, but also to minimise the fragmentation problems seen with more complex models, by penalising any idle time between pieces of work.

The definition of overlaps used until now was “any two pieces of work which run simultaneously at some moment”. In order to give increased weight to the earliest and latest pieces, which only have links in one direction, a slightly different model was adopted: the number of pieces of work overlapping with the start point, plus the number overlapping with the end point, for a given piece. An overlap score was determined for each piece of work, determined by first finding the maximum number of overlaps for any single piece. The overlap score for a piece is then obtained by subtracting the number of pieces overlapping it from the maximum found for any piece, so that pieces with few overlaps have high scores. The available piece with the highest score was to be bound to the shift prior to the start of each search phase. This in turn required modifications to the selection code, as only unassigned pieces should be considered. This can be controlled by testing the piece weight, which is negative if the piece is assigned.
4.5.2 Reading Data Files

The code to read TRACS data files was essentially a modified version of Layfield’s input code. It was assumed that the data files used would normally be checked for validity by the TRACS module, so a valid file format is assumed. Only the relief point locations and transfer times, and vehicle schedule data were used, and additional code was required to process the data into the form used in the model. TRACS describes schedules in terms of relief opportunities, and successive pairs were used to generate the pieces of work used in the model. Separate handling was required for relief points with a relief point location number of zero, which in TRACS indicates a waiting time period at a relief point. In such cases, the preceding relief point represents the arrival time and location, and the relief point with a zero location number indicates the departure time from the same location.

A 2-dimensional array was added to hold the joinup time data. Relief points are numbered consecutively from 1 in TRACS files, so with an offset adjustment, the values in the piece data could be used directly to access the array.

4.5.3 Shift Generation

An array was introduced to hold the overlap score for each piece of work, and the code to generate the values was modified to meet the new definition.

The code to populate the cover matrix was modified to handle multiple relief point locations, and the consequent differences in joinup time, depending on the relief point locations concerned. This required that the relief point locations (which were already stored in the piece data structure) were used, together with the joinup matrix, to determine whether there was adequate time available between two pieces of work. If there was not, the combination of pieces was prohibited.

4.5.4 Testing

The data file reading code was tested by verifying the printed output from a range of data sets, including sets containing vehicle routes with wait times. The new data structures were checked by reading and outputting sets of test values.

The processing of multiple relief point locations was tested using small data sets with varying joinup times, by checking the output values in the joinup and cover arrays.

The weight and overlap data were also checked using output values, including data sets in which, after reading in the piece data, weights of some pieces were set to negative values to check they were no longer considered.

No problems were found.
Figure 4.4: The Generation Algorithm

4.5.5 Evaluation

The current design of the algorithm is given in figure 4.4. Vehicle schedule and relief point data are read from TRACS VEH files, though traction types, an optional feature in TRACS II are not supported. A specified number of shifts, which comply with labour regulations, are generated. Multiple relief point locations are handled appropriately, as are waiting periods during a route.

The shift generation process uses a greedy heuristic in which the inclusion of the piece of work with the fewest number of overlapping pieces is forced. Shifts are constructed so as to maximise the sum of weights of all pieces of work covered by the shift. At this point, piece weights are simply the durations of the pieces of work.

This version of the code algorithm meets the original specification, in that it generates a partial set of shifts from a set of vehicle data, but the quality of the shifts generated is very poor. The results from the GMB set are given in figure 4.5. They show a tendency to fragment, leaving isolated pieces of work. The shifts generated are targeting appropriate areas of the schedule. An attempt to solve the remaining sub-problem using TRACS was, as expected, unsuccessful, as some work could not be covered.

The program still runs very slowly, taking up to four minutes to generate five shifts from the full GMB data set. Fragmentation is still a problem, and needs to be addressed as it seriously compromises the
chances of a soluble sub-problem being produced.

A problem with the selection of the piece to be bound is that several pieces of work can have the same overlap score, particularly after the highest priority pieces are assigned, and when this occurs, the first piece examined with that value is selected.

