University Class Scheduling Algorithms

Roy Lim

A thesis in fulfilment of the requirements for

COMP4121 Advanced Algorithms Major Project

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

October 2023

Abstract

University Class Scheduling Problem (USCP) is an important but difficult problem for universities due to the recent explosion in enrolments and courses. Currently, timetable creation is done manually at UNSW due to the number of both hard and soft constraints, tho UNSW has developed algorithms to assist the process. In addition, due to different universities having different circumstances, there is no commercial solution suitable for the problem yet. This project briefly analyzes and implements three different algorithms and discusses how they can be augmented to solve the problem. The algorithms in question are: Binary Integer Programming, Genetic Algorithms and Ant Colony Algorithm. After implementing the algorithms, we determined that Ant Colony Optimization was the most suitable candidate for a large-scale implementation due to its speed, flexibility, and portability. Further optimization is required before the algorithm is ready to be used.

Acknowledgement

I would like to extend my sincere gratitude to the following people, whose contribution made this project possible.

- Nicholas Pollock: For providing guidance on how UNSW approach timetabling from university level
- Jonny Rebolledo Moya: For recommending this project and continual support
- Zofia Krawcyzk: For providing an insider perspective to how timetabling is done on a school level
- Aleksander Ignjatovic: For teaching me algorithms within Computer Science for both COMP3121 and COMP4121

Contents

Abbreviations

CFCC	Clash-Free Course Combinations
GA	Genetic Algorithms
LP	Linear Programming
NP	Non-Deterministic Polynomial Time
UCSP	University Class Scheduling Problem

Chapter 1

Introduction

Timetable creation is a difficult problem due to the large number of both hard and soft constraints (making it an NP-Complete problem). The process is also highly specific to each university. As a result there is no commercial software that can reliably solve the problem as of yet (tho software that tackle this problem does exist).

In recent years, the number of undergraduate and postgraduate enrollments at UNSW has increased exponentially, with current over 1.5 million students enrolled in universities in Australia. UNSW has currently over 65,000 students and a research community of over 6000 [1]. With the growing number students, creating a reasonable and flexible timetable has become an increasing daunting task which is of utmost importance as bad timetabling can lead to classes at inconvenient times, potentially delaying graduation of students and reduce education quality.

The specific aim of this project is to learn about the different types of algorithms used to solve the problem, discover any hidden complexities that may not be apparent initially as well as implement some of the algorithms to assess their suitability to solving USCP.

The remainder of this thesis is organised like so: In Chapter 2 will introduce the necessary terminology, relevant algorithms as well as how the UNSW timetabling team have evolved

to tackle the problem. In Chapter 3, we will describe how the algorithms mentioned in Chapter 2 were adapted to create a timetable as well as justify any design choices that I made. In Chapter 4, I will describe any lesson I have learnt as well as evaluate the performance of the algorithms. In Chapter 5, I will discuss further research directions for USCP.

Our conclusion is given in Chapter 6, while the appendices discuss some interesting tangents I went on during my research.

Chapter 2

Literature Review

2.1 University Class Scheduling Problem

University Class Scheduling Problem (USCP) refers to the task of creating a timetable for all classes that a university is running in a given semester or trimester subject to a number of constraints. An event in this case just refers to anything students may attend (lectures, tutorials, labs, workshops e.t.c). This means each event must be assigned into a limited number of time slots and rooms.

There are two ways of classifying constraints.

The first way is classifying based on whether they are hard or soft constraints. Hard constraints refer to constraints that must be satisfied to create a valid schedule. For instance, for a given course, there must not be any two events that are running concurrently at the same time or academic staff preference (such as no 9am lectures as they need to drop their kids off at school). Soft constraints refer to constraints that are not necessary to create a valid schedule, but make it better. For instance, ensuring that there are multiple options for students to enrol in, such as having a morning and an afternoon tutorial for a class.

The second way is classifying based on their impact on the scheduling, which can be separate into event, course, and program constraints in increasing order of difficulty to schedule/how many other events they affect. Event-specific constraints refers to constraints that affect how an event can be scheduled such as what rooms are available (e.g. labs can only be done in certain rooms). This constraints tend to be hard (but not always, such as some events may prefer being in one room but doesn't explicitly require it) Course constraints refer to constraints that affect how events within a course can be scheduled. For instance, some courses require at least one lecture before a tutorial. Program constraint refer to how events between different courses need to be scheduled. For instance, a first year engineering student may enrol in the first year math and physics course at the same time. A good timetable should ensure then that the lectures, tutorials and labs for first year math and physics overlap minimally and complement each other (e.g. rooms should be spatially close to each other to minimise walking distance).

Here is a list of all the different types of constraints that may need to be accounted for when creating a timetable. At the end of each constraint, I also clarify whether the given constraint is hard (must be satisfied for a schedule to be valid) or soft (nice to have but not necessary for a schedule to be valid).

Constraints and their examples:

- **Event**
  - Event must be placed in a room that can accommodate for it (i.e. capacity of room must exceed students enrolled in that event, labs must be scheduled in the lab rooms) (hard)
  - Event must be scheduled at a reasonable time (e.g. between 9am to 6pm, tho there are exceptions). (hard)
  - Lecturer or tutor must be available for this class (hard)
- **Course**
  - For a course, there must be only one event for students to attend at any given time (hard)
  - There should not be more than 1 lecture per day (hard or soft depending on course)
  - Handling one-off events (e.g. public holiday causing a lecture to be rescheduled) (hard, tho this is typically easily to manually complete)
  - Sequencing of events (hard or soft depending on course)
    - Certain events should be scheduling sequentially. For instances, labs should be scheduling consecutively to avoid equipment from being continuously packed up and redeployed
    - Certain courses require at least one lecture before tutorials
    - Certain courses require lectures to be sequenced, e.g. the first lecture is a 2hr lecture followed by 2 1hr lectures.

