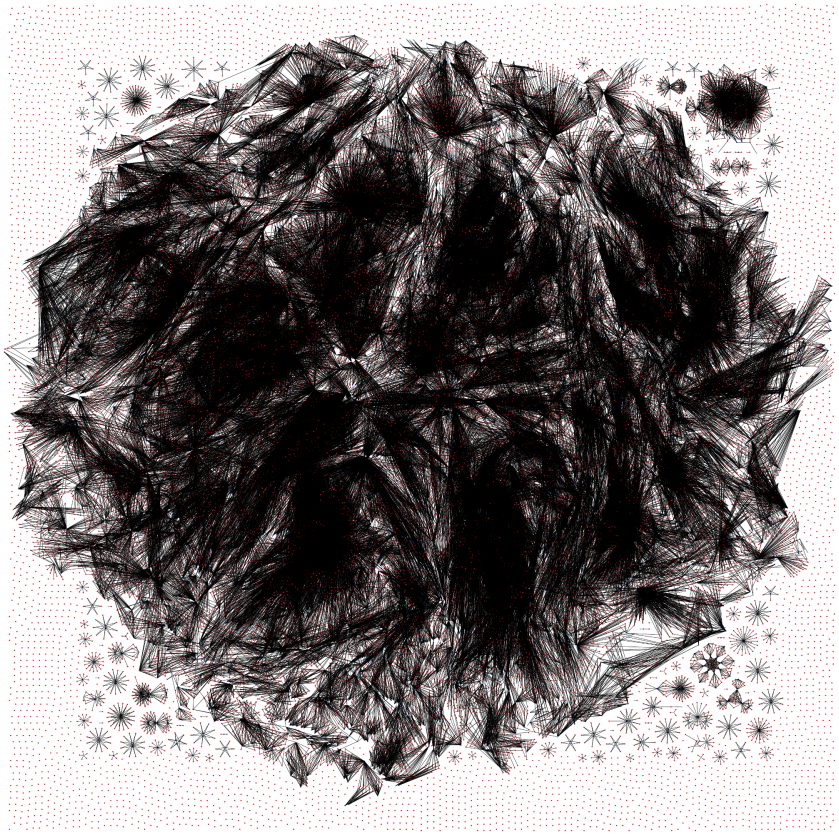
Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam



A Comparison of Heuristic Approaches to Timetable Generation at Science Park

Stephen Swatman
June 7, 2016

Supervisors: Robin de Vries BSc and dr. L. Torenvliet

Abstract

Generating timetables for large institutions is a tightly constrained NP-hard problem. In this thesis, we use heuristic methods to attempt to solve this problem for the *Science Park* campus of the University of Amsterdam which encompasses tens of thousands of activities, students, faculty members and locations. We manage to create timetables that violate none of the implemented constraints for ninety-six percent of the activities using a greedy scheduling heuristic. Using several local search optimisation techniques we further improve the timetable. Significant improvements to the optimisation speed of the local search methods are achieved using stochastic activity selection methods. Complex neighbourhood moves are shown to be ineffective compared to simpler randomisation moves due to the number of constraints. Similarly, basic hill climbing techniques prove more effective than simulated annealing strategies. While we achieve promising results, the timetabling process cannot yet be fully automated. Still, some of the techniques described in this thesis may be applied to improve the speed of the timetabling process and the program developed may serve as a tool to create a starting point for future timetables.

Acknowledgements

I would like to thank my supervisors, Robin de Vries and Leen Torenvliet for their continued support and invaluable help. Their suggestions and critiques on all levels of this thesis were essential to the project.

My thanks also go out to Reinout Verbeek for sharing his vast knowledge about timetabling problems and Monique Laurent for answering my questions about combinatorial optimisations problems.

Finally, my gratitude goes out to my family for putting up with my antics and for their unremitting support and guidance throughout the writing of this thesis as well as through my life in general.

Contents

1 Introduction **7**
- 1.1 Problem definition . . . 8
  - 1.1.1 Minimisation . . . 8
  - 1.1.2 Case-specific details . . . 9
  - 1.1.3 Representing students . . . 10
- 1.2 Definitions . . . 10
- 1.3 Thesis structure . . . 10

2 Methods **11**
- 2.1 Constraints . . . 11
- 2.2 Cost function . . . 11
  - 2.2.1 Idle time constraint . . . 12
  - 2.2.2 Timeslot constraint . . . 12
  - 2.2.3 Violated hard constraints . . . 12

3 Heuristics **15**
- 3.1 Possible reductions . . . 15
  - 3.1.1 Integer linear programming . . . 16
  - 3.1.2 Graph colouring . . . 16
- 3.2 Initial allocation . . . 18
  - 3.2.1 Activity sorting . . . 18
  - 3.2.2 Timeslot and room allocation . . . 19
- 3.3 Local search optimisation . . . 19
  - 3.3.1 Neighbourhood definition . . . 20
  - 3.3.2 Move selection . . . 21
  - 3.3.3 Activity selection . . . 22
  - 3.3.4 Hill climbing . . . 22
  - 3.3.5 Simulated annealing . . . 22
  - 3.3.6 Tabu search . . . 24
- 3.4 GRASP . . . 24

4 Implementation **25**
- 4.1 Data collection . . . 25
  - 4.1.1 Filtering . . . 26
- 4.2 Output . . . 26
- 4.3 Internal representation . . . 26
- 4.4 Parallelism . . . 27
- 4.5 Shortcomings . . . 28
  - 4.5.1 Natural language constraints . . . 28

5 Experiments **31**
- 5.1 Benchmarking . . . 31
  - 5.1.1 Measurements . . . 31

5.2 Evaluating existing timetables . . . 32
5.3 Hard- and software . . . 32

6 Results **33**
6.1 Initial allocation . . . 33
6.2 Optimisation . . . 35
6.2.1 Long-term sustainability . . . 35
6.2.2 Weighted selection parameters . . . 35
6.2.3 Effects of resource type weights . . . 35

7 Discussion **41**
7.1 Initial allocation . . . 41
7.2 Optimisation . . . 41
7.3 Weaknesses . . . 42

8 Conclusion **45**
8.1 Future research . . . 45

A Visualised completed timetable **47**

B Weights of soft constraints **49**

Bibliography **51**

CHAPTER 1

Introduction

More than twenty thousand activities take place on the *Science Park* campus of the University of Amsterdam every year. Due to a growing number of students, creating a timetable for these activities is becoming increasingly difficult. The process must also be repeated each year to account for changes in courses, the number of students and other factors. If we consider that one of the more important properties of a well-chosen timetable is that it not only valid but also an efficient allocation of the students' and faculty members' time, it is perhaps not surprising that it currently takes a small team of three people approximately two months to finalise the timetable each year.

While many commercial programs exist that attempt to solve this problem (henceforth referred to as the *timetable problem* or *timetabling problem*), these programs are hard to generalise in such a way that they can be applied to all institutions. In this thesis we will implement and compare different heuristics for generating timetables in such a way that the software produced is capable of handling the most common constraints as applicable to the Science Park campus.

The problem faced is similar to many other problems such as the *nurse scheduling problem* (NSP) which does not concern itself with students and academic faculty but rather with nurses, presumably in a medical setting. While the thematic differences are irrelevant one important difference between the nurse scheduling problem and our academic timetable problem is that different types of resources are required and available. Often, instances of such problems are partially solved beforehand and can have some allocations predefined. For example, the allocations between activities and students or activities and faculty members may already be complete as is the case at the Science Park campus. Thus, we concern us solely with finding a room and a time for each activity.

Our aim for this thesis is not to completely automate the process of timetable generation but to provide a program that can be used to create a rough draft in which as many of the activities are already planned. Indeed, when accounting for the constraints imposed (many of which are not available beforehand or provided in unstructured human language) such a problem would require a level of flexibility that is far outside the scope of this project. Instead we aim to reduce the amount of time spent scheduling simpler activities, reducing in turn the time taken to complete the entire timetabling process.

Like many problems with direct real-world implications, the timetabling problem has been solved by many people in numerous institutions every year for many years. From these repeated attempts result many different intuitive approaches that, while perhaps not formally defined or proven to be correct, allow good approximations of optimal timetables using relatively simple rules that are easy to understand and implement. This thesis aims to utilise such intuitive methods and expand on them as well as provide a comparison.

1.1 Problem definition

The timetabling problem is characterised by two different types of entities. First, activities such as lectures or labs must be scheduled to take place at certain points in time. Second, resources may be required by activities, though they may only be used by one activity at a time. The three types of resources considered at the Science Park campus are students, faculty members and rooms. Students and faculty members are distinguished from each other mostly on a conceptual basis and are functionally similar. Students and faculty members can have availability rosters indicating when they can and cannot be deployed and may be constrained to other resources to dictate that they cannot be utilised concurrently. At the Science Park campus, students and faculty members are assigned to activities before the scheduling process starts.

Location resources represent rooms and other locations in which activities may take place such as lecture halls. Location nodes are somewhat more complex than student and faculty resources as they not only inherit all the aforementioned constraints from the other types of resources but also have a limited size and a number of suitabilities. The size of a location resource dictates the maximum number of people that can be housed in that location at the same time and this number must always be equal to or greater than the expected number of people attending an activity scheduled in that location. Suitabilities are used to indicate that a location provides certain equipment which may be required by activities. Common examples of suitabilities include the presence of a blackboard, a videoprojector or computers for use by students. A location housing an activity must always provide every suitability required by that activity, though a location may have additional suitabilities that are not required by the activity.

Activities can represent many different events such as lectures, exams or labs. Each activity has a number of resources it requires, one of which should be a room in which the activity takes place. Activities are limited in the number of rooms that can house them due to the aforementioned suitability requirements and activity sizes. They may also be constrained to other activities to indicate that one activity should take place before or after the other or, conversely, that they should take place at exactly the same time albeit in different weeks. Activities that are required to take place at the same time in different weeks are commonly referred to as series of activities and are generally considered to be pleasant as they make the schedules of students and faculty members more predictable. Finally, all activities have a certain length to determine the time at which the activity ends relative to the time it started.

Because the generation of timetables is an NP-hard problem [1], generating an optimal solution for a system as large as an entire university faculty quickly becomes so complicated that it is outside the reach of modern computers. To allow us to complete this process within a feasible amount of time, our goal must not be to find the *optimal* solution to an instance of the problem but instead a solution that is an *approximation* to the optimal solution. In other words, we are tackling a minimisation problem that can be characterised as follows:

Input:	A set of activities to be scheduled each with a list of hard and soft constraints as well as a list of resources, most commonly in the form of rooms, students and faculty.
Constraints:	All activities must be assigned a time and a room, each resource may only be used by *one* activity at a time and the timetable must adhere to a list of other constraints described in Section 2.1.
Cost:	A weighted sum of all broken constraints.
Objective:	Minimisation

1.1.1 Minimisation

A requirement of an optimisation problem such as the one described above is that there needs to be a certain cost function that can be maximised or minimised. In the timetable problem, this cost function is a measure of the quality of the timetable. To compute this measurement of quality, we must first decide which properties of a timetable affect the quality of it. In other

Table 1.1: A count of different entities and relations between entities in the problem instance presented by the Science Park campus.

Entity type	Count
Activity	27162
Resource	3741
Student set	2119
Faculty	1473
Room	149
Predefined allocations	106218
Student set	76582
Faculty	29636
Suitabilities	
Activity requirements	70220
Room suitabilities	2257
Constraints	
Sequencing	37525
Avoid concurrency	5660
Student set	5506
Faculty	164
Room	0

words, we need to define a list of *soft constraints* that, when broken, do not necessarily disqualify the timetable but decrease its fitness score. Instead of a measure of fitness that increases as the timetable achieves a higher quality, we use a measurement of *unfitness* such that a cost of zero implies a perfect timetable and a higher score implies a timetable of lower quality. Violating a soft constraint should thus increase the unfitness score. As this method has a certain point at which the timetable is considered perfect, it provides measurements that are easier to interpret and provide more insight, as well as providing a clear goal to work towards.

To allow the user of the software to gain additional insight into the strengths and weaknesses of a timetable and to allow the user to tailor the timetable to a specific purpose, the cost function should be computed for the three different types of resources available and the cost of a complete timetable should be dependent on a weighted sum of these values. Raising the weight of one of the three types of resources then increases the quality of the timetable for that group but may decrease the quality for other groups. While constructing a timetable for students and faculty members may seems inherently more useful, a timetable optimised for efficient use of rooms may be useful in a setting where, for example, the use of rooms is an important financial consideration.

