Yale-NUS College

An Application of Graph Theory in the Investigation of Scheduling Problems

Cai Lize, Eddie Lim, Wang Yanhua, Zhang Liu

April 17, 2020

Contents

1 Introduction **2**

2 Question 1 **2**
- 2.1 Question 1(a) . . . 2
- 2.2 Question 1(b) . . . 4
- 2.3 Alternative Method: Coloring . . . 5
  - 2.3.1 Question 1(a) . . . 5
  - 2.3.2 Question 1(b) . . . 6
- 2.4 Question 1(c) . . . 7

3 Question 2 **9**
- 3.1 Defining the Problem . . . 9
- 3.2 Analyzing the Problem . . . 9
- 3.3 Formulating the Model . . . 10
  - 3.3.1 Collecting Data . . . 10
  - 3.3.2 Formulating Assumptions . . . 10
  - 3.3.3 Representing the Data . . . 11
- 3.4 Solving the Model . . . 12
  - 3.4.1 Designing the Greedy Coloring Algorithm . . . 13
  - 3.4.2 Randomizing Orders for Greedy Coloring Algorithm . . . 14
- 3.5 Analyzing the Results . . . 16
  - 3.5.1 Without Common Curriculum Modules . . . 16
  - 3.5.2 With Common Curriculum Modules . . . 17
  - 3.5.3 Run-time and Other Observations . . . 18
- 3.6 But... Can We Do Better? . . . 19
  - 3.6.1 Introducing Flexibility . . . 19
  - 3.6.2 Implementation . . . 19
  - 3.6.3 Results . . . 21

4 Conclusion **23**

1 Introduction

Scheduling meetings can be messy since everybody has different commitments. In this project, we apply tools in the graph theory, such as $K-$complete graphs and coloring, to analyze and resolve meeting problems. We start by examining a fictional scenario and then use the insight to investigate a real-life situation.

In Question 1, we tackle a hypothetical scenario, involving ten representatives from the eight committees of a student government. We represent the committees as vertices of a graph but present two different approaches to the question: One finds and analyzes the larget $K-$complete subgraph and the other utilizes coloring of the graph.

In Question 2, we adapt the coloring method and apply it to a real life scenario — scheduling the modules offered in AY2019/2020 Semester 2 at Yale-NUS College. Based on our understanding of Question 1, we build a computer algorithm using Python to schedule the modules.

2 Question 1

The first question we tackle is to schedule weekly meetings for a hypothetical student government.

The said student government has ten representatives and eight committees. The representatives are distributed across the committees as follows:

Academics	Alessia, Halim, Mitsuko, and Xiuying
Athletics	Anjali and Xiuying
Commencement	Anjali and Xiuying
Events	Guanyu, Linh, and Mitsuko
External Communications	Halim and Xiuying
Finance	Israa and Sulaiman
Student Life	Guanyu, Halim, Israa, and Yeong-Su
Student Organizations	Anjali, Israa, Linh, and Sulaiman

In this hypothetical scenario, each committee is expected to meet two hours per week. Notice that all of the ten representatives are in more than one committee. That will cause scheduling conflicts as each representative needs to attend multiple committee meetings and thus those meetings cannot take place concurrently. In other words, any two committees that have members in common cannot hold their meetings at the same time.

In what follows, we use techniques in graph theory to solve several scheduling problems involving this hypothetical setup.

2.1 Question 1(a)

The first question we can ask is: What is the smallest number of two-hour sessions required to schedule all 8 committee meetings so that each representative is able to attend all of their assigned meetings?

Let $G = (V, E)$. We can represent each committee as a vertex on the graph G. We then add an edge between two vertices if and only if the two committees represented by those vertices *do not* have any members in common. In the later section, we will introduce an alternative method, which is adding edges between the committees that *have* members in common.

Following the rules above, we obtain the graph G as shown in Figure 1.