  - Spread (soft)
    * If there are few events, the number of days should be minimised students attend on-campus should
    * If there are many events, the events should be spread through out the week instead of packed in 1-2 days
    * If there are multiple streams for an event, then events should accommodate different types of students (e.g. having a morning class and an afternoon class for students)

- **Program**
  - Ensure that the student has time for lunch if events are scheduled both in morning and evening (soft)
  - There must be enough rooms to accommodate all the courses at a given time (this is a non-issue for most universities but is more prevalent for other scheduling problems with more restrictive constraints) (hard)
  - Timetable should account for other courses that students are likely to take at the same time (e.g. engineering students likely take math and physics concurrently). (soft)
    * Ensure that for multiple courses, there are multiple enrolment pathways/options for as student to choose (e.g. if there is one combination of lectures, tutorial and labs that a student can attend, this is a bad schedule)
    * Clashes between courses that are taken concurrently should be minimised
    * Classes should be spaced optimally spatially, meaning students walking time between classes should be minimised
    * Classes should be spaced optimally temporally, meaning there shouldn't be an overly long gap between events (e.g. no mandatory 9am lecture followed by a mandatory 6pm tutorial on the same day.

While the list above is by no means comprehensive, it serves to help understanding the complexity of the problem and why it has been difficult to automate so far. Another

aspect to consider is that scheduling tends to be university-specific, as each university has its own struggles, which we will discuss more in: 2.5. Hence, each algorithm has to be tailored for a university. Another problem is getting a reliable and comprehensive list of constraints, which may be difficult as certain human factors are hard to quantify.

2.1.1 Criteria For An Ideal Algorithm

To solve the problem, a scheduling algorithm should have the following attributes:

- Fast (performance should be high relative to a brute force approach or naive solution as the state space grows factorially)
- Extendable/Maintainable (adding an additional hard or soft constraint should be possible even with increasing number of constraints)
- Flexible (small changes need to be possible with the algorithms, e.g. a room becomes unavailable meaning we need to reschedule an event(s), the algorithm should not need to be re-run from scratch)
- Portable (results from one term should be applicable for another term to minimise computation times)

2.1.2 Complexity Theory

UCSP is considered to be an NP-Complete [2] problem with more than one valid solution. To explain what that means, we need to investigate how complexity in computer science is used to categorised problems.

Problems are classified based on how difficult they are to solve, i.e. whether an efficient algorithm exists. Certain problems cannot be solved efficiently, or in polynomial time. A P problem that can be solved if an algorithm exists such that the worst-case complexity

is $O(N^c)$ where $c \in \mathbb{N}$ and N is the input size, where the P stands for deterministic polynomial time.

More difficult problems are sometimes classified as NP problems, where NP refers to non-deterministic polynomial time. Non-deterministic means that the Turing machine can perform more than one action when subject to constraints or a set of rules.

For a problem X to be considered NP, it must satisfy the following:

- The problem can be solved by a non-deterministic machine in polynomial time
- The solution can be verified by a deterministic machine in polynomial time (this is known as a certifier)

An example for a certifier is determining whether an integer is prime or not, which can be done in $O(n)$ be check each factor (or $O(\sqrt{N})$ time with check only up to the first $\sqrt{N}$ factors).

An NP-complete problem X can be defined as (i) it is an NP problem $X \in \mathcal{NP}$ and (ii) for all problems in NP, we can reduce the any problem in NP to X ($Y \in NP$ and $Y \leq_{\mathcal{P}} X$) in polynomial time. It can be informally define as problems that are both NP-Hard (meaning it is difficulty as the hardest problem in NP) and NP.

Famous NP-Complete problems include Travelling Salesmen and SAT (Boolean Satisfiability Problem).

For difficult problems, there are two classes of algorithms, exact and approximate. Exact algorithms guarantee the optimal solution but are costly to run. Approximate algorithms trade optimality for speed. Approximate algorithms are sometimes referred to as heuristic algorithms.

2.2 Linear Programming

Linear programming refers to the technique where a vector $\vec{x}^{\intercal} =< x_1, x_2, ..., x_n >$ is chosen such that an objective function $f(\vec{x})$ is maximised. $\vec{x}$ is subject to a number of constraints, usually in the form of inequality. The general nature of the approach it make it ideal for many applications.

An example of this is maximise $f(x, y)$ subject to:

$$f(x, y) = x + y$$
$$0 \leq x \leq 1$$
$$0 \leq y \leq 1$$

It is clear that in this case that $f(x, y)$ is maximised for $x = 1, y = 1$.

To solve linear programming problems, there are a number of algorithms. One way is called Branch and Bound, which will be used later on.

Linear Programming can be further subdivided into many different types. For instance, binary integer linear programming refers to when $\vec{x}$ consists only of 0s and 1s.

2.3 Evolutionary Algorithms: Genetic Algorithms

Genetic algorithms is a type of algorithm that mimics the process of natural selection in order to solve optimization problems [3] [4] [5]. It is a meta-heuristic algorithm, meaning that while it cannot guarantee solution is optimal, it can reach a good solution with limited information or computation capabilities. This algorithm is useful as we can adjust it based on whether we want it to explore (find new valid solutions) or exploit (using existing solution and improving them incrementally).