1.1.2 Case-specific details

Activities at the Science Park campus are scheduled in blocks of fifteen minutes and are scheduled between 8:00 and 23:00. Each day thus consists of sixty separate timeslots and each week consists of five workdays. Each activity then has a length measured in a number of fifteen minute timeslots. Generally, an activity is assumed to last for two hours (eight timeslots) although this can vary greatly. Regarding the size of the problem instance presented by the Science Park campus, Table 1.1 shows the quantities of different entities present in the system as well as the constraints posed between then.

1.1.3 Representing students

While it is possible to treat each student as a separate resource, this quickly becomes computationally expensive due to the aforementioned number of students enrolled at the Science Park campus. In many cases, groups of students also follow the same courses meaning they can be treated as though they were one single resource. The solution currently employed is to group students with similar enrolments together into so-called *student sets*, reducing the number of resources greatly. Creating unified resources from multiple students also has the advantage that it makes large groups easier to schedule by creating multiple smaller groups that can be more easily assigned a room. A downside to this approach which must be considered is that such student sets must be created beforehand which in itself can be a difficult task. Additionally, such an approach makes it more difficult to verify that a student is not involved with two activities at the same time as a student could be a member of multiple independent student sets.

1.2 Definitions

Activity	A lecture or other event requiring faculty members, a room and any number of students.
Resource	Either a set of students, a faculty member or a room.
Student set	An abstract depiction of multiple students that follow the same classes.
Faculty member	Someone who teaches an activity or is otherwise required to supervise it.
Room	A location in which an activity can take place. Often but not necessarily a lecture hall. Has a maximum capacity and a number of attributes such as the presence of a blackboard or computers for use by students.
Day	A number representing a day of the week, usually Monday through Friday although this can be configured arbitrarily to fit any use case.
Timeslot	A certain period of time in a day. Usually each slot occupies between fifteen minutes to an hour and the number of timeslots in a day is inversely related to the size of each slot.

1.3 Thesis structure

Chapter 2 explains in detail the cost function implemented by our minimisation problem as well as the constraints we use to calculate it. Chapter 3 described the different heuristic methods we implement and compare as well as some methods which we do not implement but which have shown some success in the literature. The details of our implementation on a non-theoretical basis are described in Chapter 4, followed by an explanation of the experiments performed in Chapter 5. Finally, we present and discuss our findings in Chapters 6 and 7 and conclude in Chapter 8.

CHAPTER 2

Methods

In the previous chapter, we discussed the importance of a cost function for evaluation the quality of timetables. This chapter will expand on the implementation of this cost function and the twelve constraints implemented in the timetabling program. Some complex constraints will be described in more detail as well as the way in which violated hard constraints are handled.

2.1 Constraints

Ghaemi, Vakili, and Aghagolzadeh [2] and Mushi [3] propose examples of hard and soft constraints. We have selected those constraints that are relevant to the particular case of the Science Park campus and have supplemented these with hard constraints specific to the problem instance. The constraints supported by our implementation are listed in Table 2.1. These constraints are weighed such that a resource being inconvenienced for one hour accounts for one point of unfitness (see Appendix B for more detail). It is worth noting that most of the aforementioned constraints, discounting the trivial constraint C_0 and constraint C_4, affect resources instead of activities. This has both a conceptual benefit as well as a practical benefit. Indeed, faults in university timetables are ultimately experienced by human beings (and to a lesser extent, rooms) as opposed to the activities which are merely abstract concepts. Practically, evaluating constrains from the viewpoint of the resources makes the implementation of the cost function significantly simpler as only resources have to be evaluated. Such an approach would also implicitly weigh violations by the number of resources that experience them, meaning that violations that affect many people are weighed more heavily than violations that only affect small groups of people. This effect is largely negated, however, by the grouping of students as described in Section 1.1.3.

2.2 Cost function

To score the fitness (or rather, unfitness) of a timetable, we must take into account the number of constraints that have been violated. Given a set of faculty members Q, a set of rooms R and a set of students S with respective weights w_Q, w_R and w_S as well as a set of activities A in a timetable, the unfitness of that timetable $C(A, Q, R, S)$ is computed as a function of the per-resource cost function C' and the per-activity cost function C'' as follows:

$$C(A,Q,R,S) = w_Q \frac{\sum_{q \in Q} C'(q)}{|Q|} + w_R \frac{\sum_{r \in R} C'(r)}{|R|} + w_S \frac{\sum_{s \in S} C'(s)}{|S|} + \sum_{a \in A} C''(a)$$

Table 2.1: A list of hard and soft constraints considered in our timetabling program.

Name	Hard	Entity	Explanation
C_0	Yes	Activity	Each activity must be assigned a time and a room.
C_1	Yes	Resource	Each resource may only be used at most once at any given time.
C_2	Yes	Resource	A resource may not be used at a time that it is unavailable.
C_3	Yes	Resource	A resource must provide all suitabilities for all activities scheduled requiring it.
C_4	Yes	Activity	Explicit orderings between activities must be obeyed.
C_5	Yes	Resource	Some student sets cannot be scheduled simultaneously.
C_6	Yes	Room	No room may house an activity with an expected number of students larger than the room capacity.
C_7	No	Resource	Activities should be scheduled in favourable time slots wherever possible. See Section 2.2.2.
C_8	No	Resource	Daily schedules should be as compact as possible. See Section 2.2.1.
C_9	No	Resource	Travel time between two consecutive activities should be minimised.
C_{10}	No	Resource	The number of days that have at least one activity should be minimised for each resource.
C_{11}	No	Resource	Resources should not be scheduled less than two hours per day or more than six hours per day.

2.2.1 Idle time constraint

To keep the schedules of students and faculty members as compact as possible, we impose a penalty on timetables that imply long idle times. In other words, any period of time in which a person is not involved with an activity that is both preceded and succeeded by an activity should be penalised. It is possible for a resource to experience multiple non-consecutive periods of idle time on the same day.

2.2.2 Timeslot constraint

We consider any time slot after 17:00 to be suboptimal with the weight of the constraint increasing as the starting time of the activity increases. We also consider time slots starting before 11:00 to be suboptimal (albeit less than activities scheduled during the evening) as starting the day at this or any earlier time can cause students to incur a significant sleep debt and can negatively impact the ability to concentrate in some chronotypes [4]. We calculate this penalty on a per-timeslot basis meaning that an activity that runs from 16:00 through 18:00, for example, will incur an increase in its unfitness score for all timeslots occupied after 17:00 even though the activity started before this threshold.

2.2.3 Violated hard constraints

A timetable that violates any hard constraints should be discarded as unfeasible. It is possible, however, that a heuristic produces a timetable that does violates such constraints. Indeed, no proof exists that such heuristics produce a correct output. There are two ways of preventing such invalid timetables. First, the program could attempt to resolve the problem during the initial allocation of the timetable, possibly using a backtracking technique, significantly increasing the complexity of the process. Alternatively, violated hard constraints could be allowed during the initial allocation stage but be assigned a weight so heavy that any timetable violating a hard constraint could never be considered superior to a timetable that respects all hard constraints. The constraints violations may then be fixed during the optimisation stage as described in Section 3.3.

Our implementation relies on the optimisation process to resolve any violated hard constraints and assigns a weight of 10^3 to violations. A more intuitive solution may be to assign an infinite weight to such violations to ensure that they completely invalidate the timetable. A weakness of such an approach would be, however, that all solutions violating at least one hard constraint would carry an infinite unfitness. Thus, it would be impossible to compare two timetables that both contain a hard constraint violation, even if one contains less and is thus a better timetable that may be closer to completion.

CHAPTER 3

Heuristics

The timetable problem can be approached in two fundamentally different ways. First, we can apply existing algorithms to the problem that are guaranteed to either find the optimal solution or, more realistically, find an approximation of the optimal solution. Such algorithms generally rely on algorithms for similar problem and solve the timetable problem by first reformulating it as an instance of another problem. Such an algorithmic approach can be proven mathematically to provide a solution or an approximation of a solution. Due to the size of the problem instance, however, applying exact algorithms could take an infeasible amount of time. Second, we can employ a different class of algorithms referred to as heuristics. Such approaches usually (but not always) follows simpler rules to produce a solution more quickly even if that solution is not optimal. Heuristic approaches lack proofs of correctness meaning that a solution is not guaranteed.

To decrease the complexity of our heuristic approaches further we subdivide the timetabling problem into a two-phase problem to allow us to solve the problem using two simple heuristics instead of a single complex heuristic. The first phase consists of generating an initial, flawed timetable according to a simple greedy algorithm that is designed to be both fast and easy to understand. The second step consists of improving the timetable by repeatedly applying one of several basic optimisation methods [5].

In this chapter we will first explore some reductions for algorithmic approaches as such approaches can provide insight into different methods of not only solving the problem but also storing it inside a computer in such a way that it can be efficiently modified, measured and examined. We will also describe initial allocation heuristics and optimisation strategies that have had some success in the past.

3.1 Possible reductions

A popular approach to the timetabling problem is to reformulate it as a different problem which enables the application of more well-known algorithms to the problem. Additionally, a different formulation of a problem can sometimes make it more suitable for representation in a computer program by altering the level of abstraction. A downside of such an approach is that it makes it harder for people to relate to the problem and implement the results and suggestions that result from the research as the method of solving the problem becomes more removed from the original problem. It also makes it harder to apply existing human intuition to the problem. Still, we discuss these approaches as they may provide useful methods not only for solving the problem but also for representing it in a computer program.

In the case of the timetabling problem, some problems are more suitable as reformulations than others. In previous research, the most common formulations of the timetable problem have been as an integer linear programming problem and as a graph colouring problem. An advantage of these problems is that they are well known and many existing algorithms as well as software implementations of those algorithms already exist. We will now briefly discuss these reformulations and their applications to the timetabling problem in literature.

3.1.1 Integer linear programming

Ribiü and Konjicija [6] propose a formulation of the timetable problem as an integer linear programming problem (ILP). Such a problem consists of a system of an arbitrary number of linear equations, often dependent on many variables, that must all evaluate to a certain value or range of values. Specifically, a boolean implementation of this problem (also known as a 0/1 integer programming) can be formulated such that the sum of variables indicating the number of activities that use a resource at the same time must be less than or equal to one. Given, for example, a boolean matrix x for a simplified timetabling problem such that $x_{t,l,r}$ is true if and only if lecturer l (out of a set of lecturers L) is teaching an activity in room r (an element of R) at time t (out of all possible times T), we could construct a linear equation to ensure that no room is used more than once as follows:

$$\forall t' \in T, \quad \forall l' \in L, \quad \sum_{r' \in R} x_{t',l',r'} \leq 1$$

The system of linear equations can then be expanded using similar constraints such that all requirements imposed by the specific instance of the timetabling problem is covered. While Daskalaki, Birbas, and Housos [7] provide some methods for implementing additional constraints to the integer linear programming model, the constraints provided are much more basic than those encountered in the case of the Science Park campus and solutions to such problems can be extremely complex. Additionally, while libraries exist for solving linear programming problems such as the *GNU Linear Programming Kit*[1], an integer linear programming approach to the timetabling problem is likely to provide less intuitive results that are harder for human timetablers to implement. Thus, we deem this formulation unsuitable for this thesis.

3.1.2 Graph colouring

The reduction from the timetable problem to the graph colouring problem has been conceived as early as the nineteen sixties [8] and has been expanded on as computer hardware became more capable of handling complex systems. Burke, Elliman, and Weare [9] propose a reduction from the timetable problem to the graph colouring problem in which each node represents a resource and the colour assigned to a node represents the time at which it is used. Edges are created between activities when they share a resource. The constraint that a resource may only be used by one activity at a time is then implied by the fact that such a scenario would create an edge between two activities of the same colour, invalidating the graph colouring solution and thus the timetable. An example of a simple timetable problem instance represented as a graph colouring problem is given in Figure 3.2. Badoni and Gupta [10] and Malkawi, Hassan, and Hassan [11] extend this graph colouring approach to support different constraint types. To solve a timetabling problem formulated as a graph colouring problem, a fast albeit unintuitive solution would be to use semidefinite hyperplane programming approximation algorithms which can run in polynomial time [12]. To keep the results of this thesis useful for human timetablers we avoid such complex solutions in favour of simpler heuristics.

