Solving The University Timetable Alteration Problem

by Zachary Winter

Submitted to

the Department of Computing

in partial fulfillment of the requirements for the degree of

Bachelor of Science (Science) (Honours)

at

CURTIN UNIVERSITY

December 2016

Contents

List of Figures

Abstract

The University Timetabling Problem is a much-studied area of research, with papers dating back to the 1960s (Lawrie, 1969). Many papers have been published on the subject, and the problem has spawned multiple International Timetabling Competitions (McCollum et al., 2010). Research has been undertaken in various approaches including Genetic and Memetic Algorithms, Simulated Annealing, Tabu Search, as well as various nature-inspired approaches such as Artificial Bee Colony, Ant Colony Optimization, and various combinations of the algorithms.

This thesis attempts to investigate the University Timetable Alteration Problem, which we define as attempting to fix a timetable that has been damaged due to some room or timeslot becoming unavailable. We discuss the relationship to the University Timetabling Problem, and propose and compare five approaches to fixing an altered timetable: *Greedy Fix*, *Genetic Fix*, *Memetic Fix*, *Genetic Restart*, and *Memetic Restart*.

The results show that although *Memetic Restart* produces good results, the three *Fix* methods displace less events from the original timetable.

Acknowledgments

This thesis would not have been possible without the help and support of the following people:

- My supervisors: Dr. Hannes Herrmann and Dr. David Cooper for their support and advice throughout the process of writing this thesis.
- My parents, siblings, and extended family and friends who offered unconditional love, support, and advice throughout my University studies and the work on this thesis.
- My thesis examiners for their valuable feedback.

Chapter 1

Introduction

Scheduling a timetable is a complicated task with many issues that must be considered when allocating events. Often there are many constraining factors that contribute to whether an allocation is good or bad, and it can be very difficult to get right without a great deal of effort. With the advent of the computer, it is of little surprise that timetable scheduling was a tempting choice for automation. In our research, we found papers being published on the matter as far back as 1969, with indications that earlier papers exist (Lawrie, 1969).

The Timetabling problem has been a popular topic amongst researchers in the Computing field, spawning at least three International Timetabling Competitions (ITC2002, ITC2007 (McCollum et al., 2010), and ITC2011 (Post, Di Gaspero, Kingston, McCollum, & Schaerf, 2016). Each of these competitions deals with a different aspect of timetabling. The First and Second Timetabling Competitions both focused on the University Timetabling Problem specifically, whereas the Third International Timetabling Competition focused on High School Timetabling, a similar issue, but with slightly different concerns to take into account.

The University Timetabling Problem was divided into three 'tracks' for the Second International Timetabling Competition(McCollum et al., 2010): *Examination Timetabling*, *Curriculum-Based Timetabling*, and *Post-Enrollment Timetabling*. Each of these tracks seeks to automate timetabling scheduling, however each has different requirements based on the input and expected output.

Examination Timetabling handles scheduling examinations for various courses. This track differs from the other two in a number of ways, amongst which is that multiple exams may be scheduled in the same allocation slot, provided there is enough room to house it.

Curriculum-Based Timetabling groups events into 'curriculums', with a constraint specifying that events in the same curriculum should not be allocated during the same timeslot if at all possible.

Post-Enrollment Timetabling attempts to allocate according to individual student's availabilities, and is thus the most complicated of the three tracks.

Whilst the University Timetabling Problem can be divided into these three tracks, one should note that research on one track can often still apply to the other tracks, as they are variants of the same problem. Specific concerns do vary, but many concepts remain the same.

The track we have selected to investigate is the *Curriculum-Based Timetabling* track. We have selected this track to research, as that is the track most similar to the allocation strategy currently being used at Curtin University.

The area of the University Timetabling Problem of specific interest to this thesis is the issue of fixing a timetable that has been broken, for one reason or another. This may be because of a desire to ban a specific timeslot after the initial timetable has been constructed, or it may be because a room has become unavailable for use in the middle of semester. In these situations, it would prove useful to have an automated way of fixing an existing timetable, and that makes worth investigating. We refer to this problem as the University Timetable Alteration Problem, and we will discuss it in greater detail in sections 2.5 and 3.2.

The objectives of this thesis are the following:

- To propose several approaches to solve the University Timetable Alteration.
- To compare and contrast the effectiveness of the proposed approaches.

In Chapter 2, we discuss the various approaches to the University Timetabling Problem in general as found in literature.

In Chapter 3, we define the University Timetabling Problem and the University Timetable Alteration Problem and discuss the various concerns that apply to each of the problems.

We then move on to a discussion of the solutions to the University Timetable Alteration Problem proposed by this thesis in Chapter 4.

Experiments run using the proposed algorithms are run, and the results are discussed in Chapter 5.

Finally, we conclude the thesis and discuss future work in Chapter 6.

Chapter 2

Literature Review

2.1 Introduction

The University Timetabling Problem has been proven to be an NP-Complete problem(Cooper & Kingston, 1995). This puts it in the company of other NP-Complete problems such as the Graph Coloring problem (Sabar, Ayob, Qu, & Kendall, 2012). As a member of the set of NP-Complete problems, the University Timetabling problem is known to scale extremely poorly. Solutions to NP-Complete problems may take an unfeasible amount of time to solve. As such, heuristic methods are often investigated in order to find a solution that is of acceptable quality.

When considering possible solutions for the timetabling problem, it is often useful to look at an algorithm in terms of three typical aspects that a typical search algorithm must handle: *Initialization*, *Intensification* or local search, and *Exploration* or global search (S. Fong et al., 2015).

Initialization is the stage where the initial state of the algorithm is set up. For algorithms which require an initial population, such as a Genetic algorithm, or any of the various swarm algorithms, this stage will create that initial population, either randomly, or using some heuristic method.

Intensification, or local search, is necessary in order to discover the best solution in the local neighborhood of a solution. If one neglects *intensification*, one risks missing the best solution entirely.

One should not ignore global *Exploration*, however, as without that aspect of the algorithm one risks finding a local optimal value, but missing the global optimum.

2.2 Classical Algorithms

2.2.1 Tabu Search

Tabu Search is a classical neighborhood search algorithm that maintains a list of moves or solutions that have been used recently, in order to avoid revisiting points that have already been analyzed. By using this *tabu list*, the algorithm can avoid being stuck in local minima/maxima. When applied to problems such as the University Timetabling problem, sometimes only partial solutions will be stored in the Tabu List, trading possible loss of similar, but unexplored solutions, for less of a strain on memory. Another possibility is to store moves into the *tabu list* (Amaral & Pais, 2016).

Tabu Search has also been used to great effect in hybrid with other algorithms with several algorithms competing in the Second International Timetabling Competition using it as either the main part of the algorithm, or in conjunction with other algorithms (McCollum et al., 2010).

2.2.2 Simulated Annealing

Simulated Annealing is a classical algorithm that simulates the process of annealing in metallurgy. In the process of annealing, the metal is first heated up to a high temperature, leaving its atoms in a state of high energy. Its atoms then gradually lose kinetic energy, as the metal cools, slowing the movement of the atom down. Simulated Annealing simulates this effect by assigning a 'temperature' to the algorithm, which 'cools' with each iteration of the algorithm. During each iteration of the algorithm, a random move will be generated. If that move is beneficial, it is accepted. If the move is not beneficial, it may still be accepted, depending on the temperature of the algorithm, and the amount it is worse than the current solution. The algorithm is more likely to accept a 'bad' solution, if the temperature is high. This process allows the algorithm to escape local minima, whilst still maintaining the mindset of a greedy algorithm.

This algorithm has seen use to good effect by many researchers in the field of Timetabling as both the main algorithm of the portion (Lewis & Thompson, 2015) (Ceschia, Di Gaspero, & Schaerf, 2012) (Bellio, Ceschia, Di Gaspero, Schaerf, & Urli, 2016).

Out of the five finalists of Track 2 of the Second International Timetabling Competition (ITC2007), two of the algorithms made use of Simulated Annealing in some fashion (McCollum et al., 2010).

2.2.3 Genetic and Memetic Algorithms

The Genetic Algorithm is a much-loved classical evolutionary search algorithm that has seen application in many fields, including the University

Timetabling Problem. The algorithm works by gradually changing a population of 'chromosomes' (candidate solutions) over many generations through a number of operators known as selection, crossover, and mutation operators.

A selection operator decides which chromosomes are selected for breeding. There are several ways that this can be done, including random selection, tournament selection, and roulette wheel selection. In random selection, one simply selects the chosen number of solutions at random. Tournament selection, improves on this by choosing those randomly selected candidates that improve upon the best found selection.

The crossover operator is the "breeding" stage of a genetic algorithm. Parent chromosomes are selected with some selection operator, and parts of the parents are combined to create some offspring for the next generation. Well-known crossover operators include one-point, and two-point crossover, amongst others.

One-point crossover works by one position in the parent chromosomes and creating two new chromosomes by mixing one side of one chromosome with the other side of the other chromosome. This point may be chosen at random, or it may be the median. Two-point crossover works in a similar fashion, but there are instead two points that are used to mix the chromosomes. Traditional crossover operators such as one-point or two-point crossover will often produce infeasible solutions for the University Timetabling Problem. As such, it may be desirable to produce a crossover operator that will produce feasible offspring (Thepphakorn, Pongcharoen, & Hicks, 2015a).

The mutation operator is in place to prevent the well-known problem of being "stuck" in local optima. This is sometimes implemented by 'flipping' one or more bits in the chromosome, but other methods exist as well, such as Modified Regeneration Selection (Thepphakorn, Pongcharoen, & Hicks, 2015b).

Once these operations are complete, some of the generated solutions are passed on to the next generation, to go through the process again. A process known as *elitist selection* is often put in place to ensure that the best solutions of a generation are always present in the next generation, amongst other generated chromosomes.

A problem with the genetic operators is that they can sometimes leave a solution in an invalid state. When applying a crossover operator to a candidate timetable 'chromosome', some of the 'genes', or course-room-id tuples, can go missing, resulting in double allocations of one event, and missing events in the solution, in spite of the effect that the initial population of candidate solutions may have been complete. In order to combat this property of crossover and mutation operators, Thepphakorn et al. created their own crossover and mutation operators (Thepphakorn et al., 2015b), that ensured no infeasible solutions made their way into the population. Previous approaches used a *repair* process to 'fix' infeasible solutions. (Alves, Oliveira, & Neto, 2015) mention other operators that produce feasible solutions.

Another important consideration in genetic operations is the representation of the chromosomes. Traditionally, chromosomes are represented as a string of symbols, often in binary, where the operations are performed by flipping individual bits. This works fine in some applications, but doing so in some problems, such as the University Timetabling problem, will result in multiplying the possibility of invalid candidates being generated. As such, some solutions represent their chromosomes as consisting of memes/genes that consist of tuples of the room, day, and timeslot (Thepphakorn et al., 2015b).

Sometimes, a genetic algorithm may make use some local search algorithm to try and improve a chromosome after it's been generated. Such algorithms are called *Memetic Algorithms*, and their chromosomes are referred to as *memes*.

Genetic and memetic algorithms have proven to still be a feasible algorithm to solve the University Timetabling problem (Alves et al., 2015).

2.3 Nature-Inspired Swarm Algorithms

Nature inspired algorithms have long been of great interest to researchers, with much research and many algorithms being proposed in the area. Various approaches have been proposed, and we shall discuss them in the following sections.