To understand, what genetic algorithms do, we need to first understand some key terminology. A gene is some individual parameter that can be tweaked. This is also known as a genotype, or a symbolic representation for our solution. A phenotype is the actual solution to our problem that the gene is suppose to represent. A chromosome is a collection of genes that represent a solution to some problem. A population is a group of chromosomes.

An example, suppose we have a chromosome that represent when I at university. We can let chromosome = $[True, True, False, False, False]$, which means I am at UNSW on Monday, Tuesday but not on Wednesday, Thursday and Friday. Each element in the array is a gene and True/False is the genotype (symbolic representation of when I am at uni). The phenotype would be what the True/False represent, which is when I am university.

The goal of genetic algorithms is to improve the fitness of its population. The fitness refers to how desirable or optimal a given chromosome is and is some arbitrary value. For instance, suppose we wanted to make a string as close to "abcd" as possible. The string "abce" (one letter difference) will have a higher fitness than "wxyz" (4 letter difference).

To improve the fitness, there are two main operators: mutations and crossovers.

Mutations refer to when a gene(s) changes within a chromosome. For example, suppose we had a parent chromosome "abce". A mutation may cause the last gene to change to a different letter, such as "d", meaning we get "abcd", which has a higher fitness than before.

Crossover refers to splicing two parent chromosome together to form a child chromosome.

For example, suppose we have a parent chromosomes "abwx" and "xycd". A crossover may create "abcd" (which the first 2 letters are taken from the first parent and the last 2 are taken from the second parent to create a new chromosome), leading a higher fitness than before.

A general outline of the algorithm can be found below. Stopping condition can vary.

Algorithm 1 Genetic Algorithm Overview

```
1: function OPTIMISE
2:     Initialise a population of N chromosomes
3:     while Stopping Condition is not met do
4:         Create a breeding pool based on current population
5:         Create new set of chromosomes via mutating or performing crossover with breeding pool
6:         Replace old generation with N many chromosomes with highest fitness from new set
7:     end while
       return Chromosome with highest fitness
8: end function
```

Generally stopping condition denotes when we have found a solution with a high enough fitness or if solution cannot be improved further.

An example can be found in the code in PasswordGuesser:

```
# Guessor is given a string "Hello World" and will attempt to guess it
g = PasswordGuesser("Hello World")
g.guess()
```

In this sample implementation (inspired by [3]), the fitness is the number of correct letters in each guess and only one letter mutations were used.

```
Guess: EthkZJD!rQr   Fitness: 1
Guess: EtlkZJD!rQr   Fitness: 2
Guess: HtlkZJD!rQr   Fitness: 3
Guess: HtlkoJD!rQr   Fitness: 4
Guess: HtlkoJD!rQd   Fitness: 5
Guess: HtlkoJDorQd   Fitness: 6
Guess: Htlko DorQd   Fitness: 7
Guess: Htllo DorQd   Fitness: 8
Guess: Htllo WorQd   Fitness: 9
Guess: Hello WorQd   Fitness: 10
Guess: Hello World   Fitness: 11
```

Notice that the chromosome progressively becomes more closer to the provided string, as signified by the fitness.

To augment genetic algorithms, we can change a variety of parameters.

Parameters that can be adjusted:

- Population Size (in each generation/iteration, new chromosomes are generaeted using chromosomes in previous generation, probability of being selected to create new chromosome is proportional to fitness. A higher population size encourages more exploration)
- Mutation Probability (probability of a mutation occurring when creating new chromosomes)
- Mutation Size (how many genes in a chromosome gets changed per mutation)
- Crossover Probability (probability of a new chromosome being the child of two existing chromosomes)
- Crossover Points (how many partitions are created for the child and which parent they come from)

Ideal parameter choice:

- Population Size should be large enough to encourage exploration of different schedules but small enough to avoid expensive, duplicate computation every generation.
- Mutation Size should decrease as fitness increases (as we converge to an ideal solution, changes to the existing population should be minimised, this is used in another algorithm called Simulated Annealing)

Another way we can change the algorithm is by modifying how fitness is computed. There are other fitness function beyond what has been discussed but this provides a good summary of what was considered during my research.

- Linear (e.g. fitness = number of genes that are acceptable)

  - Linear fitness functions are straight forward to implement.

- Logarithmic

  - Logarithmic fitness function encourage exploration to avoid pre-mature optimization. This is because fitness doesn't grow as fast relative to linear or exponential fitness functions.

- Exponential

  - Exponential fitness functions tend to converge quickly due each constraint being satisfied adds more fitness than the previous.

2.4 Ant Colony Optimization

Ant Colony Optimization [6] refers to a category of algorithms that are based on the behaviour of ants and their ability to find shortest paths between their colony and some objective (usually food).

Lets start with real ants. Ant colony algorithms are derived from the behaviour of ant colony and their ability to coordinate with each other to complete complex tasks. Most ants have subpar eyesight, meaning if the ants are undirected, they will wander in random directions. Suppose that an ant is able to find food and wants to guide other ants towards it. Ant can leave pheromone on their trail on the path they traverse to alert ants to where food is. These pheromone will evaporate overtime as they are volatile. Suppose we had two paths to the same food source, with one path being longer than the other. The longer path will take longer to traverse, meaning the pheromones that ants place on that track will become fainter relative to the shorter path. Over time, this leads to more ants choosing the shorter path relative to the longer path [7], thus allowing ants to "discover" the shortest path between two points. This idea of using pheromones to denote optimal path is what ant colony algorithm is based on (refer to 2.1 for a graphical representation).