![Figure 4.5: Shifts Generated - First Version.](image)

### 4.6 Improving The Model

#### 4.6.1 Design

In order to improve the selection process for the piece of work to be bound, modifications were made to the selection function. By using an appropriately weighted linear sum, it was possible to use the overlap score as the primary factor, with another parameter as a ‘tiebreaker’. The number of direct links available to a piece of work (an idea that was tried and abandoned earlier) was used. In this context a link is a piece which can immediately precede or succeed a given piece. To be a valid link, the pieces must not be already assigned, and joinup time must be adequate, but a limit is also placed on the maximum delay between the end of the first piece and the start of the next. The number of links is a measure of the flexibility available in covering a piece of work. The weighting heavily favours the number of overlaps, so that the number of links is only considered when the number of overlaps are equal, with the fewest links considered.
taking priority. This should be done in the same manner as the overlaps, generating a score. In addition, the overlap scores were recalculated after each shift was generated. The constrained variables controlling the length of the spells can be used to determine the difference between the total spell length and the sum of lengths of pieces of work covered (i.e. actual driving time). This can be used in the objective function to penalise non-driving time. Since this most commonly occurs when a driver changes vehicles, it has the effect of penalising shifts which require such changes. Larger coefficients penalise idle time more, and make changing vehicles more costly. The update overlap function can be called between cycles to update the overlaps between shifts. This keeps the emphasis on the isolated pieces of work. Finally the selection function uses a weighted sum of overlaps and available links (an idea that was tried and abandoned earlier). The weighting heavily favours the number of overlaps, so that the number of links is only used as a tiebreaker when the number of overlaps are equal.

### 4.6.2 Implementation

A matrix of links was added; this is a 2-dimensional array which records whether pieces of work can form successive parts of a spell, indicated by a value of 1, or cannot, indicated by a value of zero. In addition, any piece of work of length greater than the minimum spell length has its row-column set to 2, to reflect the fact that it does not need to be linked to another piece of work to produce an acceptable spell. The links score is then generated in a similar manner to the overlap score, by subtracting the number of links for each piece from the highest number found for any piece, so that fewer links result in a higher score. The code to select the first piece was modified to use the formula

$$1000 \times \text{Overlap\_Score} + \text{Link\_Score}$$

and an additional function call updated the overlap data between shift calls.

### 4.6.3 Testing

The links matrix was tested by outputting the contents generated and checking by hand. The selection procedure was checked by output of the values generated on a small problem set, modified to include pieces with no overlaps or permitted links. The updating of overlaps was verified only by checking changes between shift generation cycles, as the function itself had already been tested.

Figure 4.6 shows the shifts generated by this version of the code from the GMB data, using a maximum regular shift length of ten hours, or a split shift of at most twelve hours. The shifts are numbered in the order in which they were generated. The first three shifts cover the key areas of the schedule, the late evening services on boards 5, 6, and 9, and the remaining work
is not fragmented in a manner which would suggest the remaining work could not be efficiently covered. Shift 4 shows the first signs of creating a sub-problem which is infeasible, by leaving the piece from 15:16 to 16:16 on board 9 isolated. The results from this trial were used for an evaluation by submitting the remaining work as a sub-problem to TRACS II, described in chapter 5.
The aim of the algorithm was to reduce the size of the problem set, without affecting the quality of the solution obtained. If this could be done, a performance benefit may be possible, if the time taken to run the algorithm and to solve the smaller problem is less than the time taken to solve the original problem, without using the algorithm as a preprocessing step. This project was concerned with the first part of the problem: is the quality of schedule produced adversely affected?

The first phase of the evaluation was quantitative. The shifts generated should be efficient, and should, as far as possible, cover pieces of work with the required attributes; long, isolated, or running at times of very low or very high vehicle demand. The second is a more rigorous quantitative assessment, in which the quality of solution of a problem solved using the preprocessing algorithm is compared with that obtained directly. In this stage the primary concern is that the solutions generated are at least as good as those obtained without the problem reduction step. A further consideration is the computational efficiency of the preprocessing code.