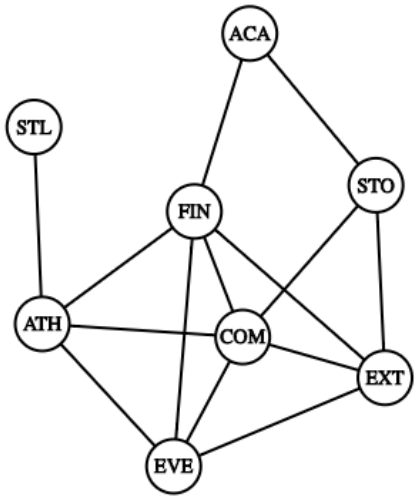


Figure 1: A graph representing the committees

This answers the question: How many committees can meet simultaneously?

That is not what the question is asking...

The question asks us to minimize the number of two-hour sessions required to schedule all the meetings. Thus, we need to fit as many meetings as possible into each two-hour slot.

Translating that problem into a graph theory problem, we need to find the largest K-complete subgraph in Figure 1, as all the committees represented by the vertices in a K-complete graph do not have members in common and thus can have meetings together.

By observation, we find that the largest K-complete subgraph in G is K_4 and there are two of them. One has the vertices {ATH, COM, FIN, EVE}, while the other contains the vertices {COM, FIN, EVE, EXT}. We will have to consider both of these cases. Why are these the only two?

Firstly, if we schedule Athletics, Commencement, Finance, and Events for the first two-hour slot, we can then remove those vertices and the edges connected to them from the graph G and obtain a new graph as shown in Figure 2a.

(a) After removing ATH, COM, FIN, EVE **(b)** After removing COM, FIN, EVE, EXT

Figure 2: Subgraphs containing the remaining vertices

This doesn't belong in the final report.

Note that now the vertex STL has a degree of 0, which means Student Life's meeting cannot be scheduled together with any of the remaining committee meetings. Additionally, {ACA, EXT, STO} do not form a K_3 graph. That means, we need at least two sessions for the three committees since they cannot all have meetings at the same time. Hence, we will need at least 4 two-hour sessions in total if we pick {ATH, COM, FIN, EVE} for the first two-hour slot.

On the other hand, if we schedule Commencement, Finance, Events, and External Communications for the first two-hour session, we can remove their vertices and obtain another graph as shown in Figure 2b.

It is clear that we only need another two two-hour sessions to schedule the remaining meetings — one session for Student Life and Athletics and the other for Academics and Student Organizations. Hence, we need three two-hour sessions in total if we pick {COM, FIN, EVE, EXT} for the first two-hour slot.

We can verify our answer when we introduce the alternative method.

Where?

How do you know you can't do better?

2.2 Question 1(b)

Next, we would like to find out what happens when we impose restrictions. Here, we investigate what happens if there are only three meeting rooms available at any given time.

That means for the first two-hour session, instead of choosing a K_4 subgraph in Figure 3, now we can only choose a K_3 subgraph.

Now, there are many options for the K_3 subgraph. It turns out, however, the choice does not change the final answer here. Regardless of which K_3 subgraph we remove first, we can always fit the remaining committee meetings into three two-hour sessions. Thus, the smallest number of two-hour sessions required is 4.

Your strategy only provides an upper bound. It does not guarantee a minimum.

In the following, we show one such example.

Firstly, we identify two distinct K_3 subgraphs as shown in Figure 1: One contains the vertices {ATH, EVE, FIN} and the other {COM, EXT, STO}. We will schedule those meetings for the first two two-hour sessions. Then, after removing those two subgraphs, we are left with just two vertices STL and ACA. Each of them

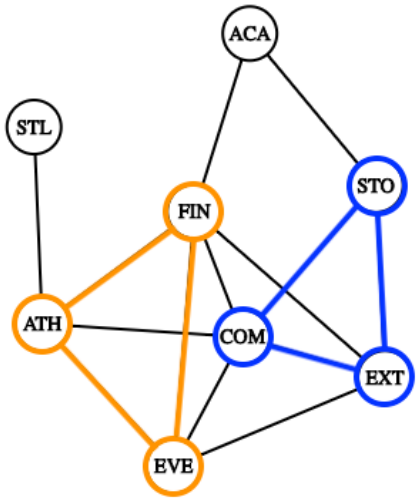


Figure 3: Two distinct K_3 subgraphs

has a degree of 0. Therefore, we need two more two-hour sessions for their meetings. As such, the total number of two-hour sessions required is 4.