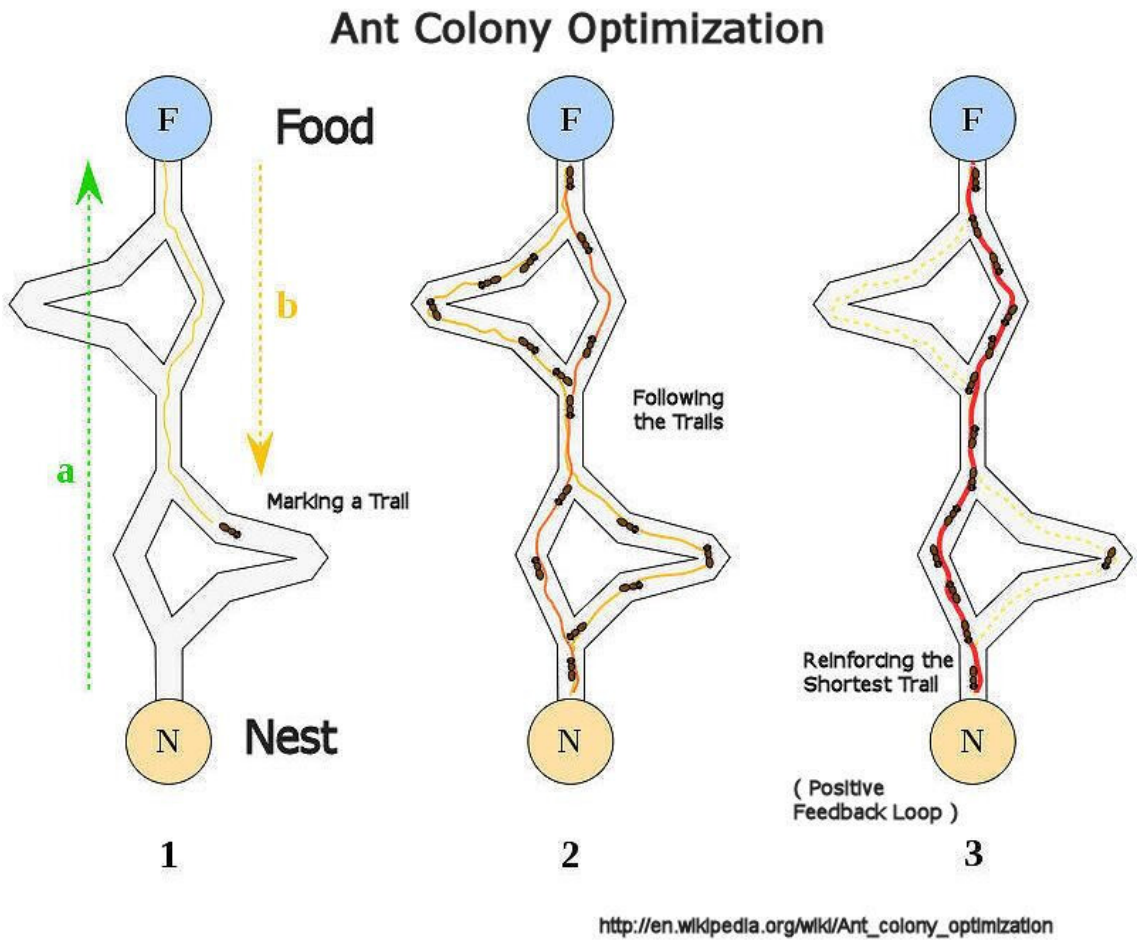

Figure 2.1: Visual Representation of Ant Colony Optimization

Ant colony algorithm is commonly used to determine the shortest path within a graph. Suppose we have an undirected, weighted graph G with $|V|$ many vertices and we want to determine the shortest path starting at vertex 0 without going over a vertex that has already been visitied (i.e. a Hamiltonian path). We can place an ant at vertex 0. The ant will then chose which vertex it would like to move to based on probability.

Let t be the time/generation in our algorithm. Let d_{ij} refer to the distance between vertex i and vertex j and enlightening information (parameter signifying distance) be $\sigma_{ij} = \frac{1}{d_{ij}}$ (this is sometimes called enlightening information and is used to compute the probability of traversing edge connecting vertex i and j). Let τ_{ij} refer to the amount of trail pheromones on the edge from vertex i to vertex j. Let α, β be arbitrary constants that signifying how important pheromone and distance are to our ant. Let S be set of vertices that the ant can travel to V - vertices that can't be reached - vertices that have already been visited before. We can let the probability that the ant moves from vertex i to vertex j at time t be:

$$P_{ij}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha}[\sigma_{ij}]^{\beta}}{\sum_{s \in S}[\tau_{is}(t)]^{\alpha}[\sigma_{is}(t)]^{\beta}} & s \in S \\ 0 & \text{Otherwise} \end{cases}$$

We will make a modification that now we can have multiple ants that are all independent on each other, let there be K ant in total. We will denote ants using k. Therefore $P_{ij}^{k}(t)$ is the probability that ant k will move from vertex i to vertex j at time t. Each iteration requires multiple ants traversing the same graph and spreading their pheromones.