[1] `http://www.gnu.org/software/glpk/glpk.html`

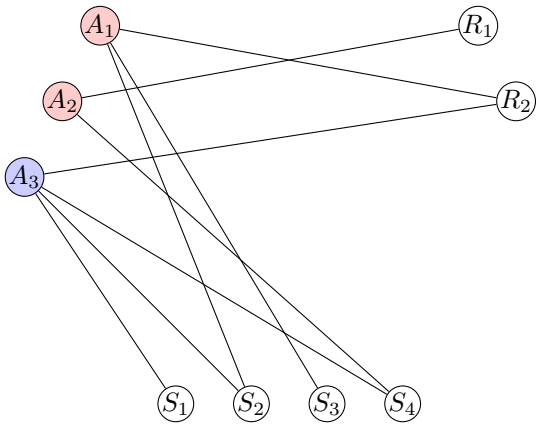


Figure 3.1: A solved timetable formulated as a graph using a tripartite construction of activities, students and rooms. Faculty members are not explicitly included here but could be grouped with students.

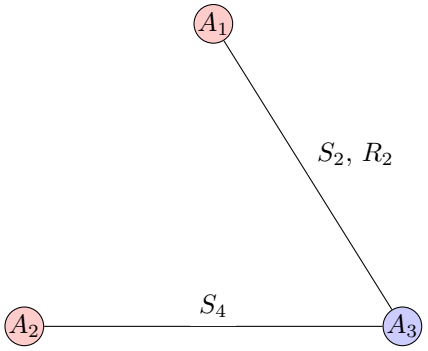


Figure 3.2: A solved timetable equivalent to Figure 3.1 formulated as a graph using a construction of activities, where edges indicate two activities cannot be scheduled at the same time because one or more resources would be used twice at the same time. Note that this graph is coloured without two neighbouring nodes sharing the same colour.

3.1.2.1 Resource-aware graph colouring

To decrease the complexity of adding additional constraints to the problem as well as make the heuristics more intuitive, we propose a slightly modified version of this reformulation which deviates somewhat from the traditional graph colouring problem. We propose a bipartite graph consisting of an independent set containing all activities and an additional independent set containing all resources. An edge exists between an activity and a resource if the activity utilises the resource. A timetable respects the constraint that each resource may only be used by at most one activity at any time if no resource shares edges with two or more activities of the same colour. Because the allocations between activities and locations are not predefined, some of these edges must be created at runtime. To simplify this process, we group the location resources in a third independent set, creating a tripartite graph. An example of such a formulation is given in Figure 3.1.

While we have indeed implemented our program using a graph-based data structure, there are some ways in which our specific timetabling problem differs from standard graph colouring problems. Perhaps the most important difference is that activities in our problem may span over multiple timeslots and as such may not be accurately described using a single colour. Indeed, an activity starting at 9:00 and which lasts for two hours will overlap a four hour activity starting at 10:00 even though both the starting time and the length differ.

3.2 Initial allocation

With the timetabling problem separated into two different problems, the first of these sub-problems can in turn be subdivided into two parts, Indeed, a heuristic for generating an imperfect initial allocation can be described as consisting of a strategy for ordering the list of activities and a strategy for selecting a timeslot for each activity. This section describes the different strategies implemented for these two sub-problems.

3.2.1 Activity sorting

As more activities are scheduled, fewer and fewer timeslots remain available which makes it harder to schedule further activities. It is therefore imperative that any timetabling solution that schedules its activities sequentially considers the order in which it does so. To analyse the different sorting strategies for activities we must first consider which factors affect the difficulty of scheduling an activity or, in other words, which factors have the largest effect on the number of possible timeslots for an activity.

Two possible factors which impact the number of available timeslots are the number of students assigned to an activity and the number of constraints imposed on that activity. Indeed, activities with a larger number of students require larger rooms which are often scarce and small activities can be scheduled in larger rooms while the opposite is not true. Similarly, the set of times in which a tightly constrained activity may be scheduled only decreases as more or the activities to which it is constrained are scheduled. To account for these two factors, we implement the following sorting strategies:

Largest activity first Schedules activities by the expected number of students with the largest activities being scheduled first.

Smallest activity first Schedules the smallest activities first.

Most constrained activity first Schedules the activities in order of descending number of constraints to other activities.

Because activities generally fall into one of several size classes, many activities have identical sizes. To allow finer control over the sorting of the activities, we allow selection of both a primary

sorting key as well as a secondary sorting key. This is implemented by sorting on the secondary key first, followed by a stable sorting algorithm using the primary sorting key. This ensures that all activities are sorted according to the primary sorting key while any set of activities with identical values is sorted according to the secondary sorting key. As not all combinations of two of these sorting keys are viable (for example, sorting by ascending size as the primary key and descending size as the secondary key accomplishes nothing), we are left with four viable combinations of sorting keys.

3.2.2 Timeslot and room allocation

The selection of timeslots and rooms, while seemingly separate problems, are deeply intertwined. Indeed, the timeslots that can be selected for an activity are dependent on the availability of rooms that can house that activity and the availability of rooms is in turn dependent on the selected timeslot and adjacent activities. The selection of a time and a room for each activity is therefore done simultaneously. We implement one complex heuristic to schedule activities as well as three trivial naive scheduling strategies for comparison purposes. The four schedulers implemented are the following:

Static scheduler Assigns to each activity the first timeslot on the first day and the first room.

Random scheduler Assigns a random day and a random timeslot to each activity as well as a random room.

Room-sufficient scheduler Assigns a random day and a random timeslot to each activity and selects the first room that is both large enough to house the activity and has all necessary suitabilities.

Smart scheduler A scheduler that attempts to intelligently select a day, timeslot and room as described in Section 3.2.2.1.

3.2.2.1 Smart scheduler

Our heuristic for intelligently selecting a suitable timeslot and room for an activity involves first retrieving a list of rooms that are capable of housing that activity, sorted by ascending capacity. Then, for each timeslot in the week that all resources for an activity are available, we then select the first (and thus smallest) room that is available at that time and a number of consecutive timeslots equal to the length of the activity. After all rooms have been visited or all timeslots have been assigned a smallest room, a timeslot and the corresponding room is selected. The timeslot selected should be positioned as centrally in the week as possible. If the activity is constrained to happen after another activity we select a timeslot that is later in the week instead and for activities constrained to happen before another activity we select a timeslot earlier in the week. This method is also described in Algorithm 1.

3.3 Local search optimisation

A simple yet effective metaheuristic for optimisation in combinatorial problems is the local search method. Local search navigates through a so-called neighbourhood of solutions to a problem by applying some operation to the problem such as swapping around activities. Given enough iterations, such an optimisation strategy should improve the quality of the solution. It is possible, however, that such an optimisation strategy reaches a local maximum fitness from which it cannot escape: it may occasionally be required to temporarily decrease the quality of a timetable to increase the quality in subsequent steps. Ahuja [13] provides a concise summary of different local search optimisation techniques for the timetabling problem. This section details the three

Data: A sorted set of activities A and a set of rooms R to be scheduled in a week containing i days and j timeslots per day.
Result: Each activity in A is assigned a time and a room
while *Any activity in A is unscheduled* **do**
  $a \leftarrow$ The first unscheduled activity in A;
  $p \leftarrow$ A fraction within $[0, 1]$ implying the preferred position of a in a week as determined by the number of constraints to other activities;
  $C \leftarrow$ An array of $i \times j$ pointers to rooms, initialised empty;
  $V \leftarrow$ An array of $i \times j$ booleans initialised to true;
  foreach *resource e required by a* **do**
    foreach *time n, m at which e is unavailable or occupied* **do**
      $V_{n,m} \leftarrow$ false;
    end
  end
  foreach *room* $r \in R$ **do**
    foreach *time* n, m **do**
      if *r is available,* $C_{n,m}$ *is empty and* $V_{n,m}$ *is true* **then**
        $C_{n,m} \leftarrow r$;
      end
    end
  end
  Select timeslot n and m such that $C_{n,m}$ is not empty and $|\frac{n \times j + m}{i \times j} - p|$ is minimal;
end

Algorithm 1: A greedy scheduling heuristic which attempts to avoid scheduling activities at times that the required resources are unavailable.

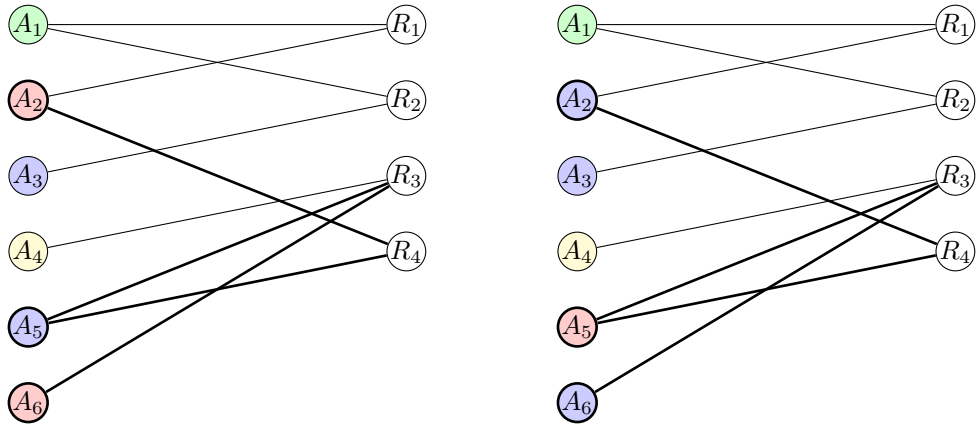
Table 3.1: A list of neighbourhood moves implemented in our timetabling program.

Name	Count	Description
$\mathcal{N}_0$	1	**Single-activity time randomisation** assignes a random new time to an activity.
$\mathcal{N}_1$	2	**Double-activity time swapping** swaps the times of two activities.
$\mathcal{N}_2$	≥ 2	**N-activity time swapping** cyclically swaps the times of an arbitrary number of activities.
$\mathcal{N}_3$	2*	**Kempe chain time swapping** creates a chain of activities and resources, then swaps those. Additional activities are selected afterwards as described in Section 3.3.1.1.

sub-problems of timetable optimisation: strategies for selection of activities to change, the way activities are modified and strategies for avoiding local optima.

3.3.1 Neighbourhood definition

The neighbourhood of any solution s within the solution space of the problem has a neighbourhood $\mathcal{N}(s)$ containing all solutions which can be reached by applying some operation to solution s. The operations which can be applied to a solution are not, however, inherently defined in the problem and must be carefully considered. Such operations generally apply to some number of activities chosen from the timetable and alter either the timeslot to which the activities are assigned or the room that the activity is assigned to [14]. In this thesis, we consider only neighbourhood moves that affect the timeslot assigned to an activity as it is generally possible to allocate an initial timetable such that the assignment of rooms is near optimal and because implementation of neighbourhood moves which alter the rooms of activities is more complex. The four different neighbourhood moves we implement are described in Table 3.1.



(a) Before (A_2, A_5) Kempe neighbourhood swap (b) After (A_2, A_5) Kempe neighbourhood swap

Figure 3.3: An example of a Kempe chain neighbourhood swap.

It is worth noting that $\mathcal{N}_1$ is simply a variant on $\mathcal{N}_2$ in which the number of activities is always equal to two. Another important consideration is that when the time assigned to an activity is modified, any serial constraints assigned to that activity are also modified accordingly. In the absence of this constraint, activities with a serial constraint would have very little chance of ever improving their impact on the timetable as any potential improvement would be overshadowed by an increase in the number of broken hard constraints.

3.3.1.1 Kempe neighbourhoods

Kempe chains, in the context of graph colouring problems, are chains of nodes connected by edges that have one of a small selection of different colours [15]. Such chains were one of the first and most promising techniques in proving the four-colour theorem and have proven indispensable in the field of graph theory since. Such chains have also had success in solving timetabling problems [16]. Our implementation of Kempe chain swapping considers two activities, A_1 and A_2. The number of selected nodes is then expanded by considering all activities that are connected to A_1 - that is to say they share a resource - and are scheduled at the same time as A_2. This process is then repeated for each newly selected node, alternately selecting activities that are scheduled at the same time as either A_1 or A_2. When no more nodes can be added to the Kempe chain, all nodes scheduled at the same time as A_1 (including A_1 itself) are rescheduled to the time at which A_2 is scheduled and similarly all activities scheduled at the same time as A_2 are rescheduled to take place at the time A_1 was previously scheduled. An example of a Kempe chain swap is given in Figure 3.3.

3.3.2 Move selection

One important aspect of any local search strategy is the choice between taking the *first* neighbour that improves the solution quality or selecting the *best* neighbour, namely the one that provides the largest increase in solution quality. These approaches are respectively referred to as *simple* and *steepest ascent* hill climbing [17]. It is worth noting that steepest ascent hill climbing requires evaluation of all neighbouring solutions, a space which, while infinitesimal relative to the complete solution space, is still very large in absolute terms. Indeed, any timetable solution has a number of neighbours that is polynomial in the number activities although this is dependent on the exact definition of the neighbourhood.