Why are both methods essential?

2.3 Alternative Method: Coloring

As promised, we will introduce an alternative method of solving the previous two questions, which is by coloring the graph.

Coloring of graph works by assigning colors to all vertices of the graph such that no neighboring vertices have the same color. The chromatic number of the graph is the minimum number of colors required to color the entire graph.

We will still have the committees as the vertices on the graph. However, instead of adding edges between committees that do not have members in common, this time we add edges where they do. The resulting graph is shown in Figure 4.

2.3.1 Question 1(a)

If there are no restrictions, we can find the number of two-hour sessions needed to schedule all of the committee meetings by simply finding the chromatic number of the graph. The committees whose vertices are assigned the same color can hold their meetings in the same two-hour slot because the vertices are not adjacent to each other.

In Figure 5a, we show that the graph can be colored using 3 colors.

Consider the subgraph consisting of {ACA, COM, ~~EXT~~, STL} and the edges connected to them. This subgraph requires 3 colors. Thus, the entire graph requires at least 3 colors. Since we can color the graph with 3 colors, 3 is the chromatic number of the graph.

Why?

The coloring of the graph tells us we need 3 different two-hour sessions to schedule all of the committee meetings — the first one for Commencement, Finance, Events, External Communication, the second one

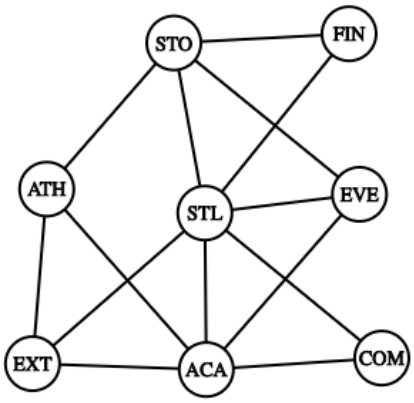
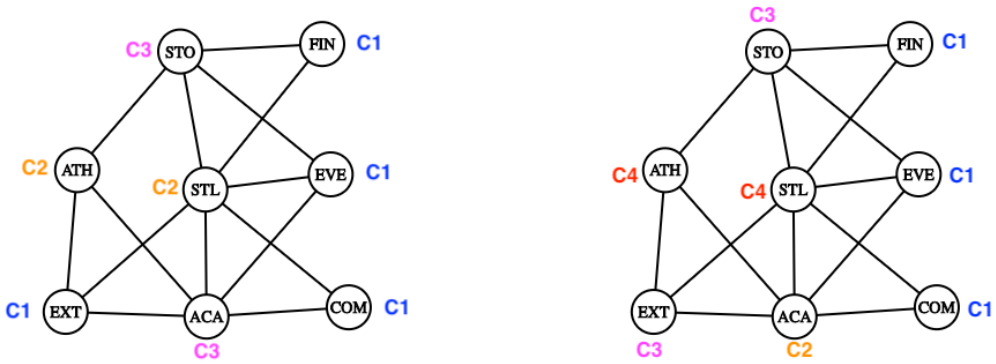


Figure 4: Adding edges where two committees have members in common

for Athletics, Student Life, and the third one for Academics and Student Organizations. The answer we obtained is indeed the same as our answer to 1(a).



(a) Coloring with no restriction **(b)** Coloring with the room restriction

Figure 5: Coloring of the graph

2.3.2 Question 1(b)

To answer Question 1(b), we need to impose a restriction on coloring, which is that one color can only be assigned to at most 3 vertices. This new rule ensures that at most 3 committees will be assigned to the same two-hour session.

Figure 5b shows the graph can be colored using 4 colors.

We can show that the graph can no longer be colored with 3 colors by considering the subgraph without the vertex ATH and the edges incident to it, as shown in Figure 6.