Over time, the pheromones will evaporate since they are volatile (this is represented with evaporation constant ρ, $1 - \rho$ is the amount of pheromone left after the evaporation process). We want to ensure that the paths travelled by the ants have fresh pheromone added. Let each ant leave behind pheromone $\Delta\tau_{ij}^{k}(t)$ when they travel from vertex i to j at time t. The amount of pheromone is dependent on total pheromone Q left over the

length traversed by the ant L_k. Let:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}(t)$$

$$\Delta\tau_{ij} = \sum \Delta\tau_{ij}^{k}(t)$$

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} Q/L_k & \text{If ant } k \text{ has traversed from vertex } i \text{ to vertex } j \\ 0 & \text{Otherwise} \end{cases}$$

Algorithm 2 Ant Colony Optimization Overview

```
1: function OPTIMISE
2:     Initialise distance and pheromones
3:     Initialise shortest path found to be have a length infinity
4:     while Stopping Condition is not met do
5:         for Each ant k in [0,..., K − 1] do
6:             Generate path traversed by ant k based on current distance and pheromone
7:             if One of the path in this iteration is shorter than shortest path found then
8:                 Replace shortest path with new shortest path
9:             end if
10:            Update the pheromones
11:        end for
12:    end while
       return Shortest Path
13: end function
```

2.5 UCSP At UNSW

This section details how UNSW tackles scheduling as well as problems faced.

2.5.1 Scheduling At UNSW

Scheduling each term starts roughly 1/2-1/3 year in advance, whereby each school must submit an online form containing all the necessary information for the university timetabling team. This form may include:

- Class Size
- Duration
- Number of meetings (e.g. Lectures, Tutorials, Labs etc)
- Streams per meeting (e.g. a tutorial may have multiple streams/options that the student can choose from while a lecture typically only has one stream).
- Special request (e.g. put a lecture in a specific room, don't schedule lectures for this course before 10am etc)

The timetabling team then must manually schedule the classes (which takes 6 - 8 weeks). Once this is done, an email is sent out to the schools to confirm the schedules match the requirements and request any changes as necessary (this is often the case due to the long time-scale between when the timetabling requirements are sent and when the school receives it). This means the timetable is finalised around 2/3 trimesters in advance.

UNSW has a unique system with how student choose their classes. UNSW allows students to choose which events and at which times they attend given a variety of options. Other university, such as USYD or UTS, have a preference-based system where students provide a preference to their schedule and the university then assigns them accordingly. Since

UNSW allows for student choice, it forces scheduling to provide more enrolment pathways (i.e. students have more than one tutorial that fits their schedule) than would otherwise be needed if schedule were created with preferences given by students.

2.5.2 Issue With Existing Literature

The existing literature does not solve the more prevailing problem with class scheduling as the constraints tend to focus on room allocation (optimising utilisation rate) or optimising preference of students and lecturers. In practise, the complexity of scheduling classes often arises from many students enrolling in a small amount of courses concurrently (for instance, it is common for engineering students to simultaneously take the introductory math and physics course in the same term) (i.e. program constraints). This means to ensure that students are able to create a reasonable timetable, classes between the introductory math and physics courses must have minimal clashes between classes. While this is easy for two courses, this question quickly balloons when considering the number of degrees and courses taken each term. As a result, the temporal constraint of allocating a time for an event such as a lecture or tutorial is significantly more important than the spatial constraint of allocating a room. (I was only able to find one papers from 1970 that discussed the idea of conflict matrix, however this was for a high school context rather than a university context [8])

This could partially be explain by the shift in the size of universities, as in recent years, the number of buildings and room available has increased substantially, meaning finding a room is non-issue nowadays. Room allocation would be more important when numbers of rooms are more constrained (such as in a high school setting, although in high school, the number of courses are substantially lower than for a university).

To tackle this problem, UNSW timetabling team has developed an algorithm called Jane Austen Algorithm to generate Clash-Free Course Combinations (CFCC for short)). In total, around 15,000 CFCC are created based on enrolment patterns and program rules which are taken into consideration when the schedules are made.

In addition, another algorithm is used to "rank" how difficult each event is to schedule. This is dependent on a number of factors such as:

- How many streams (how many of the same events) need to be scheduled (e.g. Lectures usually have one stream while tutorials have many)
- How many other courses must be accounted for to avoid clashes (i.e. CFCC)
- Ideal course sequencing (e.g., some courses require tutorials to be after lectures, meaning it would make most sense to schedule the lectures first before the tutorials)

The exact algorithms are not public for security reasons.

Chapter 3

Application of Algorithms to USCP

This chapter is focused on describing the algorithms implemented in order to solve USCP and the rationale behind the design choices made.

3.1 Python

Python was chosen as the main language for implementing the different types of algorithms.

Advantages

Python has a simple syntax with a variety of syntactic sugar (such as destructuring) that make development faster. In addition, including packages to Python can be done easily through Pip or other software relative to other languages that were considered (such as C++ and Kotlin).

Python is commonly used in Data Science and Software Engineering (to create applications, web servers etc), making the skills that I learn on this project transferable to future

endeavors.

Disadvantages

Python is also know for its slow performance. Algorithms implemented will be considerable slower compared to if they were implemented in a language with better performance such as C++. However, there are ways of speeding up the performance (such as using C++ code that is wrapped using Python).

3.2 Linear Programming

Implementation can be found in linearScheduler.py and is inspired by [9] [10].

USCP can be interpreted as a linear programming problem where each variable is a binary integer [9] [10].

Let a single day be transformed into 2 hours blocks from 9am to 6pm (this means there are 4 blocks, 9am-11am, 11am-1pm, 2pm-4pm, 4pm-6pm). The time block where chosen to ensure 1 hour lunch break in the middle.

The following notation are also used:

- e Index for event $\{0, 1, 2, 3, ..., E-1\}$ (let E be the total number of events)
- d Index for day $\{0, 1, 2, 3, 4, 5\}t$ (1 corresponds to Monday, 2 to Tuesday and so on, $D = 5$ is the total number of days)
- t Index for time block $\{0, 1, 2, 3\}$ (there are $T = 4$ per day, T is number of time blocks per day)
- r Index for room $\{0, 1, 2, 3, ..., R-1\}$ (let R be the total number of rooms available)

We can then let the binary integer be:

$$x_{edtr} = \begin{cases} 1 & \text{Event } e \text{ is on Day } d \text{ at time block } t \text{ and in room } r \\ 0 & \text{Otherwise} \end{cases}$$

Hard constraints are included as constraints for the linear equation while soft constraints are included as a part of the objective function.