Simple hill climbing, in turn, relies on selecting the first better adjacent solution while the term *first* has no formal definition in this sense. One common approach to simple hill climbing is, for a lack of ranking between neighbouring solutions, to randomly select a move through the

neighbourhood. This does not require evaluation of all neighbouring solutions as steepest ascent hill climbing does. Such an approach is often referred to as *stochastic* hill climbing. Stochastic hill climbing can, in some cases, be improved upon by not using a uniform probability distribution function but by assigning each neighbourhood move a weight as will be described in the next section. The timetabling problem provides an excellent opportunity for implementing such an approach as the unfitness of a solution is based on the individual unfitness of its activities.

3.3.3 Activity selection

Perhaps the simplest way of selecting activities is by randomly selecting two activities. This method has an important downside, however, as it is prone to selecting activities that are already scheduled in such a way that their impact on the quality of the timetable is limited. Swapping such activities would then yield very little if any improvement and effectively waste an optimisation cycle.

Considering that the overall fitness of a timetable is dependent on the fitness of each activity contained within it, we can instead sort the activities by their impact on the fitness of the timetable. Selecting the most impactful activities can then save optimisation cycles that may otherwise be spent on activities that are already ideally or nearly ideally scheduled. Such a deterministic approach can, however, easily become stuck in local optima. Indeed, two activities that both strongly negatively impact the quality of the timetable may share many of the same problems, in which case swapping them would resolve nothing.

Alternatively, a stochastic process which considers the impact of an activity on the fitness of the timetable may be used to created a weighted randomised selection process which, while more likely to select badly scheduled activities, can still modify well scheduled activities by improbably selecting activities with a low unfitness. We implement this as follows. Assuming a random number generator rand() that generates a value between zero and one and a sorted list of activities A, we randomly choose an index i:

$$i = \lfloor \text{rand}()^{\alpha} \cdot |A| \rfloor$$

This method select an index in the list at random with a bias towards indices at the front of the list. An increase in the value α will increase this bias while a decrease in α implies a more equal probability density over the length over the list. In the case that $\alpha = 1$, the behaviour is identical to a uniform random selection strategy.

3.3.4 Hill climbing

After the local search method has selected a neighbouring solution of the current solution, the difference in unfitness implied in that move must be evaluated and the move must be accepted or rejected on the basis thereof. The simplest method of deciding which moves to accept is to only accept moves which improve the quality of the timetable. This is often referred to as hill climbing. While relatively simple to understand and implement, hill climbing strategies are very susceptible to becoming trapped in local optima. That is to say, if a solution has no neighbours that are better than it, a hill climbing heuristic will never be able to make further improvements even if better solutions exist elsewhere in the solution space.

3.3.5 Simulated annealing

One way to reduce the effect of local optima on a local search heuristic is by accepting moves which *reduce* the quality of the solution in some cases. A basic method of allowing such moves is by introducing a stochastic function that evaluates two solutions (namely the current solution and a neighbouring solution) and either allows (either if the new solution is better or if a random

process success) or denies the move. An additional value T that slowly decreases, a metaphor for temperature in real-world scenarios, is also introduced to influence the aforementioned stochastic process. A high temperature makes it more likely that a detrimental move will be taken while at a lower temperature a simulated annealing heuristic will approach a hill climbing solution by only accepting moves that improve the solution quality [18].

The most widely used acceptance threshold applied to simulated annealing and the threshold implemented in our program is the following. Given a solution s and a selected neighbouring solution s' at a certain temperature T_n, the chance of the move being accepted $P(s \to s'|T_n)$ is exponential in the difference between the quality of the solution as follows:

$$P(s \to s'|T_n) = \begin{cases} 1, & \text{if } C(s') < C(s) \\ e^{\frac{C(s)-C(s')}{T_n}}, & \text{otherwise} \end{cases}$$

3.3.5.1 Cooling schedule

An important part of any simulated annealing method is the manner in which the temperature is initially determined and the manner in which the temperature is gradually decreased. This part of a simulated annealing strategy is generally referred to as the cooling schedule and differs from problem to problem. In our case, the initial temperature depends on the penalty for hard constraint violations P_{hard} such that the chance of a move that worsens the solution by that values has a chance of less than one percent of being accepted. The initial temperature T_0 can then be determined as follows. Given that our penalty for broken hard constraints equals 10^3, the starting temperature approximately equals 217.

$$T_0 = \frac{-P_{hard}}{\ln(0.01)}$$

Cooling is implemented as a power function such that the temperature of the the system after n optimisation steps, T_n, is equal to the temperature at the previous state T_{n-1} multiplied by some constant value T_Δ between zero and one. The temperature at any optimisation cycle can thus be otherwise defined as follows:

$$T_n = T_0 \cdot T_\Delta^n$$
$$T_n = T_{n-1} \cdot T_\Delta$$

Finally, to determine the base of the exponentiation T_Δ, we consider the starting temperature T_0 and the goal temperature which is arbitrarily determined to be one. As the goal temperature should be reached after the desired number of optimisation cycles k has been performed, we can determine that the starting temperature multiplied by the exponentiation base raised to the power of k should equal one. We derive the value of T_Δ as follows:

$$
\begin{aligned}
T_k = 1 &= T_0 \cdot T_\Delta^k \\
T_0 \cdot T_\Delta^k &= 1 \\
T_\Delta^k &= \frac{1}{T_0} \\
T_\Delta &= \sqrt[k]{\frac{1}{T_0}}
\end{aligned}
$$

3.3.6 Tabu search

Tabu search is another method of improving local search methods which had some success in the past [19, 20]. Tabu search assumes two basic rules. First, much like simulated annealing, tabu search methods can make neighbourhood moves that decrease the quality of the solution (although this is generally not decided using a stochastic method and no temperature exists). Second, tabu search implements its namesake memory: a list (often with a predefined maximum capacity) of *tabu* solutions that have already been visited. Such solutions are not revisited to avoid returning to previous local optima.

3.3.6.1 Obstacles

We do not implement tabu search as there are several important obstacles that prevent this. Tabu search assumes evaluation of all neighbourhood moves, selecting the best move even if that move is detrimental to the quality of the solution. While this is feasible for many problems, the neighbourhood for our particular problem is so large that this becomes unfeasible. While stochastic selection of neighbourhood moves works for hill climbing and simulated annealing, such an approach would negate the advantages gained by implementing the tabu table as the chances of a move being selected that will return to the previous state is inversely related to the size of the neighbourhood.

Additionally, tabu search requires some method of storing a solution in the tabu list. Due to the size of a timetabling problem solution as well as the fact that memory may not be continuous, storing a uniquely identifiable timetable in the tabu list presents a challenge in the area of software development. This point is further complicated by the idea that two timetables - one that is considered for a neighbourhood move and one that is stored in the tabu list - must be fast to avoid spending large amounts of time considering the membership of a candidate solution in the tabu list.

3.4 GRASP

Greedy randomised adaptive search procedures (*GRASP* for short) are popular metaheuristics that approximate the optimal solution to a problem by repeatedly executing a two-stage process of construction an initial solution and improving that solution using a local search strategy. GRASP metaheuristics thus differentiate themselves from the aforementioned local search methods in that they occupy a wider scope and indeed repeatedly execute a local search heuristic. While we do not currently implement GRASP in our program (because analysis of such metaheuristics provide little insight into the performance of constituent heuristics and because of the difficulty of implementation), such procedures have been used to successfully approximate similar problems to the timetabling problem and are advantageous for parallel programming solutions where many solution instances can be constructed in an independent fashion [21].

CHAPTER 4

Implementation

In this chapter we discuss the implementation of the aforementioned methods in the C++ programming language as well as the prerequisite stages of data collection and preprocessing. We will also discuss various formats of output and the shortcomings of our program regarding constraints. A global overview of the program developed is given in Figure 4.1.

4.1 Data collection

The timetabling process can be roughly subdivided into four parts. First, data is collected about the courses that need to be scheduled. Then, activities are split into smaller activities where required (often to fit a larger group into several smaller rooms). Third, an allocation is made between the student sets, the faculty members and the activities. It is then possible to to allocate rooms to activities. Indeed, this step requires knowledge about the size of groups as well as availabilities of students and faculty. Finally, the timetable is reviewed by the lecturers involved to ensure their demands are met.

In this thesis, we discuss only the allocation between activities and rooms since the initial data collection and final review periods are based largely on human communication with unstructured information, the parsing of which is outside the scope of this project. Introduction of a structured domain-specific language of timetable constraints could help resolve this issue in the future. The splitting of activities and allocation of students and faculty to activities is a different problem entirely and, while related, is also outside of the scope of this project. This process can also be largely automated but we use only predetermined allocations in our program.

Data collection and processing at Science Park involves two main programs. First, *DataNose* provides both a frontend for students and faculty to view their timetables as well as a powerful backend for data collection. The initial collection of course data is performed through DataNose. Second, *Scientia Syllabus+* is used to schedule each activity and contains powerful features to view constraints and find available timeslots.

While we have access to both the DataNose and Scientia Syllabus+ databases, the data is stored in the proprietary Microsoft SQL Server format which makes it difficult to convert to other database systems. While conversion tools do exist, they are either incomplete or commercial. For this reason, we have opted to construct a program in the `awk` scripting language that parses the database files and transforms them to tab-separated value files which can be easily read by lower-level programming languages such as C++. The program constructed also filters database entries to exclude any entry that is not connected to the Science Park campus. Indeed, activities and faculty from other faculties at the University of Amsterdam are also stored in these databases.

4.1.1 Filtering

The Syllabus+ database contains some entries that are irrelevant to the Science Park campus or that might even interfere with the timetabling process. It is therefore necessary to filter out some of the entries. Most importantly, we remove activities and rooms that are not related to the Science Park campus. Indeed, the databases provided include activities of other faculties at the University of Amsterdam which we ignore. Second, we remove activities which are exams, resits and several activities which should not be scheduled such as excursions and deadlines. The reason we remove exams is that those activities have certain suitabilities that cannot be fulfilled at the Science Park campus. Such activities are currently also scheduled by a central authority not involved in scheduling the rest of the Science Park activities.

To aid manual timetabling, the database also includes suitabilities that restrict activities from taking place in rooms that are much larger than the activity. This decreases the number of rooms that are viable for each activity which makes it easier for human timetablers to select a room to house an activity. An automated timetabling program does not need such aids, however, and may be hindered by them. For this reason, we remove all such suitabilities from the data during the preprocessing stage.

4.2 Output

Our implementation provides three different modes of output. First, the software is capable of representing a timetable as a comma-separated value (CSV) file which can easily be read by other programs to import a generated timetable. Second, the program may produce output in the HTML format to provide a tabular view of a timetable which facilitates visual inspection to ensure that the timetable is valid. Last, a terminal for the DOT graph description language can be used to generate a graph of the timetable as described in Section 3.1.2 that provides a global visual representation of the activities and resources. Such a graph can have tens of thousands of nodes and visualisation should only be attempted using multi-scale approaches such as the `sfdp` tool provided by *graphviz*[1]. An example of such a graph-based visualisation is Figure A.1.

4.3 Internal representation

Perhaps the simplest internal representation for a timetable is a large array with dimensions for time slots, days, weeks and rooms in which each slot can be assigned a single activity such that the room, date and time are encoded into the array index. Inserting, removing and swapping activities in such a data structure would be trivial but it would be ill-suited for estimating the quality of a timetable. Determining the amount of idle time in a student or faculty member's schedule would, for example, be a computationally expensive process as each activity would have to be evaluated to determine if a certain person takes part in it.

We have chosen for a undirected graph approach in which the entire internal representation is a tripartite graph consisting of three sets of nodes. One set of nodes contains all activities, one contains all personnel and one contains all rooms. Optionally, a fourth set could be added to represent any other resources or this could be incorporated into the set of personnel. At the start of the program, edges are created between all activities and their faculty and students. Indeed, the allocation of students and teachers to activities is already decided and outside of the scope of this project.

In our graph representation we can schedule an activity (which implies finding a time slot and a room for it) by creating an edge between the activity and the room resource as well as assigning a time to that node which is stored inside the activity node.