In their paper *'Debunking the designs of contemporary nature-inspired computing algorithms: from moving particles to roaming elephants'* (S. Fong et al., 2015), Fong, Wong, and Pichappan caution researchers about the tendency to claim their proposed nature-inspired method as 'novel', when in fact, it is merely a 'Makara Dragon', a mythical beast that has features of various other animals. By mixing and matching various parts of other algorithms, or by adding a small variation developed ourselves, we can alter the performance of the algorithm in some way, very likely achieving some measure of performance gain in some conditions. They show, through experimentation, that sometimes, in practice the more simple meta-heuristics are on par with the more novel approaches. It is thus advisable to temper one's excitement over results with a healthy dose of skepticism.

However, many of these algorithms have proven useful, and much research continues to be done in the area.

2.3.1 Particle Swarm Optimization (PSO)

Particle Swarm Optimization is a swarm-based algorithm that consists of many individual *particles* that each keep track of their own position, personal best solution, and current velocity. On a global level, the algorithm keeps track of the current best solution found. Each *particle* updates its velocity based on the personal best solution and the global best solution, with some randomness thrown into the mix. This results in each *particle* moving individually, but often in the same general direction.

Particle Swarm Optimization has been praised for being simple and straightforward to implement, having fast convergence speed, fewer parameters to set, and needing little memory for its computation (Chen & Shih, 2013) (Larabi Marie-Sainte, 2015), but concern has been expressed over the possibility of getting stuck in local extremes and not reaching the global optimum in some cases (Larabi Marie-Sainte, 2015).

Whilst the Particle Swarm Optimization Algorithm is often considered a benchmark with which to compare newer approaches, it has been reported that it can hold its own against other meta-heuristics if given the right parameters and the right problem instance to solve (S. Fong et al., 2015).

2.3.2 Artificial Bee Colony (ABC)

Artificial Bee Colony is a swarm algorithm which takes the foraging patterns of bees as inspiration for a search algorithm. It was originally proposed by Karaboga for numerical optimization in 2005 (Karaboga, 2005), but has since been applied to constrained problems, amongst them the University Timetabling Problem.

Swarm agents are divided into three types of "bees": *employed bees*, *onlooker bees*, and *scout bees*, the last two often being labeled together as *unemployed bees*. *Employed bees* search for new food sources, or candidate solutions in the vicinity of sources that are already known, and collect information on its nectar, or fitness, level. Once they find these new food sources, they bring the information back to the *onlooker bees*, who picks solutions to keep for the next iteration using a selection operator. *Scout bees* then go searching for new food sources, to replace the food sources that were not selected by the *onlooker bees*.

Issues have been found with the performance and convergence speed of the basic ABC algorithm, however. The choice of neighborhood search by employed bees has been criticized for slowing down the convergence of the algorithm, and the onlooker bees' use of roulette selection has been accused of unbalancing the exploitation and exploration (C. W. Fong, Asmuni, & McCollum, 2015). Karaboga and Basturk suggested using tournament selection in the *onlooker phase*, when dealing with constrained problems (Karaboga & Basturk, 2007). In order to provide balance to the ABC, Fong et al. proposed a variant of ABC called the *Nelder-Mead Great Deluge Artificial Bee Colony' algorithm (NMGD-ABC)*, which makes use of the Particle Swarm Optimization global best model for *employed bees*, in an attempt to improve convergence speed, and uses the NMGD algorithm in the *onlooker bee* to solve the perceived exploitation issues (C. W. Fong et al., 2015). Ghasemi, et al. investigated using genetic grouping, in order to improve the *employed bees* phase, with promising results (Ghasemi, Moradi, & Fathi, 2015). Hill Climbing Optimization has also been tried in this step (Bolaji, Khader, Al-Betar, & Awadallah, 2014), with the authors suggesting trying out other algorithms in order to increase performance.

2.3.3 Ant Colony Optimization (ACO)

Ant Colony Optimization simulates the way that ants forage for food. As ants move towards their goal, they leave pheromones on the route they take towards that goal. Pheromones naturally evaporate over time, reducing the influence of abandoned routes. Shorter routes are traveled faster, allowing more pheromones to naturally accumulate on the shorter routes. As more and more pheromones are left upon the route, more ants tend to follow the route with more pheromones, until the whole colony eventually follows the optimal route (Dorigo, Birattari, & Stutzle, 2006).

The Ant Colony Optimization method is often explained as the following simple steps per algorithm iteration (Dorigo et al., 2006) (Nothegger, Mayer, Chwatal, & Raidl, 2012):

1. Ants construct solutions, using a probabilistic method based on pheromone levels to select solution components.
2. *(optionally)* A local search is run on solutions in order to improve them.
3. The global pheromones for each step of the route are updated.

One issue that must be addressed when adapting ACO to the University Timetabling Problem, is how to adapt the concept of pheromones to the problem. In the traveling salesman problem the pheromone model is fairly straightforward, one can simply map individual nodes or the connections between nodes as the components of the solution, each component having it's own pheromone level. The University Timetabling Problem, however, is a little more complicated. The individual components of a solution are the *event-timeslot-room* tuples that make up a complete candidate solution. Instead of keeping track of each individual building block, of which there can be many convolutions, or only focusing on the *events-timeslot* tuples as they had seen in other literature, Nothegger et al. kept track of two pheromone matrices: *event-to-timeslot*, and *event-to-room*. (Nothegger et al., 2012). This makes the pheromone data structure easier to manage than a structure containing the three-value tuples, whilst ensuring more information than simply ignoring the pheromones placed on event-room tuples.

Blum et al. contend that *second order deception* can be a major issue with Ant Colony Optimization. *Second order deception* occurs when an algorithm is not a *local optimizer*, or the average solution quality is not guaranteed to increase per iteration (Blum & Dorigo, 2004). Care must be taken when developing an Ant Colony Algorithm that this does not become a problem.

In the Second International Timetabling Competition (ITC2007), an Ant Colony Optimization Algorithm, used in conjunction with a local improvement search routine, did well in Track 2 in several test instances known to be highly constrained instances with a low number of rooms (McCollum et al., 2010). In all of those instances the algorithm achieved the best results.

ACO has also lends itself to parallelization, which has been used to some degree of success (Ugat, Montemayor, Manlimos, & Dinawanao, 2012).

2.4 Hyper-Heuristics

Hyper-Heuristics have been divided into two groups (Pillay, 2016):

1. *constructive* hyper-heuristics

2. *selection perturbative* hyper-heuristics

Constructive hyper-heuristics generate solutions, whereas *selection perturba-*

tive hyper-heuristics improve solutions that have already been created, either by *constructive* hyper-heuristics, or some other method (Pillay, 2016).

Iterated Local Search with an *'add-delete hyper-heuristic'* (Soria-Alcaraz, zcan, Swan, Kendall, & Carpio, 2016) is an algorithm that incorporates a *selection perturbative* hyper-heuristic approach to the University Timetabling problem. During the *Improvement Stage*, a list of add and remove operations is generated, with each operation consisting of the add/remove instruction, and the id of the event on which the operation is being performed. When an event is 'deleted', it is merely put back on the list of unscheduled events, to be scheduled again. When an event is selected to be added to the timetable, a heuristic is chosen at random from a number of heuristics provided. A history of these add/delete lists (ADLs) is maintained during the runtime of the algorithm, and success of these lists affects how the new ADLs are generated in the next iteration.

2.4.1 Parameter Tuning

One interesting issue researchers face is attempting to determine the correct parameters to pass into their algorithm. This is often accomplished through trial and error. This issue is worsened by the fact that the best parameters for one particular instance of timetabling data may not be the best fit for another instance. Some research has gone into ways to automate this process.

In their paper *'Feature-based tuning of simulated annealing applied to the curriculum-based course timetabling problem'* (Bellio et al., 2016), Bellio et al. investigated using machine learning techniques to determine the best parameters based on features they extracted from the data for their simulated annealing algorithm. When testing against state-of-the-art algorithms, they found their algorithm matched or out-performed the other algorithms in more that half the test instances. In their second stage of their Course Timetabling algorithm, Lewis and Thompson's algorithm automatically calculated every parameter except the end temperature based on the available running time (Lewis & Thompson, 2015).

In Tabu Search, it is often necessary to tweak the *tabu tenure*, or how long a tabu item remains in the *tabu list*. One method for automating the process of determining *tabu tenure* is by using the *Fuzzy Rule Based System*, introduced and used by Pais and Amaral (Pais & Amaral, 2012) (Amaral & Pais, 2016) when applied to the Exam Timetabling track of the University Timetabling Problem. This process takes note of two values for each exam:

1. The *Frequency* of each exam changing to a timeslot
2. The *Inactivity* of each exam, or the number of iterations its assignment has not changed

Values with low *frequency* and high *inactivity* are assigned a low *tabu tenure*, and values with a high *frequency* and low *inactivity* are assigned a high *tabu tenure*, and remain in the tabu list for a longer period of time.

Ant Colony Optimization too is not free from the trouble of parameter tuning. Ugat et al. describe their experimentation of manually calibrating several parameters for their algorithm, concluding that it is of great importance to select parameters that allow for a balance of exploitation and exploration (Ugat et al., 2012).

2.5 Timetable alteration

Sometimes in the world of University Timetabling, one will find that a timetable that has already been generated needs to be changed. It may be due to a public holiday, or the venue has become unavailable. In situations like this the class may need to be reassigned, possibly for one week, possibly for the remainder of semester. Many of the same factors that one must consider when creating a timetable, must be considered when altering the timetable. To our best knowledge this reassignment is done manually. This process can be tedious.

At Curtin University, we often encounter this issue in timetabling when a public holiday observed by the University occurs during a teaching week. In that situation any classes that occur on that day are canceled, and students must either attend another class, or the class must be rescheduled. Due to the sheer number of classes held at a university on a single day, it has in the past been left up to the lecturers and teachers to try and organize a new time and venue. Research material based on this problem appears to be scarce.

Veentra and Vis tackled a similar problem (Veenstra & Vis, 2016), when they investigated filling in empty time slots in a school timetable. In order to accomplish this goal, they proposed and tested three methods: a *simple-rule-of-thumb*, a *heuristic method*, and an *integer linear model*. They found the integer linear model to produce good results, but it took too long to execute, one test running for twelve hours without results. Out of the other two methods, the heuristic method, which consisted of continually reallocating all classes in a unit whilst locking all other classes, provided a better result, whilst giving a shorter runtime than the integer linear model.

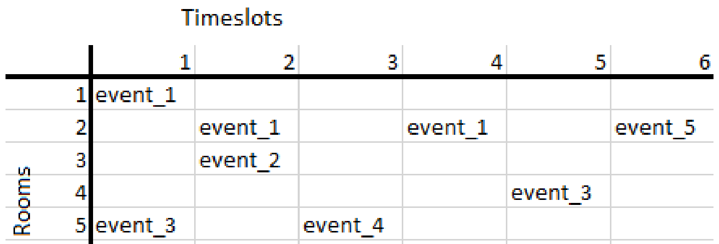
The problem investigated by Veentra and Vis shares a similarity to the University Timetable Alteration Problem, in that it too seeks to minimize disturbances to unaffected events. Their goal, however, was to minimize gaps in the middle of the timetable. The University Timetable Alteration Problem can thus be considered a different problem.

Chapter 3

Problem Description

The University Timetabling problem can be described as the attempt to allocate a list of events to various rooms and timeslots, where a timeslot consists of both a day and a period on that day. These events must be allocated in a manner that minimizes various constraints that have been placed on each event, which we shall discuss in section 3.1.