Let c_{edtr} be the preference or how optimal placing event e on day d at time t in room r is. This means the objective function (representing the soft constraints) to maximize is:

$$f(\vec{x}) = \sum_{e\in[0,\ldots,E-1]} \sum_{d\in[0,\ldots,4]} \sum_{t\in[0,1,2,3]} \sum_{r\in[0,\ldots,R-1]} c_{edtr} x_{edtr}$$

By default, value of c_{edtr} was set to 1. One such soft constraint implemented was lecture preferably placed in mornings (this was done by multiplying c_{edtr} by a factor of 1.5 if event e was a lecture).

Hard constraints that are implemented are implemented as inequalities.

- For any day, any time block and any room, there can be at most one event

$$\sum_{e} x_{edtr} \leq 1 \quad \text{For any combination of } (d,t,r)$$

- Each event can be scheduled a maximum of once.

$$\sum_{d}\sum_{t}\sum_{r} x_{edtr} \leq 1 \quad \text{For any event } e$$

- For any given course, there is at most one event scheduled at any given time. Let E' includes all events for a specific course (e.g. lectures, tutes, labs and workshops for Physics 1A) that event e is a part of.

$$\sum_{e\in E'} x_{edtr} \leq 1 \quad \text{For any arbitrary combination } (d,t,r)$$

- There are no more than one lecture per day for a given course: Let E'' include all lectures for a specific course that event e (note that event e is a lecture) is a part of:

$$\sum_{e\in E''} x_{edtr} \leq 1 \text{ For any specific day } d \text{ and any combination of } (t,r)$$

- Ensure that events are only allocated in rooms that are allowed:

$$x_{edtr} \leq 0 \text{ If event } e \text{ cannot be in room } r \text{ on day } d \text{ and time } t$$

This was implemented using Googles Optimisation Tool called Google Or-Tool [11]. To make this easier, each variable x_{edtr} is represented as an element in an array. This means for each event on each day, each time block and for every room, it must have a unique index.

Each event belongs to a course with a course index c, with a total number of C courses. Each course has a set number of events, which will be denoted as E_c.

The index can then be calculated using the following formula:

$$\text{index} = c \times (E_c \times D \times T \times R) + e \times (D \times T \times R) + d \times (T \times R) + t \times R + r$$

Some literature [10] recommends this algorithm be run twice, whereby the same options in the first round of optimization are excluded. Then we compare the first and second round optimization to see which schedule is better. This was not implemented.

In the total, the number of binary integers required is $E \times D \times T \times R$ (for each event, for each day, time, room it can be in).

3.3 Genetic Algorithms

Implementation can be found in geneticScheduler.py and inspired by [12].

The basic structure of the algorithm was outlined in 2, so we will discuss the aspects unique of USCP. The code is inspired by Jankovic [12] (Jankovic's code was written C++, which I then transferred into Python while augmenting to fit my use case). These include:

- How is a chromosome mapped to an actual schedule?

- How are mutation and crossovers performed?
- What is the stopping condition?
- How is fitness computed?

Let a chromosome represents a possible schedule. Each chromosome is an array of integers, where each index represents an event and integer in the array are a symbolic representation of the day, starting time and room of an event.

Let:

- D be number of days
- T be the number of starting times (e.g. 9am, 10am, etc).
- R be the number of rooms

To convert from a day d, time t and room r to an unique integer, I used the following formula

$$\text{symbolic representation for (day, time, room) allocation} = d \times (T \times R) + t \times R + r$$

Note that this records starting time and accounts for the duration of the event (i.e. algorithm will be able let any event start at any time between 9am and 6pm and account for duration, allowing for scheduling for events with different lengths will ensure that no events are overlapping).

Stopping condition for the algorithm is a) the best schedule that has been found so far has successfully scheduled each event and b), once a solution has been found, run the main loop (performing crossovers and mutations on the population) for 100 more iterations or number of iterations divided by 4, whichever is large. This ensure that even if a solution is found that the algorithm will continue searching for a better solution to avoid algorithm ending at a local maxima.

Parameters chosen for this genetic algorithm are:

- Mutation probability = 1 / 3 (probability that a new chromosome created is a mutated version of an existing chromosome)
- Mutation size = 1 (meaning for each mutation, randomly choose 2 events and choose new times for both)
- Crossover probability = 2 / 3 (probability that a new chromosome created is a child of two existing chromosomes)
- Crossover points = 1 (meaning for each crossover, the new chromosome consist of two partitions, one from each parent)
- Population size = 10

With every iteration, 2/3×population size are created via mutation and crossover before the top population are extracted. For crossovers and mutation, the probability of existing chromosomes being selected is proportional to their fitness (meaning good solution are more likely to be used to create the next generation).

For Genetic Algorithms, fitness function should either be linear or logarithmic.
Exponential fitness function would lead to early convergence to a solution that may not be globally optimal (even tho it seems to be local optimally) (for instance, an exponential fitness function may encourage a schedule where most but not all classes are scheduled correctly as this yielded the highest fitness early on but is in actuality may never converge to a solution where all classes are scheduled appropriately)

The fitness function implemented was linear (i.e. for each event that is scheduled with all hard constraints satisfied, the fitness is incremented by 1). If all events have been scheduled, then fitness function computed the bonus and adds log(bonus) to the fitness of the chromosome. A logarithmic bonus is used to encourage more exploration (as fitness increases slowly at first, reflecting the fact that one additional soft constraint isn't valuable as no soft constraints satisfied).