[1] `http://www.graphviz.org/`

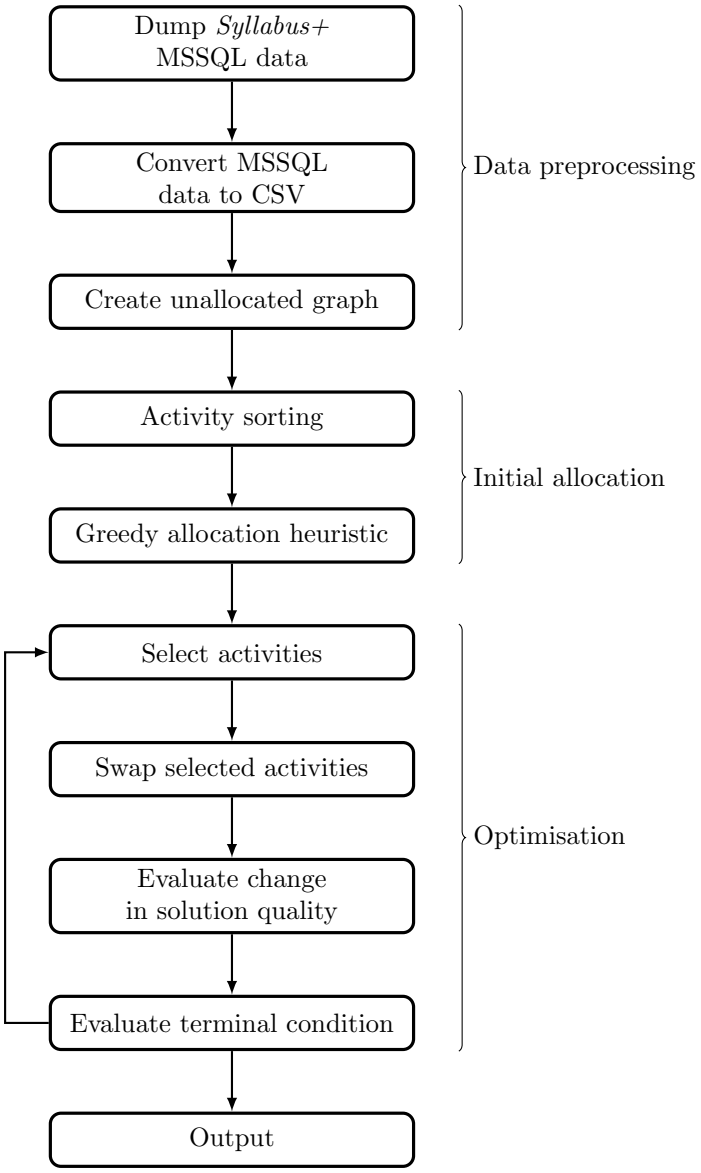


Figure 4.1: A process diagram of the timetable solver implemented.

This representation makes it much easier to evaluate the cost function for a timetable. Indeed, because all resources are grouped into sets, it is trivial to iterate over them and, as each resource contain edges to the activities it is used by, calculate values such as idle time. Additionally, storing the date and time slot inside an activity node makes it easier to validate collision and order constraints.

4.4 Parallelism

One computationally intensive part of the optimisation process and thus the entire scheduling process is the evaluation of the cost function. Indeed, the cost of the solution must be reevaluated at each optimisation cycle to compute the effect on the unfitness of that move. The cost function iterates over all activities in resources in the problem instance and does so in a fashion where all activities and resources can be treated as independent entities. It is therefore relatively uncomplicated to parallelise this process by distributing different entities over a number of con-

current threads. Our implementation employs the *OpenMP* library which allows parallelisation of a program with relatively few changes to the source code.

One caveat of a parallelised approach is that the weighted activity selection we implement relies on each activity storing a counter of its impact on the timetable quality which is set during the cost function evaluation. This may in some cases create race conditions and reduce the accuracy of the weighted selection procedure. To mitigate this drawback we supplement each activity with a mutually exclusive write lock mechanism which prevents multiple threads modifying the activity at the same time. While locking mechanisms lower the performance slightly due to threads waiting for activities to become available, we believe that the performance benefits of a parallelised approach outweigh any loss in performance due to locking.

One possible improvement that must be considered when parallelising in the aforementioned manner is the loop partitioning method used. Indeed, due to the current implementation of the input parsing, series of activities have sequential indices and activities of the same course also share the same approximate position in the array of activities. Resources tend to exhibit the same behaviour. A loop partitioning algorithm that assigns small groups of resources or even a single resource at a time is therefore likely to concurrently evaluate activities and resources that are constrained to each other. We suspect that partitioning methods that assign larger parts of the entities will perform better although we currently neither implement nor benchmark this.

4.5 Shortcomings

Constraint C_5 which prohibits concurrent scheduling of two resources is currently not fully implemented. While such constraints can be read from an input file, the smart scheduler does not account for them and they are not included in the fitness function (although evaluation of such constraints is supported here). This affects the performance of our implementation and may result in timetables performing better than they would if this constraint was implemented. While this is an obvious shortcoming, the number of constraints of this type is relatively sparse and we believe that the current fitness function still provides sufficient insight into the quality of a solution.

Additionally, constraints C_2 and C_9 which respectively assert availability of a resource and reduce the travel time between consecutive activities are fully implemented in our software but are not used as the required data is not available. We do not know of a list of travel times between locations at the Science Park campus and the availability data of resources is stored in a proprietary and undocumented format in the Syllabus+ software which we are unable to decode.

Finally, the constraint that governs sequencing between activities, C_4 is only partially implemented. Indeed, some activities are constrained to another activity such that they must not only take place before or after that activity but also such that the gap between them must be a certain number of hours or days. Our software currently only implements the basic sequencing of activities taking place before or after other activities while the minimum gap lengths are ignored.

4.5.1 Natural language constraints

It is also worth noting that many constraints are written by course coordinators as natural language instead of in a structured format, the parsing of which is outside the scope of this project. This means that it is again easier to optimise the timetable using the constraints that are available to us as compared to the actual constraints imposed. One potential solution to this problem would be to design and implement a domain-specific language such that constraints can be entered freely in a format easily parsed by a computer program. Ribic [22] presents such a

domain-specific language although not all constraints imposed at the Science Park campus are included in this proposal.

CHAPTER 5

Experiments

To test the effectiveness of our implementation described in the previous chapter and, by extension, the methods described throughout this thesis, we use the implemented program in a series of experiments to measure properties such as optimisation speed, timetable unfitness but also runtime for many different heuristics. These experiments as well as the relevant parameters are described in this chapter.

5.1 Benchmarking

Given that we consider three primary and secondary sorting methods, four allocation strategies, three local search strategies, two neighbour selection strategies and four neighbourhood definitions, we can construct 864 separate heuristics to benchmark. It is worth noting, however, that many of these combinations are not viable or identical to other combinations. Primarily during the initial allocation stage we can eliminate many combinations by ignoring duplicate sorting strategies and contradictory sorting strategies such as primarily using a largest-activity first strategy and a secondary smallest-activity first strategy. Three of four greedy allocation strategies are also unaffected by the order of activities, meaning that the actual number of initial allocation strategies is reduced from thirty-six to seven.

To further simplify our experiments, we separate the two distinct phases of initial timetable allocation and solution optimisation such that we will separately compare the seven strategies for initial allocation and sixteen optimisation methods. The experiments will be performed by executing each strategy five times. Each optimisation strategy will run for one thousand cycles before terminating. Unless otherwise noted, the weights shall be one half for student sets, one half for faculty members and zero for rooms.

5.1.1 Measurements

During the initial allocation stage we measure the unfitness of the completed timetable as well as the unfitness of the event with the largest negative impact on the solution quality. The success rate of the scheduling heuristic is measured as the percentage of activities that is both scheduled and does not break any hard constraints. The runtime in which a complete timetable is generated (excluding time spent on data preprocessing) is also measured.

The values measured for the optimisation stage in this thesis are the difference in unfitness before and after the optimisation stage, the average decrease in unfitness per step, the maximum decrease achieved in a single step, the percentage of moves that is accepted, the percentage of

moves that is an improvement, the runtime per cycle and the average improvement achieved per second.

To examine the degree in which the optimisation stage can be configured to prefer one type of resource over the other, we run experiments using the seven different permutations in which at least one of the three types of resources can be selected and equally distribute the weights over the selected resource type. The goal of such experiments is to verify that there is an observable increase in the speed of optimisation for resources when they are taken into account when the cost function is evaluated compared to cases where they are ignored. Unless otherwise noted, the weight is shared equally between students and faculty members while locations are ignored in the cost function.

The weighted stochastic local search methods depend on a power α to determine the steepness of the power function used to select activities. We experiment with different values for this power to observe the effects on the optimisation stage. Unless otherwise noted, the value for α is set to two.

Finally, while we are restricted in the number of optimisation cycles we can execute per optimisation heuristic, it is critical to analyse the behaviour of our optimisation techniques over large numbers of optimisation cycles. Indeed, one of the main drawbacks of local search techniques is that such processes can become trapped in local optima after a period of execution. Thus, we execute several optimisation heuristics (the heuristics that perform best in the short-term benchmarks) for one million cycles to observe the progression of the optimisation process.

5.2 Evaluating existing timetables

The implementation presented in this thesis allows pre-defining rooms and timeslots for certain activities which makes it possible to evaluate the current timetable and compare it to the timetables automatically generated by our software. We include this manually constructed timetable in the results for initial allocation strategies as we have no data about any optimisations on this timetable.

5.3 Hard- and software

Our program was written in the C++ programming language and compiled using the *GNU compiler collection* compiler for C++ at the highest optimisation level. The software was run on a personal computer with 8 gigabytes of memory and a first generation 2.67 gigahertz Intel i5 processor.

CHAPTER 6

Results

Due to the reliance of this thesis on a cost function, the tables and figures in this section may contain dimensionless numbers. Such measurements are expressed in terms of the aforementioned cost function unless otherwise noted. Graphs presented in this section may contain shaded areas representing one standard deviation.

6.1 Initial allocation

Table 6.1 shows the benchmarks of the seven different initial scheduling heuristics. The smart scheduler performed significantly better, regardless of sorting strategy, than any of the naive schedulers implemented. The unfitness score obtained were several orders of magnitude lower using the smart scheduler compared to the naive schedulers. The smart scheduler also achieved a success rate of at least 90% compared to rates below six percent for the naive schedulers. Execution of the smart scheduling algorithms took longer than the naive schedulers, approximately two seconds. This was approximately a factor one hundred longer than the naive schedulers.

The smart scheduler obtained the lowest unfitness when the activities were sorted primarily by size and secondarily by number of constraints in which case the unfitness equalled 63.96. The success rate using the aforementioned sorting strategy was 92.69%, the second lowest of four possible smart scheduling heuristics. The highest success rate, 96.46%, was achieved by sorting primarily on the number of constraints and secondarily on the size of the activity. The unfitness of this approach was the highest (thus the worst) of all at a value of 70.35. Smallest-first sorting strategies performed worse than similar largest-first strategies in most aspects.

Smart scheduling approaches in which the primary sorting key was the number of constraints exhibited the behaviour that the worst activities in the timetable (that is to say the activities with the highest per-activity unfitness) had a significantly higher unfitness than the same class of activities in heuristics where activity size was the primary sorting key.

The timetable as created by human timetablers achieved a success rate lower than the smart scheduler. Additionally, the handmade timetable violated some of the imposed hard constraints. The unfitness of the handmade timetable was much lower than any of the naive schedulers and the success rate was higher.

Table 6.1: The unfitness of the complete solution, unfitness of the worst activity, percentage of activities that did not violate a hard constraint and runtime for the seven different initial allocation strategies as well as the schedule as it is currently designed by human schedulers.

Scheduling heuristic	Unfitness (1)	Worst activity (1)	Success (%)	Runtime (s)
Static scheduler *	$1.042 \times 10^9 \pm 0$	$1.089 \times 10^6 \pm 0$	0 ± 0	0.02238 ± 0.0007363
Random scheduler	$1.856 \times 10^8 \pm 1.806 \times 10^6$	$6.639 \times 10^5 \pm 1.325 \times 10^5$	5.537 ± 0.1034	0.02405 ± 0.0004788
Sufficient scheduler	$4.351 \times 10^8 \pm 1.945 \times 10^6$	$6.596 \times 10^5 \pm 1.378 \times 10^5$	0.0007364 ± 0.001552	0.02365 ± 0.0006317
Smart scheduler *				
Largest first				
Most constrained first	63.96 ± 0	292.0 ± 0	92.69 ± 0	2.290 ± 0.02772
Smallest first				
Most constrained first	65.44 ± 0	236.0 ± 0	90.19 ± 0	1.695 ± 0.01923
Most constrained first				
Largest first	70.35 ± 0	446.0 ± 0	96.46 ± 0	2.099 ± 0.02051
Smallest first	69.16 ± 0	451.8 ± 0	93.91 ± 0	1.808 ± 0.10120
Human-made schedule	6.050×10^5	3.200×10^4	83.21	-

Error margins are one standard deviation.
Mean values of five executions.