The specific problem we have chosen to investigate is the University Timetable Alteration problem, or fixing a 'broken' timetable. As this problem is a vari-



Timeslots

Rooms	1	2	3	4	5	6
1	event_1					
2		event_1		event_1		event_5
3		event_2				
4					event_3	
5	event_3		event_4			
6				event_5		

Figure 3.1: An example of a possible timetable with timeslot indices along the x-axis, and room indicies along the y-axis.

ant of the University Timetabling Problem, we shall first discuss the aspects that are the same between the two problems, and then we shall discuss the different considerations of the Alteration problem. To the best of our knowledge, very little, if any, research has been done in this area.

As we noted in Chapter 1, the University Timetabling Problem is divided into three tracks: *Examination Timetabling*, *Curriculum-Based Timetabling*, and *Post-Enrollment Timetabling*. We have selected Track 2, *Curriculum-Based Timetabling*, as that is more applicable to our scenario than *Examination Timetabling*, and is simpler than *Post-Enrollment Timetabling*.

Instead of attempting to minimize schedule conflicts of individual students, *Curriculum-based Timetabling* groups courses into curriculums. These curriculums contain courses whose students are very likely be the same. By making this grouping, we can forgo the time-consuming checks of the schedules of individual students and instead check that courses whose student's are likely to intersect do not clash.

3.1 Constraints

In order to make it easier to differentiate between the goodness of one move from another, we label each constraint into one of two groups: *Hard Constraints*, and *Soft Constraints*. A Hard Constraint is a constraint that should not be broken if at all possible, and represents an impossibility, such as a teacher's inability to be in more than one place at one time. A Soft Constraint represents a preference, such as a University's desire to not have any classes scheduled after a certain hour.

Each event specifies the rooms it requires, which timeslots are unacceptable, the curriculum to which it belongs, and the teacher who teaches it. Each room specifies its max capacity. This information is used to check whether the constraints are broken every time an allocation is made.

These requirements manifest themselves as the five constraints that have been implemented: three hard constraints, and two soft constraints.

Hard Constraints:

- Room Constraint
- Timeslot Constraint
- Teacher Constraint

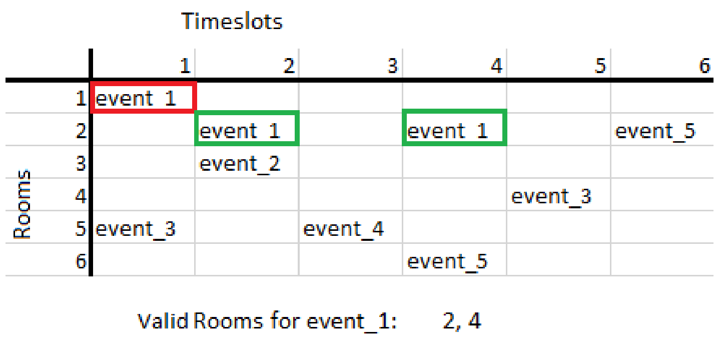
Soft Constraints:

- Room Capacity Constraint
- Curriculum Constraint

These constraints come from the Second International Timetabling Competition (McCollum et al., 2010), and the test datasets provided by them for the competition. There are more constraints described there, such as the Room Stability constraint, which specifies that all lectures of a course should be allocated in the same room, if possible. These constraints were not implemented for this thesis due to time constraints.

3.1.1 Room Constriant

A Room Constraint is violated if an event is allocated in a room that was not in the list of valid rooms. In Figure 3.2, we can see an example of a room constraint being violated. In the figure, we show that rooms 2 and 4


Timeslots
Rooms

Rooms \ Timeslots	1	2	3	4	5	6
1	event_1					
2		event_1		event_1		event_5
3		event_2				
4					event_3	
5	event_3		event_4			
6				event_5		

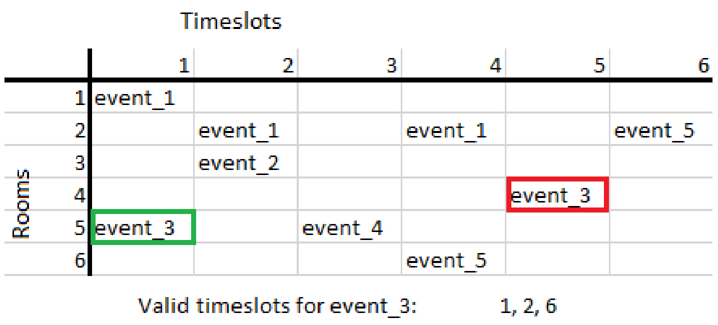
Valid Rooms for event_1: 2, 4

Figure 3.2: An example of a Room Constraint Violation.

were specified as valid rooms for *event_1*. In the timetable, *event_1* has been allocated in rooms 1 and 2. The allocations in room 2, outlined in green, are consistent with the room constraints specified for this event. The allocation in room 1, outlined in red, is not. This would result in one room constraint being reported for *event_1* in this timetable.

3.1.2 Timeslot Constraint

A violation of a Timeslot Constraint occurs when an event is allocated in a timeslot that was not in the list of preferred timeslots for that event. For example, in Figure 3.3 the list of valid timeslot indices for *event_3*, consists of indicies 1, 2, and 6. In the timetable shown, *event_3* has two allocations, one of which satisfies the Timeslot constraint, and the other of which violates the constraint. The allocation in timelslot 1, outlined in green, has been allocated in one of the valid timeslots, and thus satisfies the constraint. The second allocation of *event_3*, in timeslot 5, makes an allocation in a timeslot not contained in the list of valid timeslots, and thus results in a violation of the timeslot constraint.

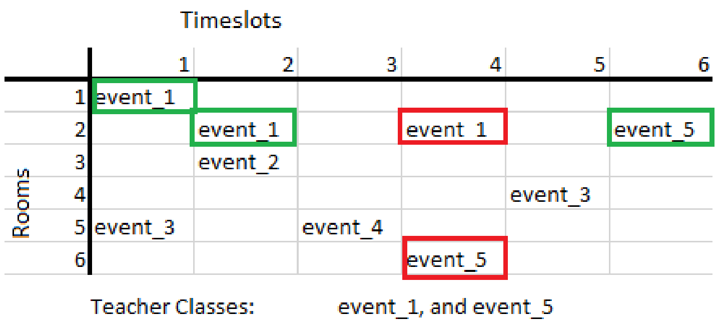


Timeslots

Rooms	1	2	3	4	5	6
1	event_1					
2		event_1		event_1		event_5
3		event_2				
4					event_3	
5	event_3		event_4			
6				event_5		

Valid timeslots for event_3: 1, 2, 6

Figure 3.3: An example of a Timeslot Constraint Violation.



Timeslots

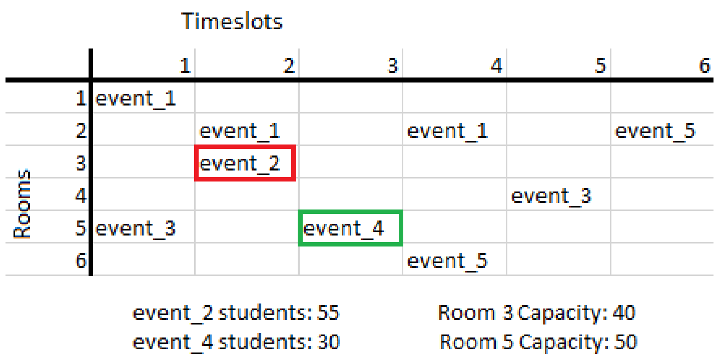
Rooms	1	2	3	4	5	6
1	event_1					
2		event_1		event_1		event_5
3		event_2				
4					event_3	
5	event_3		event_4			
6				event_5		

Teacher Classes: event_1, and event_5

Figure 3.4: An example of a Teacher Constraint Violation.

3.1.3 Teacher Constraint

A Teacher Constraint is violated if a teacher is required to be in more than one place on the same timeslot. Figure 3.4 illustrates one such violation. If a certain teacher is teaching *event_1* and *event_5*, these events must not be allocated during the same timeslot. This occurs in timeslot 4, where the two offending events are outlined in red. There are, however, several events that are allocated consistently with the teacher constraint, in timeslots 1, 2, and 6.



Timeslots

Rooms	1	2	3	4	5	6
1	event_1					
2		event_1		event_1		event_5
3		event_2				
4					event_3	
5	event_3		event_4			
6				event_5		

event_2 students: 55
event_4 students: 30

Room 3 Capacity: 40
Room 5 Capacity: 50

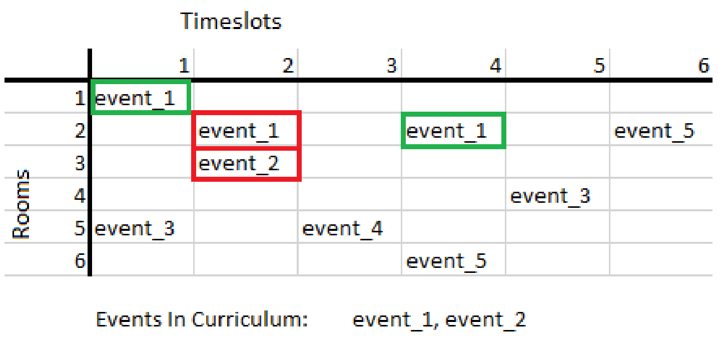
Figure 3.5: An example of a Room Capacity Constraint Violation.

3.1.4 Room Capacity Constraint

Room Capacity Constraints are violated when an event is allocated to a room that does not have enough capacity for the number of students enrolled in the event. Say that room 3 has a maximum capacity of 40 students, and room 5 has a maximum capacity of 50. *Event_2* and *event_4* have 55 and 30 students respectively, and we have allocated these events in the aforementioned rooms. *Event_4* fits into room 5's capacity easily, resulting in no constraint violation, but *event_2* exceeds room 3's capacity by 15 students! This situation, shown in Figure 3.5, results in a room capacity constraint violation, as the room capacity constraint requires that events should be allocated in rooms that have the appropriate capacity.

3.1.5 Curriculum Constraint

A Curriculum constraint is violated when multiple events from the same curriculum are allocated during the same timeslot. In the example given in Figure 3.6, *event_1* and *event_2* are both in the same curriculum. *event_1*



Timeslots

Rooms	1	2	3	4	5	6
1	event_1					
2		event_1		event_1		event_5
3		event_2				
4					event_3	
5	event_3		event_4			
6				event_5		

Events In Curriculum: event_1, event_2

Figure 3.6: An Example of a Curriculum Constraint Violation.

has three allocations, in timeslots 1, 2, and 4. *event_2* has one allocation in timeslot 2. This results in a curriculum constraint violation, as students of one curriculum may be required to attend both events at the same time.

3.1.6 Weighted Constraint Violations

Each allocation slot keeps track of each of the constraints, and whether they have been violated. Hard Constraint Violations, and Soft Constraint Violations are reported as V_h and V_s respectively.

These individual V_h, and V_s values are summed together to create V_h, and V_s values for the candidate as a whole, and are used to help determine the quality of the entire candidate, with higher violation values considered worse than lower violation values.