The conditions that were implemented:

- Hard Constraint
  - All events are scheduled
  - Lectures for the same course must be scheduled on different days
  - Events from the same course can't be scheduled at the same time
- Soft Constraints
  - Bonus point if the first event scheduled is a lecture for a course.

3.4 Ant Colony Optimization

Implementation can be found in antColonyScheduler.py and inspired by [13].

The implementation varies slightly from traditional Ant Colony Optimization algorithms as we instead change the problem to be a matching problem. Instead of an weighted, undirected graph, we instead have a bipartite graph with two disjoint sets of vertices.

Set 1 G_{event} represents all the events that need to be scheduled.
Set 2 $G_{availiabilites}$ represents all the available time slots that an event can be assigned to. Instead of a distance matrix, we will have a preference matrix which encodes any soft constraints (e.g. if we wanted morning lectures, we can adjust the preference to be higher).

Whenever we create a new schedule, we iterate through the events we need to schedule and chosen an edge connected it to a time slot in $G_{availabilities}$ that has not already been chosen. The probability of being chosen is proportional to pheromone and preference. Note that we break time into 1 hour blocks, so for one event there will be multiple edges connected to $G_{availiabilities}$ to encode the duration of the event.

Additionally, after the schedule has been created, we run a fitness function to evaluate the schedule to add any additional bonus (e.g. first event schedule is a lecture). The fitness function used is the same as one found in section 3.3. Note that fitness replaces

the distance as in the original iteration of ACO, the goal was to find shortest path while we are trying to determine the best schedule.

Parameters use:

- $\alpha = 1$
- $\beta = 1$
- $\rho = 0.2$
- $\tau_{max} = 10$
- $\tau_{min} = 1$
- Default preference = 10 (represents distance between the two disjoint sets in bipartite graph)
- Ants chosen to spread pheromone = 3
- Ants per generation = 10

Optimization that were made:

- Only the best ants in each generation can spread pheromone to speed up convergence to optimal solution (which was 3 in this implementation)
- A max (τ_{max}) and min (τ_{min}) pheromone were included to ensure that exploration is always possible (otherwise no new edge will be selected after a large number of iterations) (otherwise if $\tau_{ij} = 0$, that edge will never be used even if it is optimal).
- Pheromone is initialise initially to the max value (to encourage exploration and stop pre-mature optimization)
- When pheromone is spread, the increase is proportional to the difference between the max and current pheromone level ($\Delta\tau \propto \tau_{max} - \tau_{ij}(t)$).

- Evaporation constant is low $\rho = 0.2$ to prevent pre-mature optimization.

During this, the numpy package was used extensively to create pheromone and preference (distance) matrix as it was faster for numerical computation (e.g. evaporate pheromone can be done via a vectorised operation, making it faster and more efficient).

The conditions that were implemented:

- Hard Constraint
  - All events are scheduled
  - Lectures for the same course must be scheduled on different days
  - Events from the same course can't be scheduled at the same time
- Soft Constraints
  - Bonus point if the first event scheduled is a lecture for a course.

Algorithm 3 Ant Colony Optimization for UCSP

```
1: function OPTIMISE
2:     Initialise preference and pheromones
3:     Create a random schedule
4:     while Stopping Condition is not met do
5:         for Each ant k in [0,..., K − 1] do
6:             Create a schedule
7:         end for
8:         if Any newly created schedule has greater fitness than what was found before
   then
9:             Replace best schedule with new schedule
10:        end if
11:        Evaporate all pheromones by 1 − ρ
12:        Spread new pheromone on the path traversed by ants with the highest fitness
13:    end while
       return Schedule with best fitness
14: end function
```

Algorithm 4 Ant Colony Optimization for UCSP

```
1: function GENERATESCHEDULE
2:     for Event e in G_events do
3:         Randomly chose an edge connecting event to a time slot that has not been chosen in G_availiabilities
4:         Add any additional edges to encode the duration of the event
5:     end for
       return Schedule
6: end function
```

Chapter 4

Results

This chapter is dedicated to exploring the pro and cons of each algorithm that I implemented.

4.1 Limitations

4.1.1 Linear Programming

For linear programming, the approach has multiple limitations that prevent it from being useful. Some of the limitation is related to the implementation using Google-Or-Tools Integer Optimization.

- Algorithm is not scalable. While the algorithm is the fastest out of the ones implemented, this can be largely attributed to the small sample given to it to schedule and the fact that Google Or-Tool is highly optimized using C++ instead of Python which was used for the other algorithms. For large problems, linear programming is slow relative to heuristic algorithms.

- Algorithm is complicated. The mapping between unique index and an event is tedious to understand and allows for low number of abstractions.
- Algorithm is deterministic (can't generate random sample effectively)
- Algorithm is inflexible. Linear programming lacks an efficient way of schedule events with different duration at different start times (for instance, the approach can't schedule a 3 hour class since each time block is only 2 hours longs. The start time cannot be easily shifted either). In addition, only static constraints can be put in place (i.e, they must be set prior to the running of the algorithm, meaning we can't have constraints that change depending on what has already been allocated).

Some benefits include:

- Algorithm is fast for small input sizes. This suggest it might be more useful for other problems.