* Deterministic methods

6.2 Optimisation

Due to the large number of heuristics that can be constructed using the methods presented in this thesis, the following results for the optimisation phase were found optimising timetables generated using our smart scheduler, the activities of which were primarily sorted by number of constraints and secondarily by the size of the activity, descending.

An overview of the performance of the different optimisation heuristics is shown in Table 6.2. Simulated annealing methods (see Figure 6.2) proved to be ineffective with our current cooling schedule. In all cases the unfitness of the timetable was increased when using simulated annealing methods. Hill climbing heuristics provided a better result (see Figure 6.1) and improved the quality of the solution in most cases. Swapping of randomly selected activities (either two or a random number) showed little to no improvement in the quality of the timetable.

Randomly assigning a new time to an activity or applying a Kempe chain swap were the only neighbourhood moves out of four to provide significant improvements to the timetable quality. In both cases, the effectiveness was significantly improved when using weighted stochastic activity selection (an increase in performance from -0.29% over one thousand cycles to -0.62% for single-activity time randomisation and an improvement from -0.13% to -0.22% for Kempe chain swap neighbourhoods). The largest increase in quality within a single cycle was recorded at -7.946×10^{-2} and was achieved using single-activity randomisation.

The fraction of neighbourhood solutions that was accepted was between 8.2% and 10.7% for single-activity time randomisation and between 3.9% and 4.5% for Kempe chain swaps. Simulated annealing methods showed a fraction of accepted moves that provided an increase in solution quality approximately between one third and one half.

6.2.1 Long-term sustainability

As the weighted stochastic hill climbing heuristics using either single-activity randomisation or Kempe chain neighbourhoods displayed significant decreases in the solution quality over one thousand cycles these heuristics were executed for one million cycles to measure the long-term sustainability of these heuristics. The results of these experiments are shown in Figure 6.3. Notably, the single-activity randomisation neighbourhood heuristic initially displayed a faster decrease in solution unfitness but also converged to a horizontal asymptote more quickly than the Kempe chain swap neighbourhood. After one million cycles, the Kempe chain swap heuristic had achieved a lower unfitness than the single-activity randomisation method.

6.2.2 Weighted selection parameters

Modifying the value of α, the exponent in the power law probability function for selecting activities, showed a strong effect on the performance on the optimisation heuristics. The results of the experiments performed with this value are shown in Figure 6.4. Noticeably, exponents of four and eight performed best with exponents below or above these values gradually decreasing in effectiveness. A value of one implies a uniform selection.

6.2.3 Effects of resource type weights

The results of our experiments using different weights for different resources in the evaluation of the cost function are shown in Table 6.3. A significant improvement in the rate optimisation for a certain resource type over one thousand cycles was observed for student sets and rooms when these resource types are taken into account when evaluating the quality of the timetable. Percentual student set unfitness decrease was between -0.41% and -0.42% when not selected for optimisation and between -0.55% and -0.69% when selected. Rooms showed a similar trend

with a relative decrease in unfitness of between -0.25% and -0.30% when not considered and between -0.39% and -0.42% when selected for optimisation. Faculty member resources did not show a similar improvement in cases where such resources are considered.

Table 6.2: A comparison of different heuristic optimisation methods and their effect on the (average) decrease in solution unfitness, the maximum decrease as well as the rates of acceptance and solution improvement and wall clock runtime.

Optimisation heuristic	Final unfitness (1) (% diff.)	Average decrease (1)	Maximum decrease (1)	Acceptance rate (%)	Improvement rate (%)	Cycle rate (s^{-1})
Hill climbing						
Weighted selection						
Single-activity random	69.92 (−0.62%)	-4.380×10^{-4}	-6.724×10^{-2}	10.3%	10.3%	6.48
Double-activity swap	70.35 (−0.01%)	-8.338×10^{-6}	-2.347×10^{-2}	0.1%	0.1%	6.46
N-activity swap	70.35 (±0.00%)	0	0	0.0%	0.0%	6.31
Kempe chain swap	70.20 (−0.22%)	-1.519×10^{-4}	-2.840×10^{-2}	4.5%	4.5%	6.51
Uniform selection						
Single-activity random	70.15 (−0.29%)	-2.028×10^{-4}	-4.839×10^{-2}	8.2%	8.2%	6.61
Double-activity swap	70.35 (±0.00%)	-2.371×10^{-6}	-4.523×10^{-3}	0.2%	0.2%	6.61
N-activity swap	70.35 (±0.00%)	0	0	0.0%	0.0%	6.47
Kempe chain swap	70.27 (−0.13%)	-8.909×10^{-5}	-1.660×10^{-2}	4.5%	4.5%	6.68
Simulated annealing						
Weighted selection						
Single-activity random	70.49 (+0.20%)	$+1.379 \times 10^{-4}$	-7.946×10^{-2}	23.8%	10.7%	6.60
Double-activity swap	70.35 (±0.00%)	-2.219×10^{-6}	-1.140×10^{-2}	0.5%	0.3%	6.60
N-activity swap	70.35 (±0.00%)	0	0	0.0%	0.0%	6.45
Kempe chain swap	70.52 (+0.23%)	$+1.636 \times 10^{-4}$	-5.931×10^{-2}	12.0%	3.9%	6.58
Uniform selection						
Single-activity random	70.72 (+0.52%)	$+3.671 \times 10^{-4}$	-3.337×10^{-2}	24.7%	8.4%	6.66
Double-activity swap	70.36 (±0.00%)	$+2.051 \times 10^{-6}$	-1.421×10^{-2}	0.9%	0.3%	6.70
N-activity swap	70.35 (±0.00%)	0	0	0.0%	0.0%	6.53
Kempe chain swap	70.44 (+0.12%)	$+8.195 \times 10^{-5}$	-9.741×10^{-3}	11.1%	4.2%	6.58

Mean values of five executions except for the maximum decrease which is the maximum over five executions.

Table 6.3: The effects of one thousand optimisation cycles on the quality of timetables measured on a per-resource-type basis using different resource type weights. Measured using a weighed stochastic hill climbing heuristic employing a singe-activity randomisation neighbourhood.

Weights			Unfitness difference after 1000 cycles		
Student	Faculty	Location	Student	Faculty	Location
1	0	0	−0.50 (−0.55%)	−0.27 (−0.55%)	−1.35 (−0.28%)
0	1	0	−0.38 (−0.42%)	−0.27 (−0.55%)	−1.20 (−0.25%)
0	0	1	−0.37 (−0.41%)	−0.24 (−0.48%)	−1.91 (−0.39%)
$\frac{1}{2}$	$\frac{1}{2}$	0	−0.63 (−0.69%)	−0.29 (−0.58%)	−1.48 (−0.30%)
$\frac{1}{2}$	0	$\frac{1}{2}$	−0.58 (−0.64%)	−0.28 (−0.57%)	−2.04 (−0.42%)
0	$\frac{1}{2}$	$\frac{1}{2}$	−0.38 (−0.42%)	−0.26 (−0.52%)	−2.04 (−0.42%)
$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	−0.57 (−0.63%)	−0.24 (−0.49%)	−2.03 (−0.42%)

Mean values of five executions.

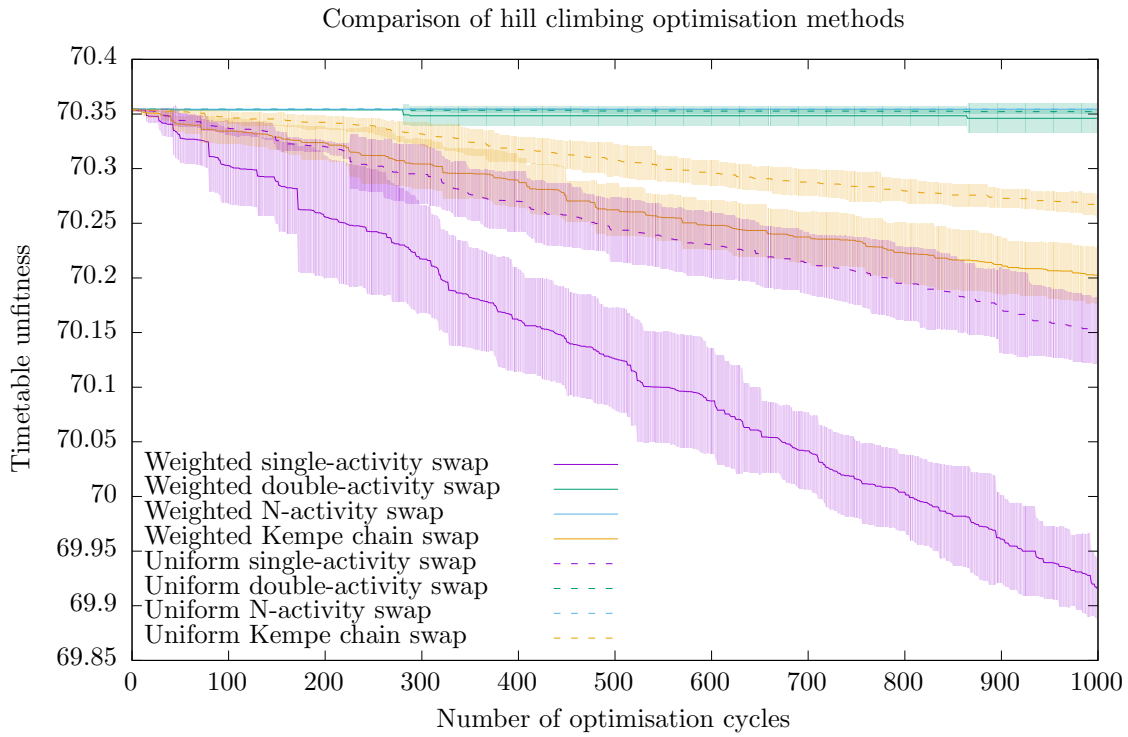


Figure 6.1: The progression of eight hill climbing heuristics over one thousand optimisation cycles. Measured over five executions.

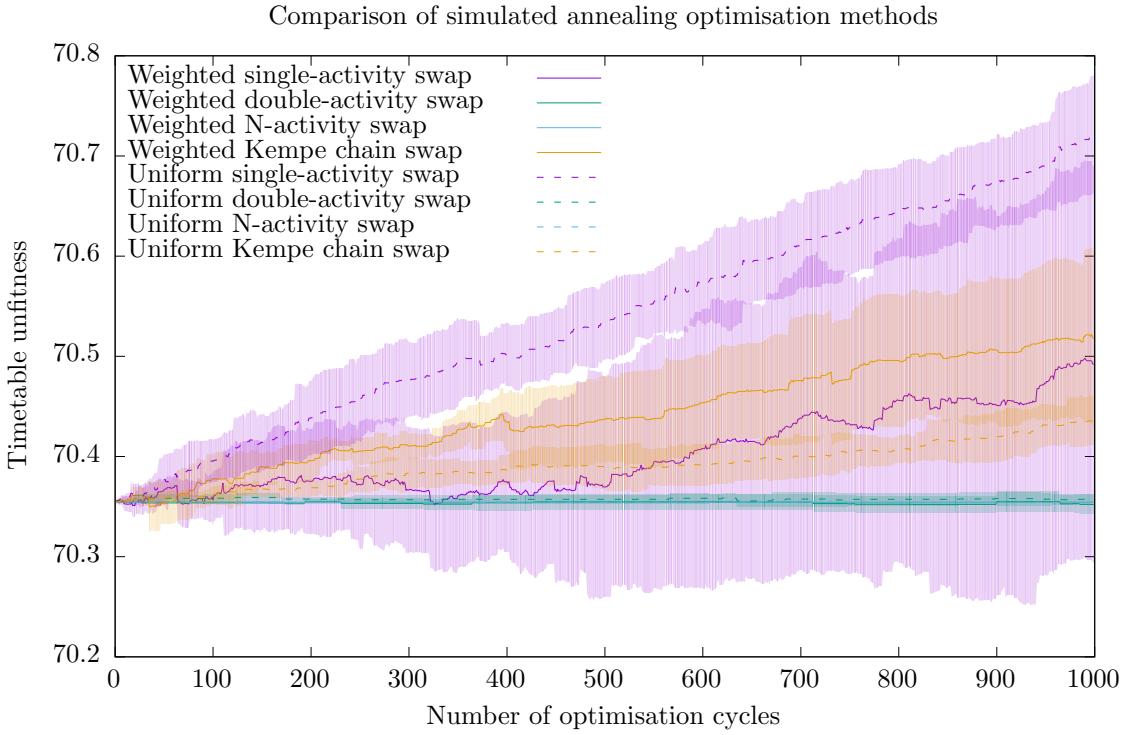


Figure 6.2: The progression of eight simulated annealing heuristics over one thousand optimisation cycles. Measured over five executions.