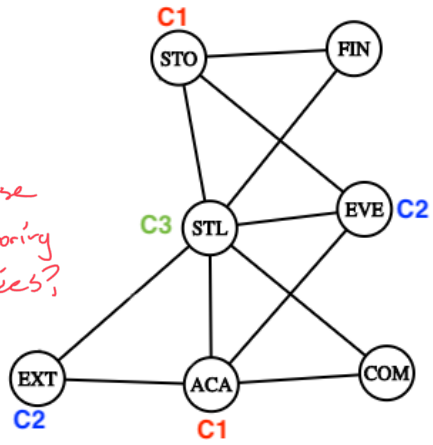


Figure 6: Subgraph after removing ATH

Consider the 5-cycle in the graph consisting of vertices {STO, EVE, ACA, EXT, STL}. We know that this cycle requires at least 3 colors since it is an odd cycle.

Now consider the vertices FIN and COM. FIN can be assigned C2, since it is adjacent only to C1- and C3-colored vertices. Now there are 3 vertices colored with C2, so COM, which is also adjacent only to C1- and C3-colored vertices, can no longer be colored with C2. It will be colored with a new color C4, instead. On the other hand, if we color COM with C2 instead, FIN will have be assigned C4. Thus, this subgraph requires at least 4 colors, which means the original graph cannot be colored with fewer than 4 colors. Since we can color the original graph with 4 colors, 4 is the smallest number of two-hour sessions required.

The coloring tells us that we can assign Commencement, Events, Finance to the first two-hour session, Academics to the second, External Communication and Student Organizations to the third, Athletics and Student Life to the fourth.

Although this coloring gives a different schedule from our previous answer, the smallest number of two-hour sessions needed is still 4.

While our previous method of finding the largest $K-$complete graph may seem more straightforward than coloring, we need to decide which subgraph to be removed at every step. The success of this method thus largely depend on good observation and judgement, which can be hard to verify. Coloring, on the other hand, is a more systematic and verifiable method that can be generalized to fit more situations.

With the next question, we will see the advantage of coloring in adapting to a different restriction.

2.4 Question 1(c)

We can further complicate the question by asking if the number of hours will be improved if we allow the meetings to be split into two one-hour sessions for each committee.

That means each vertex on our graph will get two colors. One for the first hour and the other for the second hour. Our answer to 1(b) is 4, which means in total $4 \times 2 = 8$ hours are needed to schedule all the meetings. Thus, any number less than 8 would be considered an improvement.

We find that we can double-color the graph with 7 colors, as shown in Figure 7.

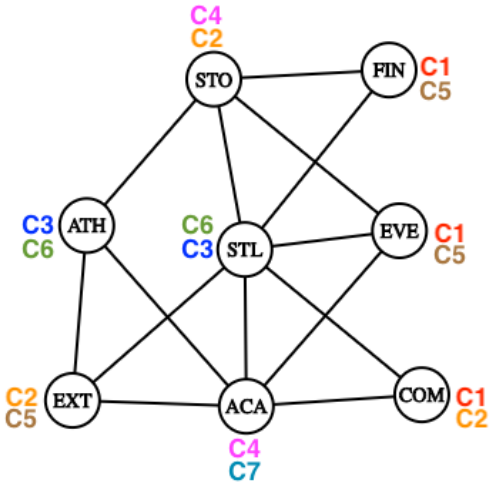


Figure 7: Double-coloring the graph

at most

That means in total we need 7 hours to schedule all eight committee meetings if we allow the sessions to be split into two one-hour sessions.

However, can we improve even more? That is, can we double-color the graph with fewer than 7 colors? We can answer that question by going back to the context.

Imagine a table for scheduling the meetings, as shown in Table 1. On the leftmost column, it is the time slots. On the right, we can fill in each cell with the committee that has meeting at the corresponding time. Each cell on the right represents a one-hour meeting slot.