In order to make it easier to compare the quality of one candidate with another, we combine the V_h and V_s values, as well as the number of unallocated events ($N_{unallocated}$), of each candidate into a single value called V_w, or the weighted violations. The formula we developed and used for this calculation is as follows:

$V_w = 100 * N_{unallocated} + 5 * V_h + V_s$

This formula weights the number of unallocated events higher than the other two variables, as unallocated events indicate an incomplete solution, which we wish to avoid having in our population. Hard constraint violations are likewise weighted higher than soft constraint violations, as we wish to favor results with less hard constraint violations in the population.

3.2 The University Timetable Alteration Problem

After a timetable has been generated, there is a possibility that the requirements of the timetable may change when the timetable is already in use. For example, the roof of teaching room may collapse mid-semester, resulting in that room being made unavailable for the rest of the study period, or perhaps a timeslot becomes unavailable for some reason after the initial timetable has been generated. This will likely result in several allocations being made invalid, and being in need of reallocation. We refer to this problem as the *University Timetable Alteration problem.*

The University Timetabling Alteration Problem is, at its heart, the same problem as the University Timetabling Problem. The various constraints described above, such as the Room Capacity Constraint and the Timeslot Constraint still apply, and the objective of the algorithm is still to minimize the number of hard and soft constraint violations whilst trying to ensure that every event finds an appropriate allocation slot.

There is, however, another issue that becomes important when altering a

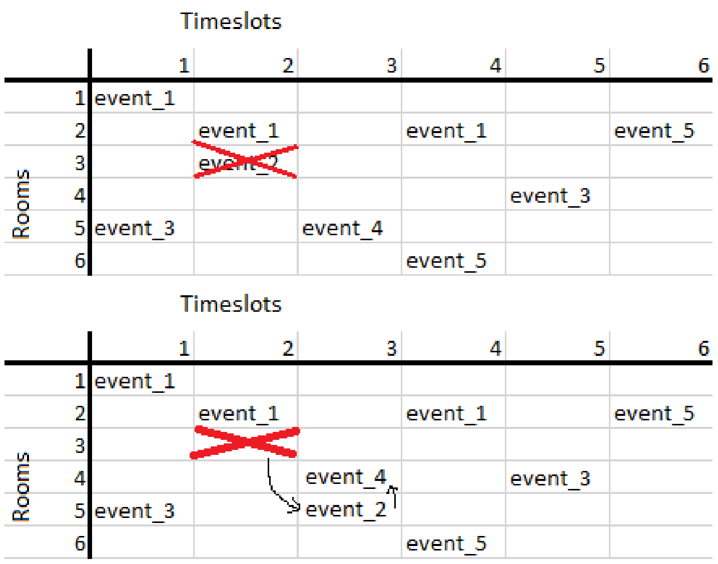


Figure 3.7: Example of an event being displaced by a re-assignment.

pre-existing timetable. If that timetable is already in use, it may be desirable to minimize the number of allocated events moved, or displaced from their original allocation. Students and teachers alike would likely prefer to keep changes to their schedule at a minimum, if it can be helped. Thus, we have an extra goal to minimize: the number of displaced events. We define a *Displaced Event* as any pre-allocated event that was not in a banned allocation slot before the allocation algorithm ran, but was still moved to a different room or timeslot afterward. This occurrence is illustrated in Figure 3.7, where *event_2* has been moved from (room 3, timeslot 2), to (room 5, timeslot 3), displacing *event_4*.

Chapter 4

Our Solution

Five approaches were created and tested as potential solutions to the University Timetable Alteration Problem:

- *Greedy Fix*
- *Genetic Restart*
- *Memetic Restart*
- *Genetic Fix*
- *Memetic Fix*

We shall first give a quick overview of how each of these approaches work, and then we shall discuss the implementation of the algorithms in greater detail.

The simplest approach to used to fix a timetable that has had events banned, is to greedily search for the best placement for an event in the timetable, placing an event in the first suitable allocation slot that is empty. This approach uses the *Greedy-Allocate* algorithm which will be discussed in section 4.2.2. This approach should minimize the number of displaced events, as it will not displace any events unless it cannot find another suitable location for the event it is trying to allocate. This approach is good for minimizing the number of displaced events, but it does little to minimize the number of other constraint violations.

Another approach we have attempted was to ”start over” with the new constraint requirements. When a timeslot or room has been banned globally for a timetable, this also adjusts the set of constraints for each event. Using the adjusted description of the events to be allocated in the timetable, as well as the constraints that apply to that timetable, we can easily run through a genetic or memetic algorithm to create a new timetable that satisfies these constraints. We call this approach either *Genetic Restart* or *Memetic Restart*, depending on whether or not the *memetic step* (described in section 4.2.6) was taken in the algorithm. Whilst the goal of these algorithms is to generate timetables that minimize the weighted violations, they make no effort to generate a timetable similar to the original timetable which was altered by a ban. This approach mirrors applying existing solutions to the University Timetabling Problem to our problem, and serves as a baseline with which to compare our the other three proposed algorithms.

The final two approaches proposed in this thesis are slight variations of *Genetic* and *Memetic Restart*, where the initial population of the algorithm consists of variations of the original timetable. This allows the original timetable to maintain an influence on the population, so that any solution that comes out of a population will be based on the original timetable to some extent. As with the previous two approaches, we refer to these approaches as *Genetic Fix* and *Memetic Fix*, depending on whether the *memetic step* was enabled.

In order to investigate the University Timetable Alteration Problem, we required timetables with which to test our algorithms. As such, timetables were generated using the Memetic algorithm used in the *Memetic Restart* approach. This approach was chosen due to its performance in reducing weighted violations in initial testing.

An implementation of our algorithms have been made available online (Winter, 2016).

4.1 Data Representation

When developing an algorithm, it is important to understand some of how the data is stored.

A fundamental data structure used in the project is the 'Instance'. The 'Instance' contains the description of the problem, as defined by the test file which was loaded. For the purposes of testing we implemented a loader for the '.ectt' file extension for loading the datasets downloaded from the web (Bonutti, Di Gaspero, & Schaerf, 2016).

Amongst the information the Instance contains is a list of every event that needs to be scheduled in a solution, as well as any constraints that apply to that solution. A list of teachers are also maintained, with their schedules being made available to be updated by the allocation algorithm, in order to make checking for violations to the teacher's schedule easier. The number of rooms, days, and timeslots are also specified here, for reference when creating potential solution.

Each 'Event' also has a 'Constraint Level' that estimates how hard it will be to find an appropriate allocation for that event. This estimation is performed by adding the number of valid rooms to the number of banned timeslots for that event. Events with less constraining factors will be easier to allocate, and thus can be left to allocate later in the algorithm.

Each potential solution created by the algorithm is known as a candidate solution, or 'Candidate'. Each candidate solution consists of a two-dimensional container of Allocation units, known as the allocation table. This table is indexed by timeslot index along the width dimension, and by room index along the height dimension. These allocation tables can be visualized in the same manner shown in Figure 3.1.

Each candidate also contains a queue where unallocated events are placed. Items are removed from the queue when they are given a place in the allocation table, and are placed back in the allocation queue if they are unallocated or replaced. In order to make allocation easier, the allocation queue is kept sorted by 'Constraint Level'. This enables our allocation algorithm to greedily select events that are harder to allocate first, leaving the easier allocations for last, decreasing the chance that we will fail to create a satisfactory candidate with all events allocated. This form of greedy heuristic was taken from literature (Sabar, Ayob, Kendall, & Qu, 2012) (Abdelhalim & El Khayat, 2016), and is known as the Largest Degree First Heuristic.

4.2 General Overview

In order to solve the University Timetable Alteration Problem, as described in Chapter 3, several variants of a Genetic or Memetic Algorithm were created, as well as a Greedy Allocation approach. The Greedy Allocation approach, called *Greedy Fix* used the *Greedy-Allocate* algorithm, which is described in section 4.2.2, and is also an important part of the other algorithms.

The other four approaches (*Genetic Fix*, *Memetic Fix*, *Genetic Restart*, and *Memetic Restart*) are variants of one core algorithm, which will be given a general overview in this section, with the various steps being given more detail in the following sections. We refer to each approach separately, as the subtle differences between them have a marked effect on their results, as we shall see in Chapter 5.

The basic framework of the core algorithm is as follows:

```
MEMETIC/GENETIC AGLORITHM
INPUT:
    Instance: a specification of the problem instance.
    G: number of generations
    N: number of candidates to keep per generation
    T: percentage of N candidates to be randomly
    selected for tournament selection
    E: "elite selection"; the percentage of N
    candidates that are selected to go on to the
    next generation.
    M: mutation rate
    doMemetic: Whether or not to run the memetic
    step
Algorithm:
1: Generate initial population of N candidates.
2: WHILE we have not found a perfect solution, or
generation number < G
3: Select the top E candidates from the previous
generation using ELITE-SELECTION and put them into
the new generation.
4: WHILE the number of candidates in this
generation < N
5: using TOURNAMENT-SELECTION, pick two candidates.
6: Perform a CROSSOVER operation with the candidates
from 5.
7: At random (weighted by M), MUTATE the resulting
candidates.
8: IF doMemetic, perform the MEMETIC STEP on the
candidates.
9: Add the children to the list of candidates for this
generation.
10: END WHILE(4).
11: Sort the candidates by weighted violations.
12: END WHILE(2).
```

We shall now discuss each of the steps taken in the algorithm.

4.2.1 Generating Initial Population

For any memetic or genetic algorithm, it is important to consider how we generate an initial population of N chromosomes, or candidate solutions. The initial population is vital as it provides the basis on which new candidates are formed. New chromosomes are created using aspects of the chromosomes that are already in the population. As such, the initial population will have an effect on the candidates that are generated from it.

Two methods of generating the initial population were used in the algorithms developed in this thesis. The first approach was to use *Greedy-Allocate* to generate the required number of candidates from scratch. This approach is used in both of the *Restart* methods mentioned above, as well as when we are generating an initial timetable to use in testing. For this approach, called *Generate-Random-Candidates*, we run through the *Greedy-Allocate* algorithm on new empty candidates N times. This approach will generate the required N candidates for the initial population, but has no guarantee that the candidates will be similar to any initial timetable.

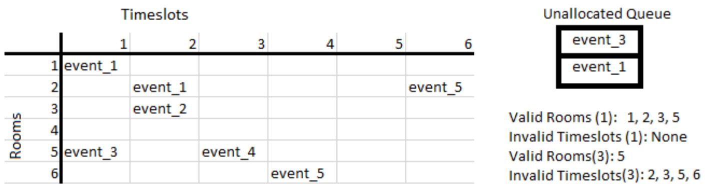
The second method of generating an initial population is to take a previous timetable with a number of events being unallocated from a ban, and allocate the unallocated candidates also using *Greedy-Allocate*. This minor change has the advantage of providing a population that is similar to the original timetable, whilst still providing a population of the required size for the rest of the algorithm.

Providing an initial population that is similar to the original timetable is beneficial, as it will allow the algorithm to draw from candidates that displace fewer events from their initial allocation slot in the original timetable. As was discussed in Section 3.2, reducing the number of displaced events caused by fixing the timetable is important in the University Timetable Alteration problem.

4.2.2 Greedy Allocation

One of the most important parts of the algorithm is the greedy allocation method, since it is used by several of the other parts of the core algorithm as well as in the *Greedy Fix* approach.

In order to understand the Greedy Allocation, we must introduce the concept of the unallocated events queue. Each problem instance has a set number of events, that are determined when the instance is loaded. Whenever an operation is done on a candidate solution to that instance, we must ensure that no events are lost, even if they remain unallocated. In order to facilitate this, all unallocated events are kept into a queue. This queue is kept in order, sorted by its constraint level.