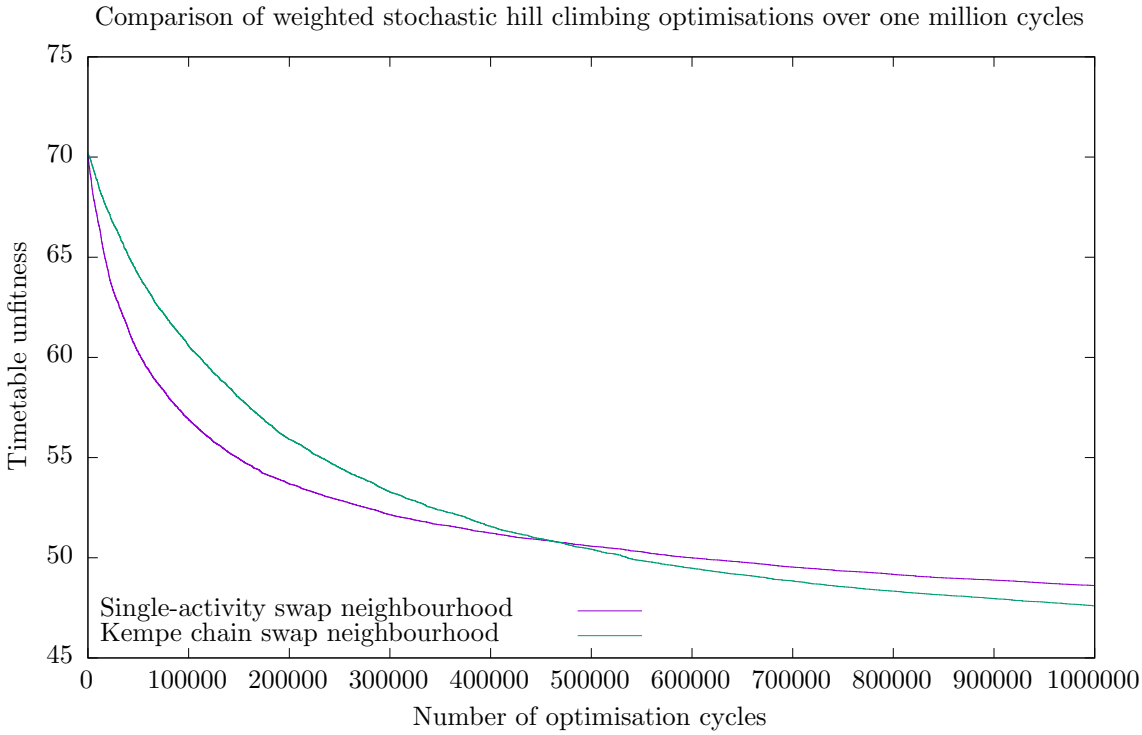


Figure 6.3: The progression of weighted stochastic hill climbing methods with either single-activity randomisation or Kempe chain swapping neighbourhoods over one million optimisation cycles.

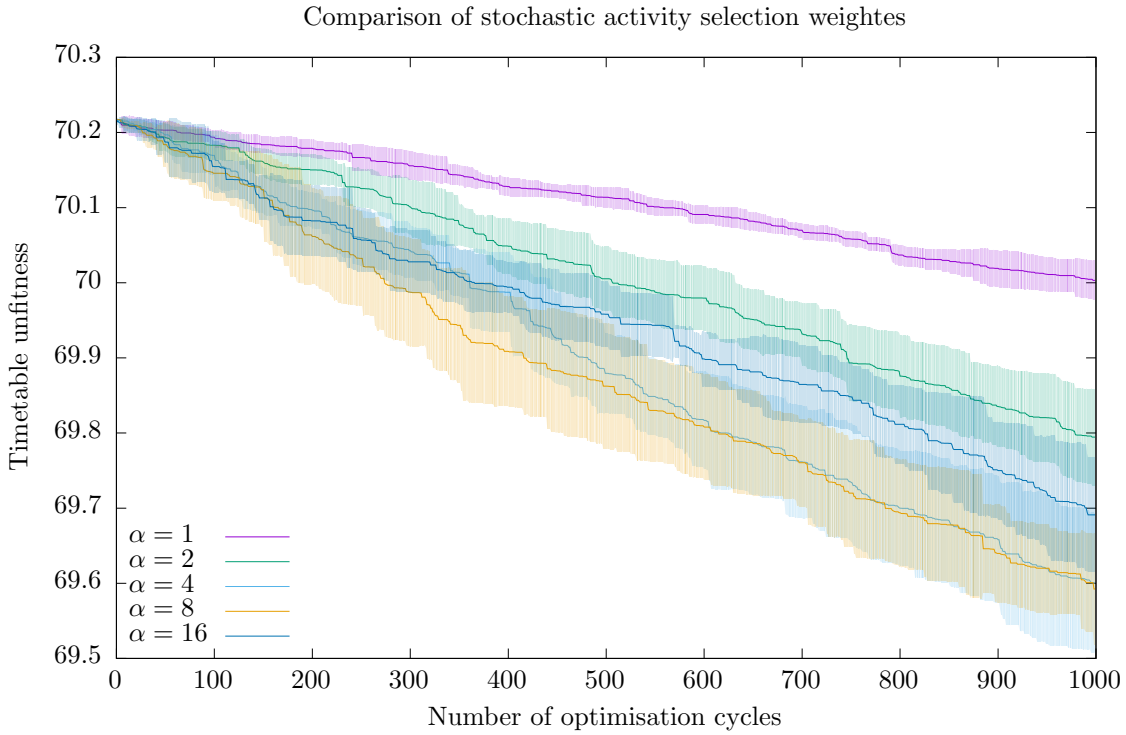


Figure 6.4: The effect of different α values for the weighted activity selection based on a power law probability function. Measured using a weighed stochastic hill climbing heuristic employing a singe-activity randomisation neighbourhood. Measured over five executions.

CHAPTER 7

Discussion

In an effort to partially automate the timetabling process at the Science Park campus we have compared different heuristic methods for assigning rooms and times to activities. These heuristic methods were comprised of seven different initial allocation methods and sixteen optimisation strategies. Although many of these strategies proved to be ineffective, some heuristics have shown promising results during both stages of the problem.

7.1 Initial allocation

During the initial allocation phase of the timetabling process, our smart scheduler is capable of creating a timetable in which up to ninety-six percent of activities are free of hard constraint violations. As such a timetable can be generated in a matter of seconds we believe that the greedy scheduling algorithm designed is a success and may be used to accelerate the timetabling process. As the percentage of activities that is successfully scheduled does not change during the optimisation phase, we consider the success rate a more important measure of the quality of a timetable than the unfitness given that no hard constraints are violated. Using the number of constraints by which an activity is bound as the primary sorting key and the size of it as the secondary sorting key is then the most effective sorting strategy for our smart scheduler. While our smart scheduler also outperforms the manually created timetable in our benchmarks, this does not imply that it is objectively superior as described in Section 7.3.

7.2 Optimisation

Following initial allocation, only two of the four neighbourhood moves implemented proved to be effective for optimisation. The cyclical swapping of two or more randomly chosen activities achieved little to no improvement in any of the benchmarks. A possible explanation for this behaviour is that, due to the number of constraints between activities, the chance that a change in the time assigned to an activity increases the quality of the timetable is low. Indeed, randomising the time of a single activity yields a rate of improvement of only ten percent. Cyclically swapping randomly selected activities behaves similarly to assigning each activity a random time but applies the operation to all activities simultaneously. Therefore, the chance of a cyclical swap move yielding an improvement may be exponential in the number of activities. This does not fully explain the phenomenon, however, as the chance of a double activity time swapping move being successful is in all cases less than the square of the change of a single activity randomisation being accepted.

One remarkable finding is that while single-activity randomisation neighbourhood moves show the fastest optimisation early in the optimisation phase they quickly approach a horizontal asymptote. These heuristics are then surpassed by Kempe chain swapping neighbourhoods which, while initially slower, are more effective after a large number of cycles and eventually surpass the randomising heuristics. It may be possible to combine these properties in a single heuristic which dynamically selects one of the two successful neighbourhood moves. A heuristic which operates in a solution space comprising of other heuristics is referred to as a hyperheuristic and may prove useful for our timetabling problem.

In problems such as the timetabling problem where it is infeasible to examine the entire neighbourhood of a problem, stochastic local search proves to be an effective alternative. Especially in cases where a meaningful sorting of entities can be achieved such as in the timetabling problem where activities are sorted by their individual impact on the timetable unfitness a weighted approach can be effective. In our experiments the weighed selection heuristics which achieved the best results (that is to say with an exponent between four and eight) showed an improvement in solution quality approximately three times greater than a uniform stochastic method.

The hill climbing local search method performed much better in our tests than the more complicated simulated annealing method. None of the simulated annealing methods resulted in an improvement in the quality of the timetable. A different cooling schedule may prove to be more successful in the future. Indeed, our results are in conflict with many authors who have shown good results in the past using this method although the problem was in most cases less constrained than the instance presented at the Science Park campus.

7.3 Weaknesses

The most significant shortcoming in our implementation is the fact that four of the twelve constraints imposed are only partially implemented. This makes it significantly easier to generate a valid timetable and means that a comparison between our implementation and a manual timetable is somewhat flawed. Human timetablers must also account for other constraints that are not stored in the provided databases. It is therefore not surprising that our scheduling heuristics can outperform a timetable that took months to design.

Further complicating comparisons between timetables is the fact that we have evaluated the quality of the timetables using a single cost function which may be a bad representation of the perceived quality of a timetable in the real world. This also means that manually created timetables may be designed with different goals such that they may perform poorly in our measure of timetable quality but excel in other measures. Despite these drawbacks, however, our cost function accounts for the hard constraints such that it is at least an accurate measure of the number of violated hard constraints and thus the validity of the timetable.

Due to the time required to create timetables with a realistic number of optimisation cycles, we have been unable to generate benchmarks for all combinations of heuristics. Instead we have often chosen a single partial heuristic which performed well in previous tests. A consequence of this is that the results are incomplete and may not give a complete overview of the strengths and weaknesses of different heuristics. Indeed, it may be possible that an optimisation heuristic that has performed below par during the initial allocation stage would have excelled during the optimisation stage due to a different internal structure of the timetable. Similarly, the number of optimisation cycles used to benchmark the optimisation strategies may be insufficient. We have shown that some optimisation strategies such as Kempe chain swap neighbourhoods perform better than single-activity randomisation when the number of optimisation cycles is large but we have not had sufficient time to assert that this is not also true for other optimisation strategies.

The poor performance of the simulated annealing strategies may be attributed to a poorly designed cooling schedule. As we have only implemented one simple cooling strategy, we cannot determine if the poor performance of simulated annealing is due to the poor cooling schedule

or due to the nature of the problem in which the vast majority (between ninety and ninety-five percent) of neighbourhood moves will not result in an improvement.

CHAPTER 8

Conclusion

In this thesis we have presented a heuristic approach to timetabling that can create a feasible timetable within a short amount of time. While the program written does not account for all constraints posed, we believe that the software can partially automate the timetabling problem by providing human timetablers with a base timetable to expand upon and improve. Indeed, ninety-six percent of activities can be scheduled in a valid manner using the automated techniques described in this thesis. We hope that this can save time on the timetabling process which can then be spent perfecting the timetable to improve the overall quality of it for students and faculty.

While we are convinced that the best way of generating an initial timetable is by using a smart scheduler and using most-constrained-first activity sorting followed by estimated size sorting, the distinction between optimisation heuristics is less clear. In our current implementation, however, hill climbing using weighted stochastic activity selection proved highly effective at improving the quality of a timetable. Combined with either a single-activity randomisation neighbourhood or a Kempe chain neighbourhood, the unfitness of a timetable can be reduced significantly.

8.1 Future research

One bottleneck in the performance, and as a direct consequence the the number of optimisation cycles that is feasible, is the speed at which one optimisation cycle can be executed. A large portion of the time spent in one optimisation cycle is used to reevaluate the quality of the timetable after a candidate neighbourhood move. It is worth noting then, that each neighbourhood move only modifies a relatively small number of activities meaning that a lot of time is spent reevaluating activities for which the impact on the timetable has not changed at all. A selective method of evaluating the timetable quality in which only activities that have been modified since the last evaluation are updated may speed up the evaluation of the timetable qualities by several orders of magnitude.