Hour	Meeting Rooms		
1			
2			
⋮			
n			

Table 1: Scheduling table

First, we consider what is the fewest number of hours needed to schedule all committee meetings, ignoring all the scheduling conflicts. We know there are 8 committees and each has a two-hour meeting. In total, we need at least $8 \times 2 = 16$ one-hour meeting slots. However, there are only 3 meeting rooms available at any point in time. Therefore, we cannot schedule all of the committee meetings within 5 hours or less because $5 \times 3 = 15 < 16$ meeting slots. In other words, $n \geq 6$. ✓

Now consider what happens if $n = 6$. When $n = 6$, there are in total $6 \times 3 = 18$ one-hour meeting slots

available. If we want to utilize 16 of them, by the Pigeonhole principle, there can be at most 2 hours where only two of the meeting rooms are occupied.

The graph in Figure 1, however, shows neither STL nor ACA is part of a K_3 graph. That means, when either of these two committees hold their meetings, only one other meeting room can be occupied. Thus, there will be at least 4 hours where only two of the meeting rooms are occupied. Hence, we cannot schedule all of the committee meetings within 6 hours.

Since we have shown that we can schedule the meetings within 7 hours, 7 is the optimal answer.

3 Question 2

In this part, we will apply our knowledge and methods from Question 1 to investigate a real life scheduling problem. We will investigate problems involving modules offered at Yale-NUS College in AY2019/2020 Semester 2. For the list of modules, we refer to `https://yale-nus.instructure.com/courses/2861`.

Since the graphs we will be dealing with are much larger than the ones in Question 1, we will learn from our solution to Question 1 and create a computational model to simulate the coloring process.

In what follows, we adopt the development cycle for modeling learned from Scientific Inquiry 2 as a framework to approach the problem.

3.1 Defining the Problem

We begin by defining the scheduling problem of our interest.

The overarching question is: What is the maximum number of time slots we can remove from our current timetable while still having the same total number of modules?

In order to tackle this question, we divide it into the following parts:

1. Without the Common Curriculum classes, what is the maximum number of time slots can we remove from our current timetable while still having all the current modules?
2. With the Common Curriculum classes, what is the maximum number of time slots can we remove from our current timetable while still having all the current modules?

The reason for such a division is that the Common Curriculum is a unique feature of Yale-NUS College, where all the first- and second-year students take the same classes at the same time. As the Common Curriculum imposes some additional restrictions in our algorithm, we would like to investigate the simpler case first by removing the Common Curriculum classes.

What are the restrictions?

3.2 Analyzing the Problem

The questions we are investigating are essentially coloring problems.

What are the edges?

We can represent the modules as vertices in a graph and find the number of colors needed to color the graph. There are in total 188 modules offered in AY2019/2020 Semester 2, which means there will be 188 vertices.

Applying our coloring method, we add edges between vertices representing modules that cannot be scheduled together. We can find ~~the~~ number of colors needed to color the graph using a greedy algorithm.

The number of colors represents the number of time slots needed to schedule one session of all the modules. Under our assumptions, which are explained later, the total number of time slots needed in a week to schedule all the modules is two times the number of colors needed to color the graph.

an upper bound on

Why is this here?

3.3 Formulating the Model

3.3.1 Collecting Data

We collected data on the current modules and represented them in a spreadsheet with four columns: module code, level, instructor, major.

	A	B	C	D
1	Module Code	Level	Instructor	Major
2	YSC1212-1	1000	DANVY	MCS
3	YSC1212-2	1000	DANVY	MCS
4	YSC1216	1000	WERTZ	MCS
5	YSC2209	2000	FRANCESCA	MCS
6	YSC2213	2000	STAMPS	MCS
7	YSC2221	2000	RAZVAN	MCS
8	YSC2229	2000	SERGEY	MCS
9	YSC2232	2000	WERTZ	MCS
10	YSC2239-1	2000	VAN DEN BOOM	MCS
11	YSC2239-2	2000	VAN DEN BOOM	MCS
12	YSC2244	2000	BODIN	MCS
13	YSC3217	3000	BODIN	MCS

Figure 8: The head of the spreadsheet

Some module codes have an additional index attached to them, such as "YSC2239-1" and "YSC2239-2". The index is to indicate that the module has two different groups in a week.