Timeslots

Rooms	1	2	3	4	5	6
1	event_1					
2		event_1				event_5
3		event_2				
4						
5	event_3		event_4			
6				event_5		

Unallocated Queue

event_3
event_1

Valid Rooms (1): 1, 2, 3, 5
Invalid Timeslots (1): None
Valid Rooms(3): 5
Invalid Timeslots(3): 2, 3, 5, 6

Figure 4.1: Incomplete Timetable with events in the unallocated queue.

An example of an unallocated queue can be seen in Figure 4.1. Two events remain in the unallocated queue, *event_3*, and *event_1*. Of these two events, *event_3* contains more constraints, as it has less rooms that are valid for allocation, and more invalid timeslots. As such, it has been placed ahead of *event_1* in the queue.

When performing greedy allocation, we take each event off the unallocated queue one at a time, and attempt to find an appropriate allocation slot for it. In order to do this effectively, we create a list of each appropriate timeslot and room for the individual event, and explore random combinations of the two until we either find an empty timeslot, or exhaust the possible timeslots.

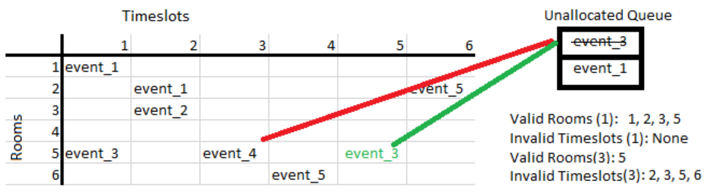


Figure 4.2: Example of item being allocated from the unallocated queue into the timetable.

This process is shown in Figure 4.2. The first allocation slot we select is timeslot 3, and room 5. When we take a look at the allocation slot at these coordinates, we see that the slot is already taken. As such, we attempt to find another appropriate slot. The next slot selected from the list of valid rooms and timeslots for *event_3* is timeslot 5, room 5. This slot is empty, and we place the event there, and remove it from the unallocated queue.

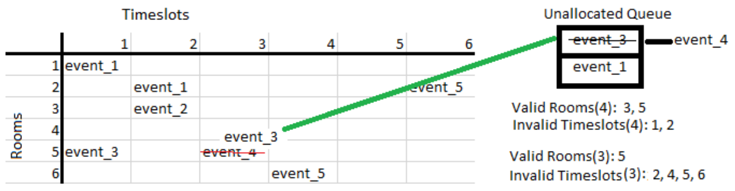


Figure 4.3: Example of Greedy-Allocate solving an event clash.

One could reasonably conceive of a situation where all valid timeslots and rooms are already filled. Such a situation is shown in Figure 4.3. In this timetable, the only valid room for *event_3* is room 5, and the only valid timeslots are timeslots 1 and 3, both of which are filled for room 5. The solution of *Greedy-Allocate* is to place *event_3* into one of those timeslots, and place the displaced event back onto the unallocated queue for reallocation, where it should be able to find another valid allocation slot.

This method of greedily allocating a timetable was adapted from an algorithm found in literature (Bratković, Herman, Omrčen, Čupić, & Jakobović, 2009).

4.2.3 Elite Selection

In order to ensure that our algorithm never goes backwards (a state referred to as *First Order Deception*(Blum & Dorigo, 2004)), we select a percentage of the previous generation to go on automatically to the next generation, which is tunable via a parameter passed into the algorithm. This ensures that the best candidate of each generation will always be at least as good as any candidates of the previous generation. This approach is found in literature (Thepphakorn et al., 2015b).

4.2.4 Crossover Operator

One of the fundamental concepts of a genetic algorithm is the crossover operator. For this operation, we created our own crossover operator, inspired by one-point crossover, as it was simple to implement and produces feasible children.

Selection of the parent chromosomes is important, as we want to select parents with better solutions. To find a parent, we use a tournament selection operator. We randomly choose some percentage of chromosomes, as specified by the tournament rate, and select the best of chromosome out of those chromosomes. This enables us to potentially combine two good solutions into an even better one.

Once two parents are selected, a copy of one of the parents is created to serve as the child. We then select a number of allocation slots from the other parent to copy over to the child. The number we specify is half the number of total events in the Instance. These events are then allocated into the child with any duplicate events being removed. This process is illustrated in Figure 4.4.

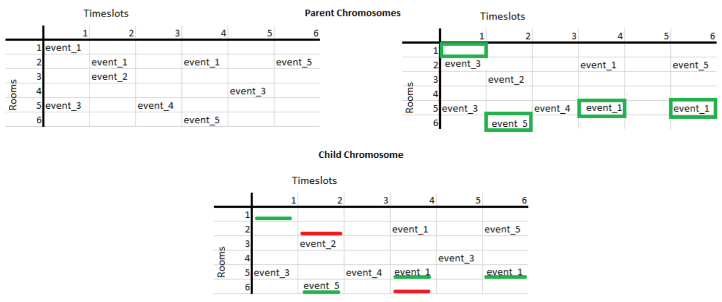


Figure 4.4: An example of the Crossover Operator.

Any events that may have been displaced by the crossover operation are reallocated using *Greedy-Allocate* in order to produce feasible offspring. This may occur if one of the allocation slots taken from one parent was filled by another event in the other parent. This process is called *Repair*, and has been cited as one method of ensuring genetic operations produce feasible offspring(Thepphakorn et al., 2015b).

This process is repeated a second time with the other parent copied instead, so as to insert the same number of events into the next generation, as the parents of the chromosomes will not necessarily go into the next generation, as they may not have been selected by elite selection.

4.2.5 Mutation Operator

A common staple of a genetic or memetic algorithm is a mutation operator, and our algorithm includes one also. A classical mutation operator for a genetic algorithm is to flip some of the bits in the bits in a gene, however, in our circumstance, this may lead to invalid genes. In order to maintain the integrity of every gene, whilst still enabling global exploration, we implemented an operator that dealt with candidates at a higher level than manipulating the bits.

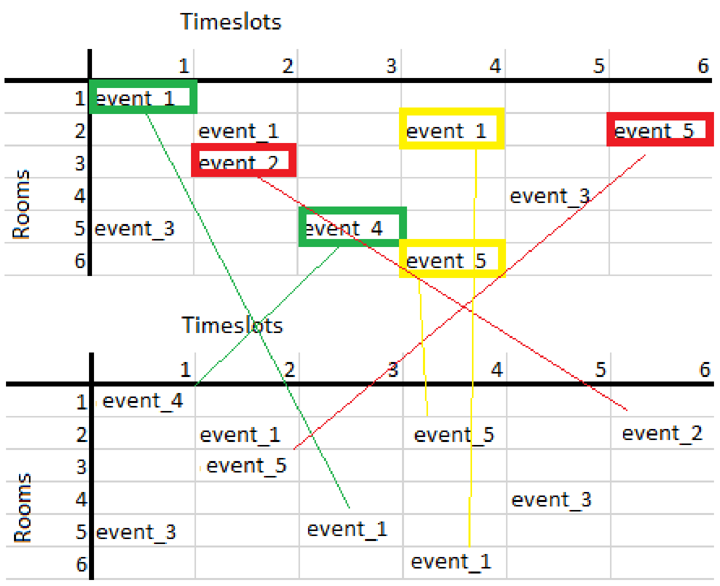


Figure 4.5: An example of the mutation operator.

For each gene randomly selected for mutation, we pick a random number of events to swap, between 20% and 80%. We then choose two allocation slots at random, and swap the event allocations with each other. This maintains the integrity of the timetable, but allows for exploration. This process is illustrated in Figure 4.5.

4.2.6 Memetic Step

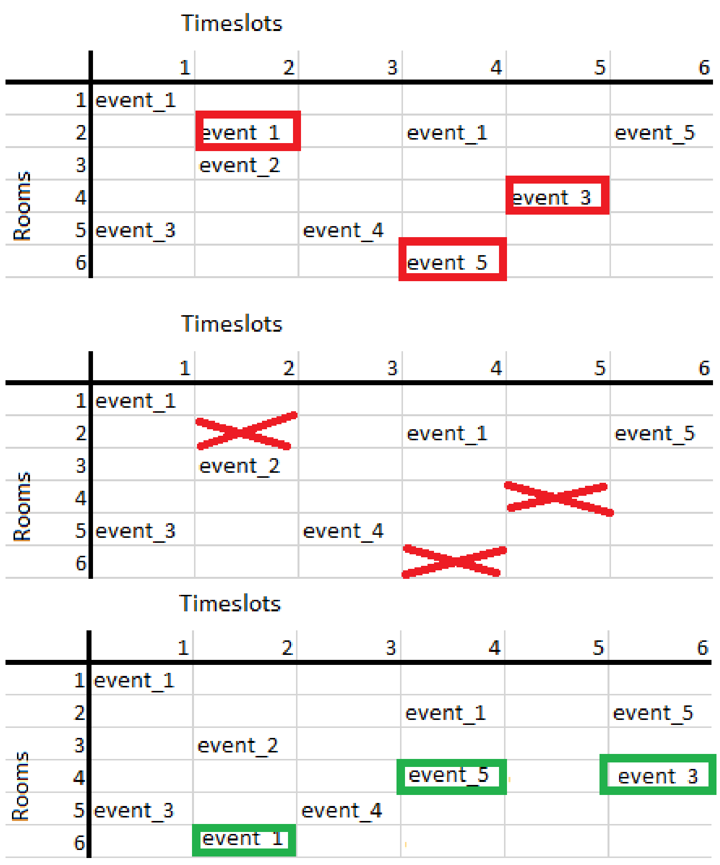


Figure 4.6: An example of the memetic step.

The difference between a genetic and memetic algorithm is the process of attempting to greedily improve each chromosome. We created our own memetic step for use in our memetic algorithms *Memetic Fix* and *Memetic Restart*.

In order to greedily improve each gene of our chromosome, or candidate solution, we first deallocate any allocation slots that are any constraint violation, either hard or soft. In an attempt to improve the genes of our chromosome, we then call *Greedy-Allocate* to greedily find better positions for any deallocated events to allocated. This is illustrated in Figure 4.6.

Whilst this step is beneficial for improving the weighted violations of a candidate solution, it should be noted that this step will result in the displacement of offending allocations by definition.

4.3 Modifying the Timetable

In order to provide test data for our 'timetable fixing' algorithms, we had to first provide a mechanism for choosing parts of the timetable to 'break'. To do this, we implemented several methods of 'banning' timeslots and/or rooms for events.

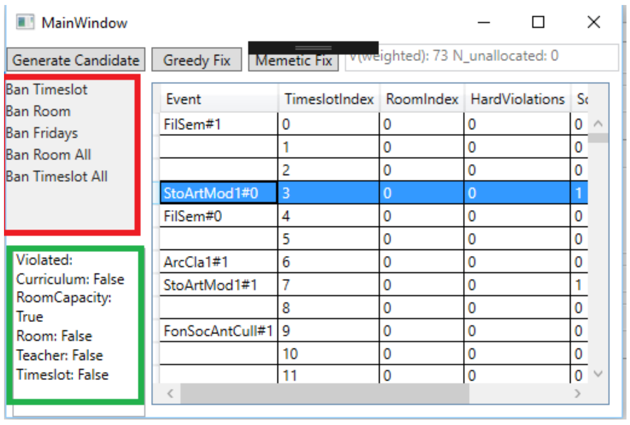
Individual courses can be banned from specific timeslots or rooms by using the methods called *BanTimeslotForCourse* or *BanRoomForCourse* respectively. These methods are examples of simple forms of timetable alteration, but are often simple enough to fix without resorting to more complicated algorithms. In order to provide more complicated tests, three more methods were written and used in experimentation. *BanRoomForAll*, *BanTimeslotForAll*, and *BanDayForAll* are the three methods that were written to facilitate more complicated scenarios on which to test the capabilities of the algorithms. *BanRoomForAll* and *BanTimeslotForAll* ban an entire row or column of the allocation table of a candidate solution, whereas *BanDayForAll* bans several timeslots for all events, effectively banning multiple columns of the candidate's allocation table.