Partially due to the lack of the aforementioned programming optimisation, the speed at which a timetable can currently be optimised is measured in tens of cycles per second. This bottleneck strongly limits the depth at which we can evaluate different heuristics. Indeed, the vast majority of heuristics are currently limited to one thousand cycles while our results show that some heuristics progress differently over larger numbers of cycles than others. Should the evaluation of the timetable cost be optimised, the benchmarks described in this thesis should be repeated using a larger number of optimisation cycles to gain more insight in the effectiveness and sustainability of different heuristics.

The unexpectedly low performance demonstrated by simulated annealing approaches in our implementation may be partially due to the cooling schedule implemented. The current cooling schedule demonstrates frequent transitions to states with higher unfitness. Better results may be achieved by adopting a cooling schedule with a lower chance of accepting detrimental moves with very small increases in the unfitness of the timetable. Indeed, due to the large number of optimisation cycles such small increases can have a large combined impact on the quality of the timetable.

Furthermore, a possible improvement to the optimisation step would be the ability to choose between one of the implemented neighbourhood moves as one move may change the structure of the timetable such that a different move is more likely to succeed. One example of such behaviour is the Kempe chain swapping move which depends on the existence of many different Kempe chains although the move itself does not create new chains while a move such as single-activity time randomisation can create new chains. A hyperheuristic could be designed to choose one of the implemented neighbourhood moves at each optimisation step. Even an extremely basic hyperheuristic randomly choosing a neighbourhood move may provide an increase in both the short-term performance of the optimisation process as well as the sustainability of it in cases where the number of optimisation cycles is very large.

During the data collection stage, to mitigate the problem of having many constraints written in unparsable human language, we suggest that a domain specific language be developed or expanded such that course coordinators could enter the requirements of their courses in a manner that is both easily written and understood by a human and can be parsed to constraints in a computer program. Limited research has been done on this subject and may prove essential to increasing the level of automation within university timetabling.

Finally, we propose a way to implement the heuristics found in this thesis into the work flow of human timetablers. Namely, the optimisation techniques described could be used to suggest changes to a timetable to persons experienced with the process of timetabling. Examination of the suggested change by the operator would then lead to a decision used to train the parameters of the optimisation heuristic via machine learning to improve the quality of future suggestions. Such a method could simultaneously provide insights into the optimal parameters for the cost function and optimisation heuristics and improve the quality of the timetable.

APPENDIX A

Visualised completed timetable

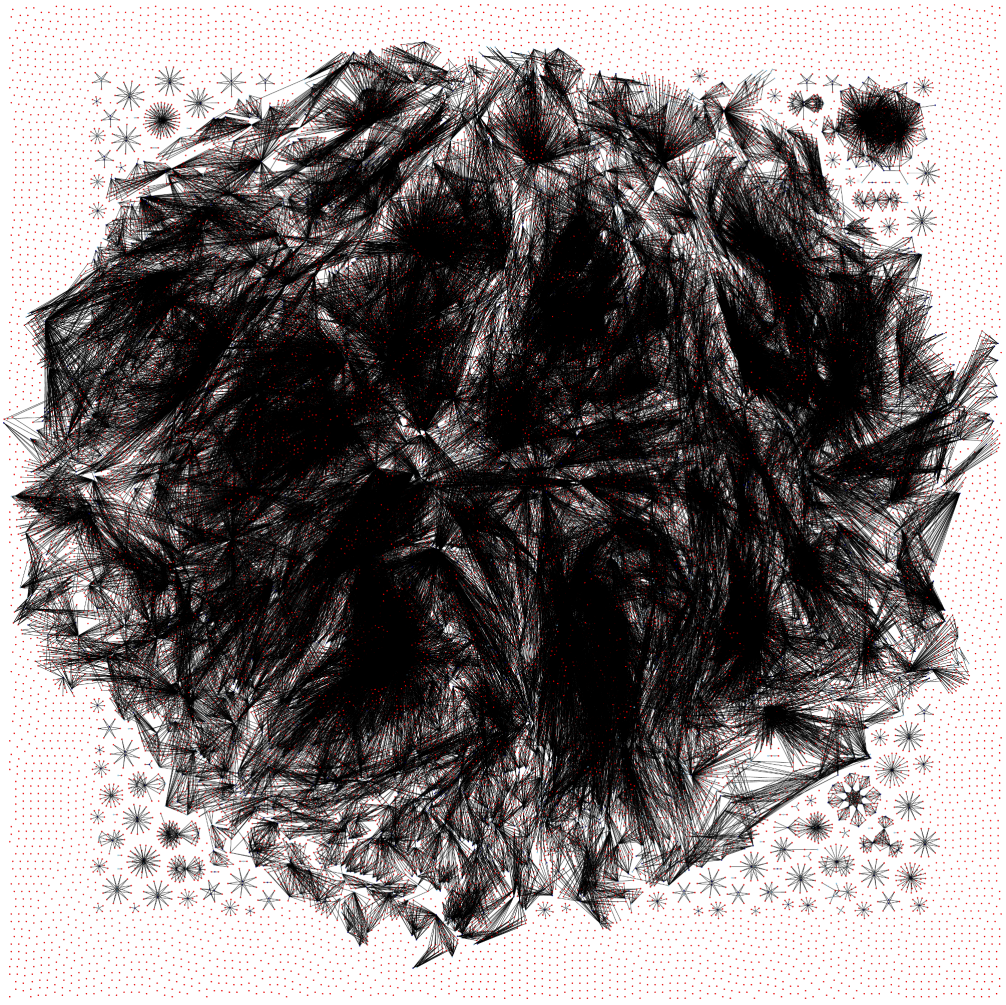


Figure A.1: A completed timetable of all activities scheduled at the Science Park campus in the academic year 2015-2016, where red nodes are activities, blue nodes are resources and pink nodes are rooms.

APPENDIX B

Weights of soft constraints

Table B.1 details the weights of soft constraints implemented in our timetabling program. These weights were established in agreement with Mr R.J. Verbeek who is currently heavily involved in the timetabling process. The constraints presented here are weighted to the guideline that one hour of inconvenience of any kind is weighted as approximately one point of unfitness or one quarter of a point per timeslot.

Table B.1: The weights of the soft constraints considered in our timetabling program.

Name	Weight
C_7	Given an activity, the unfitness of that activity for a particular resource equals $\frac{1}{4}$ for each timeslot occupied before 11:00 and $\frac{1}{2}$ for each timeslot after 17:00.
C_8	Given a resource, the unfitness of that resource is increased by $\frac{1}{4}$ for each timeslot which is not occupied but for which there is an activity both before and after that timeslot on the same day.
C_9	Not currently implemented.
C_{10}	Given a resource, the unfitness of that resource is increased by one for each day that the resource has at least one activity scheduled.
C_{11}	Given a resource and a day, the unfitness of that resource is increased by $\frac{1}{4}$ multiplied by the difference between the number of occupied timeslots in the given day less than eight (two hours) or more than twenty-four (six hours).

Bibliography

[1] Tim B. Cooper and Jeffrey H. Kingston. "The complexity of timetable construction problems". In: *Practice and Theory of Automated Timetabling*. Springer Science, 1996, pp. 281–295. DOI: 10.1007/3-540-61794-9_66.

[2] Sehraneh Ghaemi, Mohammad Taghi Vakili, and Ali Aghagolzadeh. "Using a genetic algorithm optimizer tool to solve University timetable scheduling problem". In: *2007 9th International Symposium on Signal Processing and Its Applications*. Institute of Electrical & Electronics Engineers (IEEE), Feb. 2007. DOI: 10.1109/isspa.2007.4555397.

[3] A.R Mushi. "Tabu search heuristic for university course timetabling problem". In: *African Journal of Science and Technology* 7.1 (June 2010). DOI: 10.4314/ajst.v7i1.55191.

[4] Julio Fernández-Mendoza et al. "Nighttime sleep and daytime functioning correlates of the insomnia complaint in young adults". In: *Journal of Adolescence* 32.5 (Oct. 2009), pp. 1059–1074. DOI: 10.1016/j.adolescence.2009.03.005.

[5] Rhydian Lewis, Ben Paechter, and Olivia Rossi-Doria. "Metaheuristics for University Course Timetabling". In: *Evolutionary Scheduling*. Springer Science and Business Media, 2007, pp. 237–272. DOI: 10.1007/978-3-540-48584-1_9.

[6] Samir Ribiü and Samim Konjicija. "A two phase integer linear programming approach to solving the school timetable problem". In: *Proceedings of the ITI 2010, 32nd International Conference on Information Technology Interfaces*. 2010, pp. 651–656.

[7] S. Daskalaki, T. Birbas, and E. Housos. "An integer programming formulation for a case study in university timetabling". In: *European Journal of Operational Research* 153.1 (Feb. 2004), pp. 117–135. DOI: 10.1016/s0377-2217(03)00103-6.

[8] D. J. A. Welsh. "An upper bound for the chromatic number of a graph and its application to timetabling problems". In: *The Computer Journal* 10.1 (Jan. 1967), pp. 85–86. DOI: 10.1093/comjnl/10.1.85.

[9] E. K. Burke, D. G. Elliman, and R. Weare. "A University Timetabling System Based on Graph Colouring and Constraint Manipulation". In: *Journal of Research on Computing in Education* 27.1 (Sept. 1994), pp. 1–18. DOI: 10.1080/08886504.1994.10782112.

[10] Rakesh P. Badoni and D.K. Gupta. "A graph edge colouring approach for school timetabling problems". In: *International Journal of Mathematics in Operational Research* 6.1 (2014), p. 123. DOI: 10.1504/ijmor.2014.057853.

[11] Mohammad Malkawi, Mohammad Al Haj Hassan, and Osama Al Haj Hassan. "A New Exam Scheduling Algorithm Using Graph Coloring." In: *Int. Arab J. Inf. Technol.* 5.1 (2008), pp. 80–86.

[12] David Karger, Rajeev Motwani, and Madhu Sudan. "Approximate graph coloring by semidefinite programming". In: *Journal of the ACM* 45.2 (Mar. 1998), pp. 246–265. DOI: 10.1145/274787.274791.

[13] Ravindra Ahuja. "Very large-scale neighborhood search". In: *International Transactions in Operational Research* 7.4-5 (Sept. 2000), pp. 301–317. DOI: 10.1016/s0969-6016(00)00009-5.

[14] Marco Chiarandini et al. “An effective hybrid algorithm for university course timetabling”. In: *Journal of Scheduling* 9.5 (Oct. 2006), pp. 403–432. DOI: 10.1007/s10951-006-8495-8.

[15] A. B. Kempe. “On the Geographical Problem of the Four Colours”. In: *American Journal of Mathematics* 2.3 (Sept. 1879), p. 193. DOI: 10.2307/2369235.

[16] Mauritsius Tuga, Regina Berretta, and Alexandre Mendes. “A Hybrid Simulated Annealing with Kempe Chain Neighborhood for the University Timetabling Problem”. In: *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*. Institute of Electrical & Electronics Engineers (IEEE), 2007. DOI: 10.1109/icis.2007.25.

[17] Gary A. Kochenberger Fred Glover. *Handbook of Metaheuristics*. Kluwer Academic Publishers, 2003.

[18] R.W. Eglese. “Simulated annealing: A tool for operational research”. In: *European Journal of Operational Research* 46.3 (June 1990), pp. 271–281. DOI: 10.1016/0377-2217(90)90001-r.

[19] Halvard Arntzen and Arne Løkketangen. “A Tabu Search Heuristic for a University Timetabling Problem”. In: *Metaheuristics: Progress as Real Problem Solvers*. Springer Science and Business Media, pp. 65–85. DOI: 10.1007/0-387-25383-1_3.

[20] Luca Di Gaspero and Andrea Schaerf. “Tabu Search Techniques for Examination Timetabling”. In: *Lecture Notes in Computer Science*. Springer Science and Business Media, 2001, pp. 104–117. DOI: 10.1007/3-540-44629-x_7.

[21] Thomas A. Feo and Mauricio G. C. Resende. “Greedy Randomized Adaptive Search Procedures”. In: *Journal of Global Optimization* 6.2 (Mar. 1995), pp. 109–133. DOI: 10.1007/bf01096763.

[22] Samir Ribic. “A school timetable description language”. In: *2012 20th Telecommunications Forum (TELFOR)*. Institute of Electrical & Electronics Engineers (IEEE), Nov. 2012. DOI: 10.1109/telfor.2012.6419556.