For the Common Curriculum modules, we used additional indices to indicate the different timings and classes. The first index indicates whether the class is the AM (1) or PM (2) session. The second index indicates the class group. For example, "YCC1131-1-5" represents the Group 5 of Scientific Inquiry 2 in the A.M.

For cross-listed modules, we will only consider their primary majors.

3.3.2 Formulating Assumptions

To proceed with modeling, we will need the following assumptions to simplify our problem:

1. A weekday has 5 1.5 hour time slots, 0900-1030, 1030-1200, 1300-1430, 1430-1600, 1600-1730.
2. All modules have two 1.5-hour sessions per week.
3. The two sessions fall either on Monday and Thursday or Tuesday or Friday.
4. Within the same major, 1000/2000 level modules do not clash, while 3000/4000 modules do not clash.
5. If a module has more than one group, it can clash with other modules from the same level.

The second and third assumptions are necessary because not all classes currently follow the same time slots. Most of the classes are 1.5 hours in length and has two sessions per week. Classes such as YLC1201 Beginning Chinese 1 has 4 one-hour sessions per week, while YSC2213 Discrete Mathematics has two two-hour sessions per week. Thus, we standardize all the modules to follow the same schedule so that we can represent all of them as vertices of the same graph.

In addition, the third assumption implies that we just need to schedule the first session of all modules to obtain a weekly schedule, because Thursdays' and Fridays' schedules follow Mondays' and Tuesdays' respectively. As a result, we are also able to reserve Wednesday as a free day for students and faculties to hold meetings.

Following from the first and third assumptions, a week in our timetable has $4 \times 5 = 20$ time slots. Following from the second assumption, we only need to consider 10 time slots, since we only need to schedule the first session. Hence, to answer our 2 research questions, we can only remove slots from the current timetable if we obtain less than 10 time slots.

The fourth assumption is reasonable as the 1000/2000 level modules are designed mainly for freshmen and sophomores, while the 3000/4000 level modules are designed for juniors and seniors. Thus, it is less likely that somebody taking a 3000/4000 level module will also take another 1000/2000 level module from the same major.

The fifth assumption is reasonable. Since there is more than one session of the same module, interested students can select just one of the multiple sessions that do not clash with their other electives.

3.3.3 Representing the Data

As mentioned previously, we will answer the question using a graph and implement the coloring algorithm in Python. We will now define the variables in the context of our scheduling problem.

This is not a definition.

Definition

A **vertex** in the graph represents a module.

Definition

Two vertices are **neighbors** if the two modules which they represent have a clash in schedule.
Two modules clash if they meet either one of the following conditions:

1. they are from the same major *and* from the same level (1000/2000 or 3000/4000)
2. have the same instructor

Specific to Common Curriculum modules, each Common Curriculum module clash with:

1. all other Common Curriculum modules *unless* they have the same module code and belong to the same AM/PM session (e.g. Scientific Inquiry AM session).
2. other modules of the same instructor

The first condition of being neighbors follows from our assumption 3. The second condition is simply necessary. The same professor cannot teach two different modules at the same time.

With the data and assumptions, we can generate the corresponding graph using Python, as shown in Figure 9.

There should be a single definition for the "Module Graph".