Trials on TRACS II used the prespecification feature of TRACS II to assign the shifts generated by the preprocessing algorithm. The full set of generated shifts was used initially, and if no solution was possible shifts were successively removed (starting with the last shift generated) and the problem repeated.

TRACS II solves the original GMB problem in a few minutes, producing a solution requiring 16 shifts, of which 7 are split shifts.
5.1 First Working Model

The first trial on TRACS II used data from the first version of the software and attempted to find solutions using the first n shifts generated by the algorithm. The results are given in figure 5.1, where ‘ns’ indicates no solution - TRACS was unable to generate shifts to cover some pieces of work, so no solution could be produced. It is significant that even one pre-set shift can have such a considerable effect on the quality of the complete solution obtained.

As expected, the results showed that use of the preprocessing algorithm adversely affected the quality of the solution, and further development was required.

5.2 The Final Version

Running the final version of the code on the GMB data produced the shifts shown in figure 4.5, with an execution time of several minutes. The shifts are numbered in the order of their generation. The results are an improvement over those generated by the previous version, in that there is less fragmentation. The shifts generated cover the late evening parts of the schedule when few vehicles are operating, and are efficient in that the hours worked are close to the maximum times permitted by the labour regulations.

There are signs of problems in two points; the last piece of work on board 8 and the piece on board 9 starting at 15:16 both appear to be isolated, and potentially will be difficult to assign to an efficient shift.

The results were used in a trial run using TRACS II, again by using the pre-specification facility. The results are given in 5.2.

The solution obtained using three predefined shifts included almost 3 hours of overcover (i.e. work covered by two shifts).
5.3 Conclusions

Even on the set on which the algorithm was developed, only two pre-specified shifts can be assigned before the quality of the overall solution is affected. Given that there is a probability that, to some extent, the algorithm will be tuned to this data set, the results are not encouraging.

The implementation of the algorithm is very slow, taking around three minutes to generate five shifts from the GMB data. This is of the same order as the time taken by TRACS to solve the whole problem, despite TRACS running on less powerful hardware.

An algorithm of this type would be of most use in dealing with large problems, where greater benefits in performance could be realised, but without a great increase in performance, the existing implementation is incapable of processing such problems within a reasonable time span.

5.4 Extensions

It is apparent from the first evaluation that even assigning a single shift can adversely affect the quality of the complete solution obtained, if the shift chosen is a poor one.

There is a cut off point beyond which the greedy heuristic reduces the quality of the overall solution. It is known that greedy heuristics cannot efficiently solve problems of this type, so there must be a point at which the heuristic fails. This suggests that the state of the remaining uncovered work, rather than the shift or shifts generated, could be the basis for an objective function when attempting to generate partial schedules. The evaluation function obviously would not indicate how the remaining work would be covered, but could indicate the presence of obstacles to an efficient schedule, such as isolated pieces of work. If such a measure, perhaps using parameters like those used in this project, were devised, it may be possible both to generate better shifts, and to stop the search at a suitable point automatically, based on the expected difficulty of covering the remaining work.

One of the key components of the algorithm is the constraint on each decision variable which enforces the selection of legal combinations of pieces of work. As implemented, a scalar product

<table>
<thead>
<tr>
<th>pre-set shifts</th>
<th>shifts in solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>ns</td>
</tr>
<tr>
<td>4</td>
<td>ns</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 5.2: TRACS Trial Final Version
is used, and this is computationally very expensive, especially as the values are binary. A much more efficient model seems possible using bitstrings and bitwise AND and OR operators to achieve the same result.