4.1.2 Genetic Algorithm

Genetic algorithm are limited by their concept. Some limitations include:

- Crossovers are not useful. Crossovers (whereby a new child chromosome is created using two existing, parent chromosomes) are not useful for this type of problem as the algorithm struggles when optimal solutions differ greatly, as is in the case of USCP.
- Algorithm output is not transferable. A solution generated for one semester/trimesters is not useful for the next since the whole algorithm needs to re-run if any input parameters change (e.g. what rooms are available).

Some benefits include:

- Algorithm is extendable. Genetic algorithm remains a good algorithm as it can be easily extended by changing the fitness function using a strategy pattern.

4.1.3 Ant Colony Optimization

Ant Colony Optimization was found to be the best. It could be seen as an improvement from genetic algorithms as randomly choosing time slots using crossovers and mutation, each choice is deliberate (using preference and pheromone to generate a probability of selecting edges).

Advantages of Ant Colony Optimization:

- Algorithm is faster. Choice of schedule is created based on probability rather than being purely random like with GA
- Algorithm is extendable. Adding additional criteria can be done easily by extending the fitness function or adjusting the preference matrix.
- Algorithm is flexible and portable. Since the algorithm uses pheromone and preference matrices, these can be easily adjusted to account for changes (e.g. room becomes unavailable). The algorithm can also use pheromone and preference matrices from previous runs, meaning that we can use subsequent runs are more efficient.

4.2 Recommendation

I would not recommend Linear Programming or Genetic Algorithms for USCP. Instead, I would recommend Ant Colony Algorithm for the various benefits I have described in the previous section. Notably, the algorithm ability to choose intelligently using pheromone and probability allows it to consistently outperform Genetic Algorithms in time required and scales well relative to Linear Programming for large problems due to its ability to explore the state space in an efficient manner.

4.3 What I Have Learn and Relevant To Future Career

During this project I have learnt:

- How to structure a research paper.
- Find, read and understand research papers.
- Python and its benefits (e.g. its Object Oriented aspects that made implementing the algorithms easier)
- Numpy and other useful packages in Python.
- How to ask for help and interview people.

Chapter 5

Future Direction and Optimization

This chapter is dedicate to outlining what I would recommend in future endeavors

5.1 Optimization for existing code

- More testing to determine which parameters work the best (this was skipped due to limited time).
- Implementation in another language with a focus on performance (such as C++ or Rust)
- More vectorized operation (e.g. with numpy)
- For Ant Colony Optimization, implementing a Segment Tree for lazy propagation of pheromone
- Order of scheduling events in Ant Colony Optimization could be dictated by Jane Austin Algorithm (i.e. ordering events based on difficult to schedule)

5.2 Other Promising Algorithm Candidates

- Particle Swarm Optimization
- Artifical Orca Optimization
- Simulated Annealing

Chapter 6

Conclusion

In conclusion, we were able to:

- Research and implement the following algorithms:
  - Linear Programming
  - Genetic Algorithm
  - Ant Colony Optimization
- Determine the ant colony optimization is the best choice out of the three

Appendix A

Interesting Topics

This chapter is dedicated to all the interesting topics that I found during my research project, but are not relevant to USCP and thus are placed here.

A.1 Graph Colouring

Graph Colouring describes a subset of problems created in the 1700s that looks at graph theory, which can be further seperated into vertex and edge colouring. Edge Colouring refers to the problem where in a graph, we can assign colours to each edge such that for a vertex, not two incoming edges have the same colour.

This can be used to determine how many exam time slots are required. Suppose we had a undirected, unweighted graph with vertices that represent courses. An edge exists between course A and course B if there is a student that take both courses. The number of unique colours required to colour this graph is the number of unique time slots required to allow for every student to take an exam with no conflicts.

References

[1] Jan 2023. [Online]. Available: https://universitiesaustralia.edu.au/university/unsw-sydney/

[2] M. Al-Betar and A. T. Khader, "A hybrid harmony search for university course timetabling," in *A hybrid harmony search for university course timetabling*, 01 2009.

[3] C. Sheppard, *Genetic algorithms with python.* Create Space, 2019.

[4] D. Shiffman, *The nature of code simulating natural systems with processing: Version 1.0.* s.n., 2012.

[5] TheAlgorithms, "The algorithms - python," https://github.com/TheAlgorithms/Python/tree/master, 2016.

[6] M. Dorigo and T. Stutzle, *Ant colony optimization.* MIT Press, 2004.

[7] S. Goss, S. Aron, J.-L. Deneubourg, and J. Pasteels, "Self-organized shortcuts in the argentine ant. naturwissenschaften 76: 579-581," *Naturwissenschaften*, vol. 76, pp. 579–581, 12 1989.

[8] B. L. Higginbotham, "Student scheduling: A solution method for the conflict matrix," 1970.

[9] A. Wasfy and F. Aloul, "Solving the university class scheduling problem using advanced ilp techniques," 01 2007.

[10] J. Samiuddin and M. A. Haq, "A novel two-stage optimization scheme for solving university class scheduling problem using binary integer linear programming," *Operations Management Research*, vol. 12, no. 3, pp. 173–181, Dec 2019. [Online]. Available: https://doi.org/10.1007/s12063-019-00146-8

[11] L. Perron and V. Furnon, "Or-tools," Google. [Online]. Available: https://developers.google.com/optimization/

[12] M. Jankovic, "Making a class schedule using a genetic algorithm," Jan 2008. [Online]. Available: https://www.codeproject.com/Articles/23111/Making-a-Class-Schedule-Using-a-Genetic-Algorithm

[13] R. Ge and J. Chen, "Analysis of college course scheduling problem based on ant colony algorithm," *Computational Intelligence and Neuroscience*, vol. 2022, p. 7918323, Aug 2022. [Online]. Available: https://doi.org/10.1155/2022/7918323