As fixing a timetable that has been modified with a method from the first two ban methods would be a simple fix, we forgo running tests for those methods. We instead focus on attempting to fix alterations made by the last three banning methods, as these provide a more complex problem to solve.

The algorithms for the last three banning algorithms have been provided in the appendix.

4.4 GUI interface and Constraint API

In order to visualize the results, and to get information about which constraints were being violated, a simple graphical user interface was created. Using this interface, we were able to see which timeslot-room tuples were violating constraints, which events were allocated there, and which constraints those tuples were violating.



V(weighted): 73 N_unallocated: 0

Event	TimeslotIndex	RoomIndex	HardViolations	Sc
FilSem#1	0	0	0	0
	1	0	0	0
	2	0	0	0
StoArtMod1#0	3	0	0	1
FilSem#0	4	0	0	0
	5	0	0	0
ArcCla1#1	6	0	0	0
StoArtMod1#1	7	0	0	1
	8	0	0	0
FonSocAntCull#1	9	0	0	0
	10	0	0	0
	11	0	0	0

Violated:
Curriculum: False
RoomCapacity: True
Room: False
Teacher: False
Timeslot: False

Figure 4.7: Screenshot of the GUI used in testing with violation detection functionality.

In Figure 4.7, we can see the the graphical interface in action. A timetable solution has been generated, and the allocation slot at timeslot 3 and room 0 has been selected. Looking at the lower left corner, one can see that the event allocated in this timeslot and room is causing a room capacity violation. Above the lower left corner, we can see a menu that will allow us to place a ban on the room or timeslot selected, for either the event that was allocated there, or all events in the timetable. We can then attempt to reallocate or fix the timetable if we so desire.

The user interactions provided by the GUI allow for the user to customize ban methods and at runtime, but also made automating tests difficult. To compensate for this, a simple console program was created specifically to run tests for recording experiment results. This simple console program is described in the methodology section of the next chapter.

Whilst not used in the final experiments, this tool was invaluable to visualize the results of the algorithm, and to help identify issues with the algorithm during implementation.

Chapter 5

Experiment Results

In this section, we shall discuss the methodology we took in performing experiments for both generating University Timetables, and for fixing timetables that have been 'broken' by the various ban methods we implemented. We shall then show and discuss the results of our experiments.

5.1 Methodology

All experiments were run on a desktop computer of the following specifications:

- Operating System: Windows 10 Home (64-bit)
- Processor: Intel®Core^{TM} i5-3570 CPU @ 3.40 Ghz
- Ram: 8 gigabytes

In order to run each experiment, a small test program was written. This test program would allow multiple values to be specified for several of the parameters, allowing different combinations of different values for each parameter to be tested. We could then run the tests for a couple of hours, and save the results to a csv file. These results were then put into Microsoft Excel ®spreadsheets in order to aid in analyzing the data and to generate graphs. These documents, as well as an example of one of the test scripts can be found on the online repository (Winter, 2016).

Each of the three ban methods described in Section 4.3 were tested separately:

- *BanRoomForAll.*
- *BanTimeslotForAll.*
- *BanDayForAll.*

Only one ban method was selected for testing at a time, as any ban call affects both the individual candidate, as well as the problem instance. If multiple banning methods were called, they would have a cumulative effect on the reallocation process, and thus the later banning tests would be at a greater disadvantage than the former tests.

In order to guarantee that a solution existed for the ban day method, an extra day was added to the initial allocation to ensure that a solution could be found.

Every experiment performed on our algorithms were tested with test data from the test datasets we downloaded from the web (Bonutti et al., 2016). These datasets came from the Second International Timetabling Competition.

Due to time constraints, experiments were run on only the first six test datasets available, with the following parameters for the four algorithms that used paramaters:

- Candidate Size: 150
- Number of Generations: 15,000
- Tournament Selection Rate: 50%
- Elite Percentage: 50%
- Mutation Rate: 25%

The first two parameters were selected to allow the algorithms to run for a non-trivial amount of time, whilst still allowing enough time to run multiple experiments. The latter three were selected from parameters that appeared reasonable from early testing.

In order to attempt to show any possible variance between multiple runs of the algorithm, each algorithm tested was run three times on the same dataset.

For each of the six datasets used in testing, an initial timetable had to be generated. These timetables were generated using the same algorithm used in the Memetic Restart algorithm. This algorithm had shown promise in achieving better results in initial testing, and so was selected to generate the initial timetables. Due to the absence of a serialization implementation for candidate solutions, a new timetable had to be generated for each ban test.

The results of each test were output in a human-readable output, which displayed information such as the values to which the parameters were tuned, the seed provided to the random number generator, the number of constraints violated (both hard and soft, as well as the weighted violations, the number of displaced events, and the time taken. When banning events from rooms, timeslots, or days, the specified room, timeslot, is logged. Each reallocation of modified timetables logs the number of events effected by the ban, and the number of other events that were moved when re-assigning the timetable.

The number of displaced events was not included in the weighted violations calculation, due to some complications in implementation. This will be discussed in Chapter 6.

5.2 Results

After running the tests using the methodology described above, we collected and analyzed the results of the experiments. In this next section we shall show and discuss the results.

5.2.1 Timetable Alteration

When testing performing tests for the University Timetable Alteration problem, we used the first six of the thirty datasets downloaded the aforementioned website. This decision was made due to time constraints.

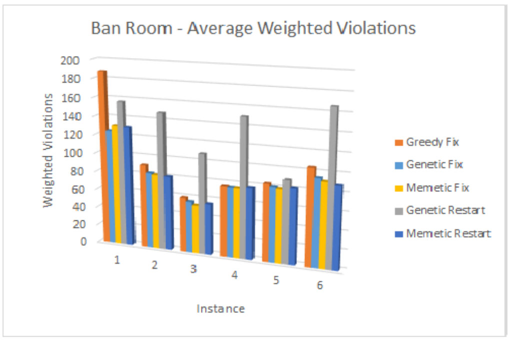
The results of the timetable alteration experiments have been grouped together by the ban method applied to the initial timetable: *BanDay*, *BanTimeslot*, and *BanRoom*. Two measurements of note have been analyzed and their results noted in Figures 5.1 and 5.3. These measurements are the weighted violations (described in section 3.1.6) and the Displaced Events (described in section 3.2). We shall analyze the results of these two graphs in turn.

Graphs in both Figures 5.1 and 5.2 are formatted in the same way. For each of the three measurements taken for each ban method for each algorithm, the three measurements were averaged, and the average was plotted in a clustered column graph. The results for each of the five approaches tested are grouped together by the test instance from which the measurements were taken. The instances are numbered from one to six, and the instance numbers correspond to the equivalent *comp0x.ectt* file downloaded from the web (Bonutti et al., 2016).

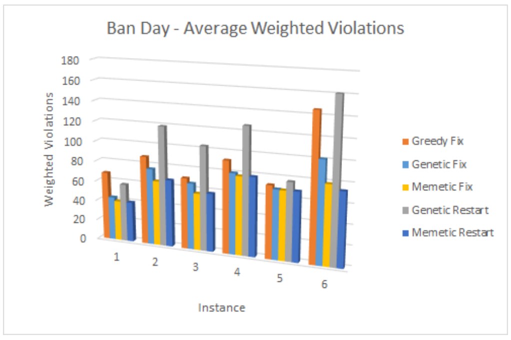
For each of the graphs, the clusters for each instance appear in the same specific order: *Greedy Fix* first, then *Genetic* and *Memetic Fix*, then finally *Genetic* and *Memetic Restart*.

Weighted Violations

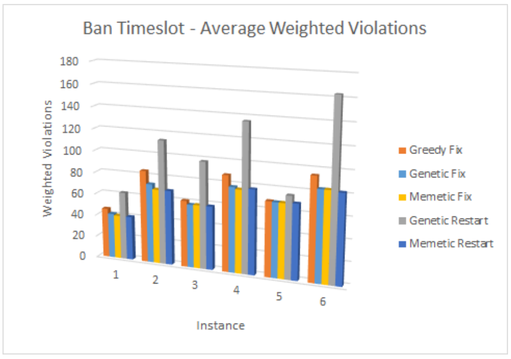
When comparing the results in figure 5.1, we can see that in general the *Genetic Restart* algorithm performs worst of the five algorithms tested in terms of weighted violations, the exception being for Instance 1 in Figures 5.1a and 5.1b. This lack of performance can be attributed to the algorithm restarting from scratch and not having the advantage of the memetic step to speed up the process of finding a more optimal solution. The instances where it outperforms *Greedy Fix* can be attributed to the fact that it does search for more optimal solutions, whereas *Greedy Fix* will give the first solution it finds. If given enough time, *Genetic Restart* should find a lower value for weighted violations than *Greedy Fix*, but *Greedy Fix* has the advantage of starting from a good solution, due to the fact that it builds upon the original solution.



(a)



(b)



(c)

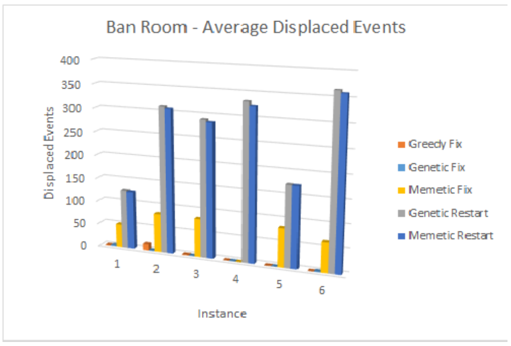
Figure 5.1: Comparison of the weighted violations from the five algorithms tested.

Greedy Fix gives the next best solution in most of the test instances. As noted above, it benefits from its starting point in allowing it to generate a solution that is comparable to to the other solutions at times. Through many of the test instances in the graphs in Figure 5.1, one can see that *Greedy Fix* can generate solutions that are similar in quality to the other solutions. It almost always results in a higher number of violations than the remaining solutions, however, and at times it's solutions can have much higher violations than the other solutions. The major drawback of this solution is that it will only generate one solution. The first set of allocations that *Greedy-Allocate* attempts to make will not necessarily be the best set of allocations, as was discussed in section 4.2.2.

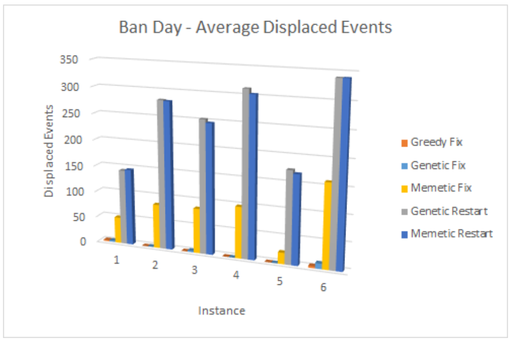
Genetic Fix gives the next highest value for weighted violations, though it is sometimes close to the values given by *Memetic Fix* and *Memetic Restart*, and even beats them in the measurement for Instance 1 in Figure 5.1a. *Genetic Fix*has an advantage over *Genetic Restart* in that it uses a method of generating the initial population from the original solution. This allows the algorithm to start from a better solution, like *Greedy Fix*, but explore other potential solutions. Being a genetic algorithm, it also allows its population to improve over time, allowing for better solutions to be found.