The objective function might be better based on the uncovered work, rather than that assigned to shifts, since it is crucial that the remaining assignment problem can be solved by TRACS II. If a measure of difficulty of the remaining sub-problem can be devised, perhaps with parameters like those used in this project, it may be possible both to generate better shifts, and to stop the search at a suitable point automatically, based on the expected difficulty of the remaining work.
Bibliography


Appendix A

This project gave me an opportunity to look a little deeper into an area which I found particularly interesting, and has consequently been very enjoyable. As a research-based project, it also proved immensely frustrating, satisfying, and highly educational. It is in the nature of research projects that solutions are not guaranteed (as was the case here). This does not appeal to everyone. I found the idea of trying something completely different very appealing, but it also proved challenging, and at times very stressful.

One source of difficulty was the combination of a research project, requiring familiarisation with a lot of new material, and the use of unfamiliar software (Ilog Solver) which also had to be learned from scratch. Whilst this is sometimes unavoidable, as in this case, with hindsight I would prefer to avoid such a situation if possible.

Good project management is essential, and starts long before the project does. The administration and setting up of final year projects takes half of the first semester, and cannot be avoided. To make the best use of available time I took five of my remaining eight modules in the first semester, and three in the second. This allowed me to concentrate on regular modules before Christmas, and the project in the the new year, which I felt worked very well.

I would also recommend learning about \LaTeX during the first few weeks of the year, before a project is assigned. It is easy to use, and its referencing system (Bibtex), alone is worth the effort. Learning Bibtex before I started background research would have saved me quite a bit of effort during the writing up of that work.

I scheduled over a month for the writing up, despite also planning to write up some of the work as I went along. This included a week of slack in case of problems. This proved to be about right, and I would not recommend assigning less than a month for the writing up of the work. Throughout the project I kept notes on everything I did. The records consisted of a lab book, in which I logged all the work and testing done, and a diary, which was very informal and used to record thoughts and ideas as they occurred, as well as to track progress. Both proved
invaluable, both during the writing up, and on several occasions when I needed forgotten details of work done weeks before. There were also occasions when I had failed to record what turned out to be crucial details, and had to repeat previous work.
Appendix B

Two data sets were used for testing; the full GMB data set, and a small set, purpose built and containing items necessary for testing but not present in the GMB data - work with a start time of zero, a board containing a waiting period, multiple relief point locations, and isolated work. The approach was to output results to file and verify the data manually. In some cases the results were checked using subsets of the data, e.g. testing the derived data, which on the GMB data set comprised several large 2 dimensional arrays.

<table>
<thead>
<tr>
<th>Section</th>
<th>Tasks</th>
<th>Comments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Loading Primary Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Numbering pieces of work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.2</td>
<td>Start and end times</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.3</td>
<td>start and end locations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.4</td>
<td>Zero departure times</td>
<td>GMB data set has no zero departure times and was not used in this test.</td>
<td>Pass</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>1.5</td>
<td>Waiting en route</td>
<td>GMB data set has no boards with en route waiting and was not used in this test.</td>
<td>Pass</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>Section</td>
<td>Tasks</td>
<td>Comments</td>
<td>Result</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>2</td>
<td>Generated Data: Piece Attributes</td>
<td>Duration of Piece</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Weights</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>2.2</td>
<td>Overlaps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Tasks</th>
<th>Comments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Generated Data: Matrix Values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Cover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>3.2</td>
<td>Links</td>
<td>Pieces of duration greater than the minimum spell length should have their [row][col] value set to 2.</td>
<td></td>
</tr>
<tr>
<td>3.2.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Final1 Data</td>
<td></td>
<td>Pass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Tasks</th>
<th>Comments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Run-time Updates</td>
<td>Finall data set was not used for these tests</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Weights</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>4.2</td>
<td>Links</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>4.3</td>
<td>Links</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>Section</td>
<td>Tasks</td>
<td>Comments</td>
<td>Result</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>5</td>
<td>Shift Generation</td>
<td>Final1 data set was not used for these tests.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Regular Shifts Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>5.2</td>
<td>Split Shifts Only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>5.3</td>
<td>Both Enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
<tr>
<td>5.4</td>
<td>Changes To Driver Hours</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4.1</td>
<td>GMB Data</td>
<td></td>
<td>Pass</td>
</tr>
</tbody>
</table>