The final two algorithms *Memetic Fix* and *Memetic Restart* both make use of the *memetic step* described in section 4.2.6. This step allows the algorithm to greedily attempt to improve each child solution in each generation. This step appears to have an effect on the quality of the solutions in terms of the number of weighted violations. Both algorithms achieve better results on average than the other three approaches. Both algorithms are at comparable performance, with *Memetic Restart* and *Memetic Fix* each acheiving the best result in terms of weighted violations at various times in the test results. In these two algorithms, we see the method of generating the initial population appears to have little effect on the number of weighted violations when the *memetic step* is involved, and that the *memetic step* appears to be beneficial for minimizing the weighted violations.

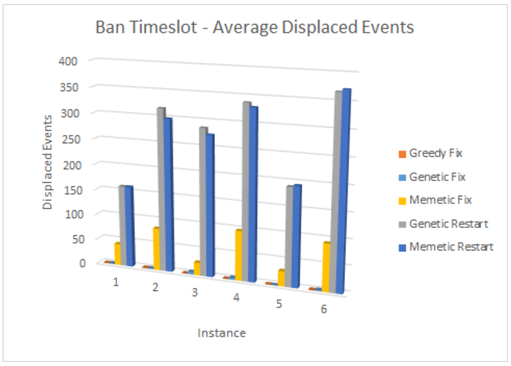
Displaced Events



(a)



(b)



(c)

Figure 5.2: Comparison of the Displaced Events from the five algorithms tested.

If one were to look at the results for the weighted violations alone, one could be forgiven for deciding that *Greedy Fix*, *Genetic Fix*, and *Memetic Fix* hold no advantages over *Memetic Restart*, which appears to produce good solutions most of the time. In the University Timetable Alteration problem, the number of Displaced Events becomes important, as was described in section 3.2.

The graphs in Figure 5.2, shows the number of events displaced by each algorithm in the tests conducted. Displaced Events are events that were unaffected by the initial ban that changed the timetable, but were displaced by the algorithm that generated a replacement table.

As expected, *Genetic Restart*, and *Memetic Restart* failed to minimize the number of displaced events, producing solutions that on occasion displaced hundreds of more events than the other algorithms. This is, of course, due to there not being any mechanism in place to facilitate minimizing the number of displaced events for these two approaches.

Memetic Fix is the algorithm with the next highest value for displaced events, though its average number of displaced events are much lower than those of the two *Restart* algorithms. The way the initial population is generated helps ensure that solutions generated will be similar to the original solution, and thus help to steer the candidates in a generation to have less displaced events. The aggressive nature of the memetic step, as described in section 4.2.6, is undoubtedly the cause of much of the displacement, as the algorithm itself calls for the deallocation of many events per generation. If *Memetic Fix* succeeds in improving the weighted violations due to its memetic step, it will result in more events being displaced. There are occasions in the data where *Memetic Fix* achieves no displacements, but if one takes note of how those instances compare with the results for *Genetic Fix*, one will see that *Genetic Fix* achieved the same value. It is thus reasonable to conclude that any advantage *Memetic Fix* gives in terms of minimizing weighted violations, it gives up in causing an increase in Displaced Events.

The remaining two algorithms, *Greedy Fix* and *Genetic Fix*, both perform favorably in the number of events displaced by its operation, each only occasionally straying from the lowest number of displaced events.

Greedy Fix achieves its low displacement value as it should only displace events if there are no other suitable allocation slots to place an event, and it will make no adjustments to the timetable in order to try and improve its number of weighted violations. This results in a minimal number of events displaced by the algorithm.

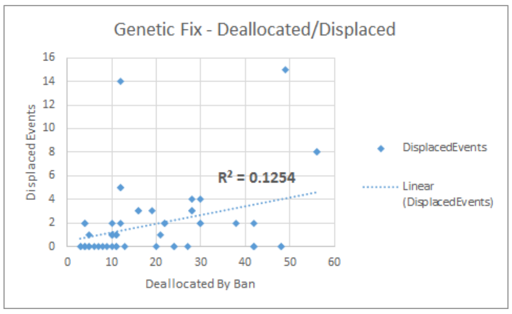
Genetic Fix also achieves a small number of displaced events in every test, only occasionally producing more displaced events. Like *Memetic Fix*, the *Genetic Fix* algorithm takes advantage of an initial population that consists of variants of the original timetable. This allows the algorithm to consider alternate generated variations of timetable, some of which may not require an event to be displaced, which accounts for occasions one can observe in the data where *Genetic Fix* has less displaced Events than *Greedy Fix*. Although *Genetic Fix* does not perform aggressive deallocation of events on the scale of the memetic step, it can deallocate events on occasion in pursuit of a lower number of weighted violations. As there is currently no limitation on deallocating events from the initial timetable, this can result in a higher number of displaced events as can be observed on occasion in Figures 5.2b and 5.2c.

Banned Events vs Displaced Events

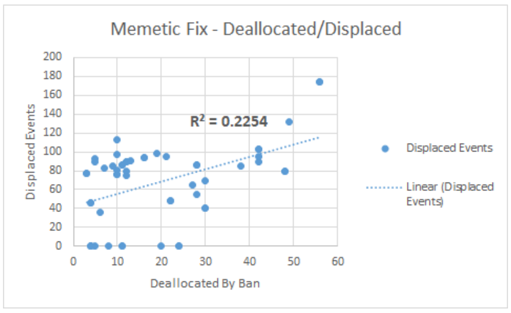
Another comparison we attempted to make was to determine if there was a relationship between the number of banned events, and the number of events displaced by a solution. Data from the experiments run for the Alteration problem were collected and classified by the algorithm used to fix the timetable. For each algorithm, we collected fifty-four datapoints per algorithm, which were plotted in a scatter graphs which can be seen in Figure 5.3. After plotting the points in a graph with the number of Displaced Events plotted along the y-axis and the number of banned events plotted along the x-axis, we attempted to determine a linear relationship amongst the data.

As one can see by observing the graphs, the attempt to find a linear relationship between the two values was not successful. The value for the coefficient of determination (R^2), which is often used in statistics to determine whether or not the data fits a model, is quite low in all five graphs, with the highest value being 0.2254, and the lowest being 0.0035. Typically, the closer the R^2 value is to 1.0, the more confident one can be that a relationship exists between the data. As these values are quite low, we are forced to conclude that no such relationship exists.

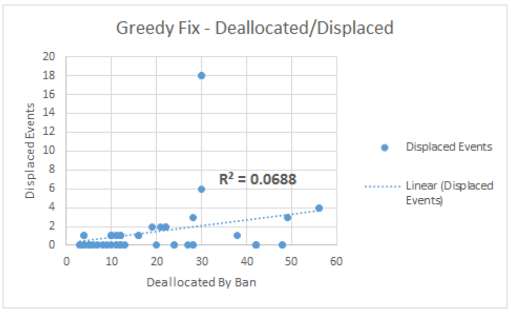
From observation of the data, no direct relationship between the two values is obvious. The choice of algorithm seems to have the strongest effect on the number of displaced events, as can be observed in these graphs, and was discussed in the previous section. We conclude that the number of banned events and the number of events displaced are not directly related.



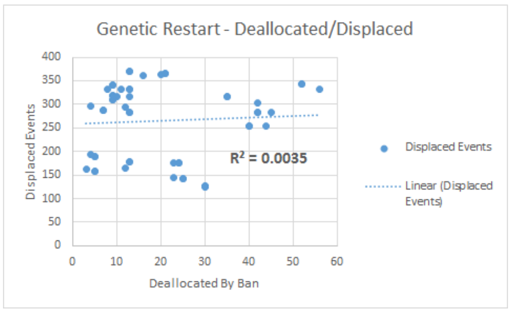
(a)



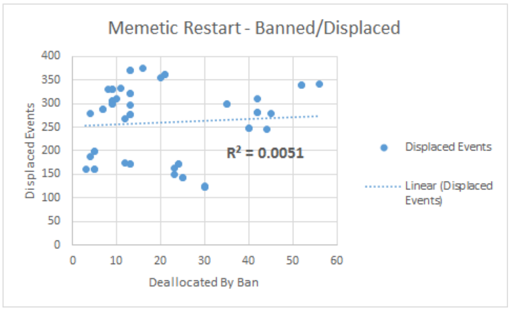
(b)



(c)



(d)



(e)

Figure 5.3: Attempt at finding a linear relationship between Deallocated Events and Displaced Events for each algorithm.

Chapter 6

Conclusion and Future Work

6.1 Summary

This thesis investigated five methods of fixing a university timetable that has been compromised through three different ban methods. These methods were *BanDayForAll*, *BanRoomForAll*, and *BanTimeslotForAll*. The five methods investigated were *Genetic Restart*, *Memetic Restart*, *Greedy Fix*, *Genetic Fix*, and *Memetic Fix*. The first two algorithms are solutions to the University Timetabling problem, applied to University Timetable Alteration, with no thought into the number of displaced events. The final three are approaches designed to attempt to minimize the number of displaced events.

Of these five methods, *Genetic Restart* was found to perform the worst of the five. High values for both weighted violations and Displaced Events showed that the algorithm was unsuitable for use when compared to the other algorithms proposed.

Memetic Restart fared better in terms of weighted violations, in which it often performed best of the five approaches. In spite of its success in weighted violations, it failed to minimize the number of events displaced by the algorithm, resulting in a high number of displaced events.

Memetic Fix took the advantage of the memetic step and used it to achieve a number of weighted violations comparable to *Memetic Restart*, and saw some improvement in the number of displaced events from the two *Restart* methods.

The algorithm that achieved the least number of displaced events on average was *Greedy Fix*, which often achieved the minimal number of displaced events of the five algorithms. Its limitations in exploration limited its effectiveness in minimizing the number of weighted violations, however.

The final algorithm that was investigated was *Genetic Fix*, which achieved a weighted violations value that was often comparable to the performance of *Memetic Fix* and *Memetic Restart*. It also displaces only slightly more events than *Greedy Fix*.

6.2 Future Work

Due to time constraints and other complications, there were various areas of investigation that we would have liked to have taken, but have left for further research in the future.

As was pointed out in Section 5.1, the parameters we used in are experiments were taken from initial testing. As pointed out in literature (Bellio et al., 2016) (Lewis & Thompson, 2015) (Ugat et al., 2012), it is important to ensure that correct parameters are chosen. It would be enlightening to show the effect of changing the parameters of the algorithms, and measuring the effect it has on the results.

Another area to research would be to include the number of displaced events in the weighted violations value for the *Memetic* and *Genetic* algorithms developed for this thesis. Allowing this could possibly allow those algorithms to be more competitive with *Greedy Fix* in terms of the number of displaced events, and may even allow the algorithms to exceed *Greedy Fix* in terms of minimizing the number of displaced events.

References

Abdelhalim, E. A., & El Khayat, G. A. (2016). A utilization-based genetic algorithm for solving the university timetabling problem (uga). *Alexandria Engineering Journal*.

Alves, S. S., Oliveira, S. A., & Neto, A. R. R. (2015). A novel educational timetabling solution through recursive genetic algorithms. In *2015 latin america congress on computational intelligence (la-cci)* (pp. 1–6).

Amaral, P., & Pais, T. C. (2016). Compromise ratio with weighting functions in a tabu search multi-criteria approach to examination timetabling. *Computers & Operations Research*, *72*, 160 - 174. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0305054816300363` doi: http://dx.doi.org/10.1016/j.cor.2016.02.012

Bellio, R., Ceschia, S., Di Gaspero, L., Schaerf, A., & Urli, T. (2016). Feature-based tuning of simulated annealing applied to the curriculum-based course timetabling problem. *Computers & Operations Research*, *65*, 83–92.

Blum, C., & Dorigo, M. (2004). Deception in ant colony optimization. In *Ant colony optimization and swarm intelligence* (pp. 118–129). Springer.

Bolaji, A. L., Khader, A. T., Al-Betar, M. A., & Awadallah, M. A. (2014). University course timetabling using hybridized artificial bee colony with hill climbing optimizer. *Journal of Computational Science*, *5*(5), 809 - 818. Retrieved from `http://www.sciencedirect.com/science/article/pii/S1877750314000441` doi: http://dx.doi.org/10.1016/j.jocs.2014.04.002

Bonutti, A., Di Gaspero, L., & Schaerf, A. (2016). *Curriculum-based course timetabling.* Retrieved 6/12/2016, from `http://tabu.diegm.uniud.it/ctt/index.php?page=instances`

Bratković, Z., Herman, T., Omrčen, V., Čupić, M., & Jakobović, D. (2009). University course timetabling with genetic algorithm: A laboratory excercises case study. In *European conference on evolutionary computation in combinatorial optimization* (pp. 240–251).

Ceschia, S., Di Gaspero, L., & Schaerf, A. (2012). Design, engineering, and experimental analysis of a simulated annealing approach to the post-enrolment course timetabling problem [Journal Article]. *Computers & Operations Research*, *39*(7), 1615-1624.

Chen, R.-M., & Shih, H.-F. (2013). Solving university course timetabling problems using constriction particle swarm optimization with local search. *Algorithms*, *6*(2), 227. Retrieved from http://www.mdpi.com/1999-4893/6/2/227 doi: 10.3390/a6020227

Cooper, T. B., & Kingston, J. H. (1995). The complexity of timetable construction problems. In *International conference on the practice and theory of automated timetabling* (pp. 281–295).

Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. *IEEE computational intelligence magazine*, *1*(4), 28–39.

Fong, C. W., Asmuni, H., & McCollum, B. (2015). A hybrid swarm-based approach to university timetabling [Journal Article]. *IEEE Transactions on Evolutionary Computation*, *19*(6), 870-884. doi: 10.1109/TEVC.2015.2411741

Fong, S., Wang, X., Xu, Q., Wong, R., Fiaidhi, J., & Mohammed, S. (2015). Recent advances in metaheuristic algorithms: Does the makara dragon exist? [Journal Article]. *The Journal of Supercomputing*, 1-23.

Ghasemi, E., Moradi, P., & Fathi, M. (2015). Integrating abc with genetic grouping for university course timetabling problem. In *Computer and knowledge engineering (iccke), 2015 5th international conference on* (pp. 24–29).

Karaboga, D. (2005). *An idea based on honey bee swarm for numerical optimization* (Tech. Rep.). Technical report-tr06, Erciyes university, engineering faculty, computer engineering department.

Karaboga, D., & Basturk, B. (2007). Artificial bee colony (abc) optimization algorithm for solving constrained optimization problems. In *Foundations of fuzzy logic and soft computing* (pp. 789–798). Springer.

Larabi Marie-Sainte, S. (2015). A survey of particle swarm optimization techniques for solving university examination timetabling problem. *Artificial Intelligence Review*, *44*(4), 537–546. Retrieved from `http://dx.doi.org/10.1007/s10462-015-9437-7` doi: 10.1007/s10462-015-9437-7

Lawrie, N. L. (1969). An integer linear programming model of a school timetabling problem [Journal Article]. *The Computer Journal*, *12*(4), 307-316.

Lewis, R., & Thompson, J. (2015). Analysing the effects of solution space connectivity with an effective metaheuristic for the course timetabling problem. *European Journal of Operational Research*, *240*(3), 637–648.

McCollum, B., Schaerf, A., Paechter, B., McMullan, P., Lewis, R., Parkes, A. J., . . . Burke, E. K. (2010). Setting the research agenda in automated timetabling: The second international timetabling competition [Journal Article]. *INFORMS Journal on Computing*, *22*(1), 120-130.

Nothegger, C., Mayer, A., Chwatal, A., & Raidl, G. R. (2012). Solving the post enrolment course timetabling problem by ant colony optimization. *Annals of Operations Research*, *194*(1), 325–339.

Pais, T. C., & Amaral, P. (2012). Managing the tabu list length using a fuzzy inference system: an application to examination timetabling. *Annals of Operations Research*, *194*(1), 341–363.

Pillay, N. (2016). A review of hyper-heuristics for educational timetabling. *Annals of Operations Research*, *239*(1), 3–38. Retrieved from `http://dx.doi.org/10.1007/s10479-014-1688-1` doi: 10.1007/s10479-014-1688-1

Post, G., Di Gaspero, L., Kingston, J. H., McCollum, B., & Schaerf, A. (2016). The third international timetabling competition. *Annals of Operations Research*, *239*(1), 69–75. Retrieved from `http://dx.doi.org/10.1007/s10479-013-1340-5` doi: 10.1007/s10479-013-1340-5

Sabar, N. R., Ayob, M., Kendall, G., & Qu, R. (2012). A honey-bee mating optimization algorithm for educational timetabling problems. *European Journal of Operational Research*, *216*(3), 533–543.

Sabar, N. R., Ayob, M., Qu, R., & Kendall, G. (2012). A graph coloring constructive hyper-heuristic for examination timetabling problems. *Applied Intelligence*, *37*(1), 1–11. Retrieved from `http://dx.doi.org/10.1007/s10489-011-0309-9` doi: 10.1007/s10489-011-0309-9

Soria-Alcaraz, J. A., zcan, E., Swan, J., Kendall, G., & Carpio, M. (2016). Iterated local search using an add and delete hyper-heuristic for university course timetabling [Journal Article]. *Applied Soft Computing*, *40*, 581-593.

Thepphakorn, T., Pongcharoen, P., & Hicks, C. (2015a). Modifying regeneration mutation and hybridising clonal selection for evolutionary algorithms based timetabling tool [Journal Article]. *Mathematical Problems in Engineering*.

Thepphakorn, T., Pongcharoen, P., & Hicks, C. (2015b). Modifying regeneration mutation and hybridising clonal selection for evolutionary algorithms based timetabling tool. *Mathematical Problems in Engineering*.

Ugat, E. B., Montemayor, J. J. M., Manlimos, M. A. N., & Dinawanao, D. D. (2012). Parallel ant colony optimization on the university course-faculty timetabling problem in msu-iit distributed application in erlang/otp. *GSTF Journal on Computing (JoC)*, *1*(4).

Veenstra, M., & Vis, I. F. (2016). School timetabling problem under disturbances. *Computers & Industrial Engineering*, *95*, 175–186.

Winter, Z. (2016). *University timetable alteration repository.* Retrieved from `https://github.com/Plasticcaz/University-Timetable-Alteration`

Appendix A: Pseudo Code

Main Algorithms

```
GreedyAllocatate
INPUT: candidate solution
1. Sort a candidate's unallocated queue for
   Greedy Allocation.
2. While the unallocated queue is NOT empty:
3.  Grab the next unallocated event off the queue.
4.  Initialize/Keep queues of all valid rooms and
    timeslots for the event.
5.  Go through each combination of rooms and
    timeslots in the valid room and valid timeslot
    queues.
6.  For each combination check to see if the
    slot is empty.
7.      If it is, allocate the event there.
8.      If you get to the end of all valid rooms/
        timeslots, pick a random room and timeslot,
        and allocate there, putting any displaced
        events back into the queue.
```

```
GENERATE_RANDOM_CANDIDATES
Input:
    N: the number of candidates to generate
    Instance: a specification of the problem instance
Algorithm:
1: Create a list of size N.
2: WHILE the list is not full
3: Generate a candidate solution of the Instance using
GREEDY-ALLOCATE
4: IF the total unallocated events of the candidate
is zero, add it to the candidate list.
5: END WHILE(2)
6: Sort the candidate list by V_w.
```

```
CROSS Algorithm
Input:
    Child: The clone of parent A that will
    be the child
    Parent: the other parent
    N: the number of allocation slots to
    draw from Parent, usually half the
    number of total events.
Algorithm:
1: FOR 0 to N
2: Select a random timeslot
3: Select a random room
```

```
4: IF an event is in that allocation slot
in the Child, Deallocate that event.
5: Allocate the event from the parent's
allocation slot in the child's allocation
slot.
6: END FOR(1)
7: Call GREEDY-ALLOCATE on child to allocate
any events that have been displaced.
8: Re-Evaluate the child for constraint
violations.
```

```
MUTATE Algorithm
Input:
    Candidate: the candidate to be mutated
Algorithm:
1: Pick N of events between 20\% and 80\%
of the total events.
2: FOR 0 to N
3: Pick two random allocation slots in the
timetable.
4: Swap the events allocated in those allocation
slots.
5: END FOR(2)
6: Re-Evaluate the Candidate for constraint
violations
```

```
MEMETIC STEP Algorithm:
Input:
    Candidate: the candidate to improve
Algorithm:
1: FOR EACH allocation slot in the timetable
2: Deallocate any event with a Weighted
Violation value > 0.
3: END FOR(1)
4: Call GREEDY-ALLOCATE to allocate any displaced
events.
5: Re-Evaluate the Candidate for constraint
violations
```

```
ELITISM-SELECTION Algorithm
Input:
    Candidates: List of candidates to draw from.
    Children: the list in which to insert elite
    candidates
    Percentage: the percentage of candidates to
    take.
Algorithm:
1: Calculate the number of candidates to draw
from the candidate list, N, using the specified
Percentage.
2: Sort the candidates list, with solutions with
the lowest V_w value being at the front.
3: Add the first N candidates from the candidates
list to the children list.
```

Banning Algorithms

```
BAN–TIMESLOT–FOR–ALL Algorithm
Input:
    timeslotIndex: the index of the timeslot to ban.
Algorithm:
1: FOR EACH event in the Instance
2: IF the event timeslot ban list does not contains
this timeslot, insert this timeslot to the list.
3: END FOR(1)
4: Deallocate any events that are allocated on the
given timeslot.
```

```
BAN–DAY–FOR–ALL Algorithm
Input:
    day: day index to ban
Algorithm:
1: FOR EACH period in the periods per day
2: Convert the period and day into a timeslot index
3: FOR EACH event event in the instance
4: IF the event timeslot ban list does not contains
this timeslot, insert this timeslot to the list.
5: END FOR(3)
6: Deallocate any events that are allocated on the
given timeslot.
7: END FOR(1)
```

```
BAN–ROOM–FOR–ALL Algorithm
```

```
Input:
    bannedRoomIndex: The index of the room
    to ban
Algorithm:
1: FOR EACH event in the instance
2: Remove the room index from the list of valid
rooms, or create a list of valid rooms without this
event if no list exists.
3: END FOR(1)
4: Deallocate any events that are allocated in the
given room.
```