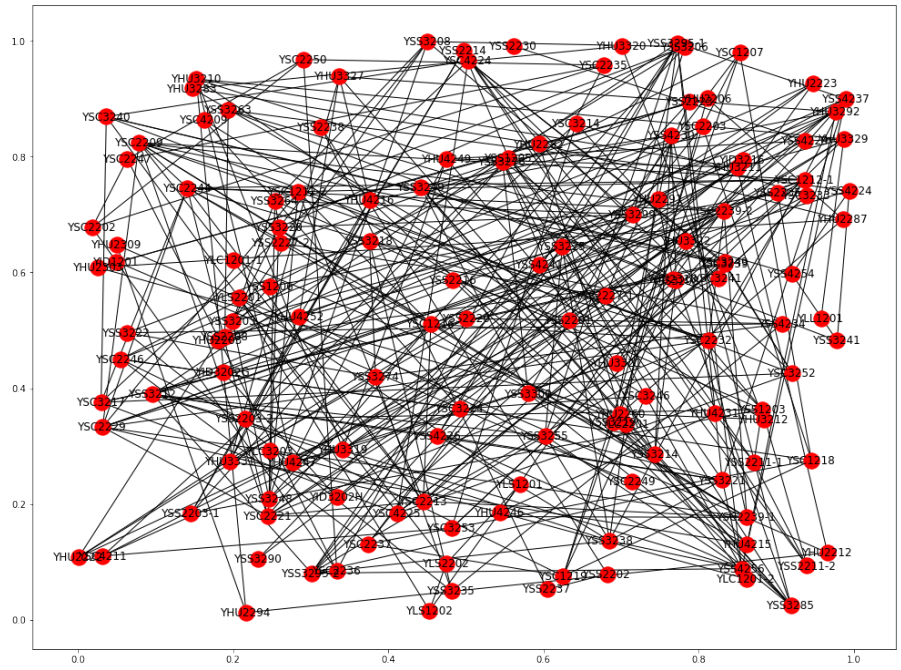
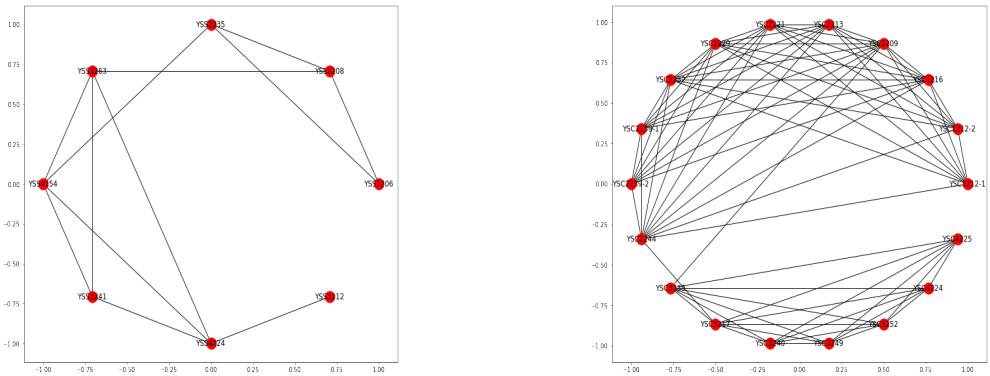


Figure 9: Graph of all modules, excluding CC

In what sense?

The graph of all modules is too complicated, so we also look at subgraphs of individual majors, as shown in Figure 10.



(a) Graph of PPE modules

(b) Graph of MCS modules

Figure 10: Graphs of individual majors

Notice that not all majors have the same number of modules.

In addition, modules across different majors only clash when their instructor is the same, which does not happen between any major, except when the major is 'CC' (our notation for common curriculum). Hence, the subgraphs of majors are mostly disconnected, and the number of colours we get depends mainly on the major with the most modules, i.e. MCS.

This is not true...

3.4 Solving the Model

To solve the model, we create a neighbor dictionary for each vertex.

Definition

A **neighbor dictionary** is implemented as a Python dictionary consisting of a list of *keys* representing the vertices and, corresponding to each key, a list of *values* representing the neighbors of the vertex.

Definition

An **order** is a list of numbers denoting the order of visit for each vertex during the coloring algorithm.

With the above definitions, we implemented the neighbor dictionary in Python. In this process, the input was the .csv spreadsheet obtained in the data collection step and the output was a dictionary, which is the neighbor dictionary storing the module information and their relationships (in terms of scheduling). The detailed implementation is shown in Appendix A.

We solved the model by implementing a randomized greedy coloring algorithm in Python.

3.4.1 Designing the Greedy Coloring Algorithm

First, we implemented the greedy coloring algorithm.

In essence, the greedy algorithm gives an order to the vertices in a graph $\{v_1, v_2, \ldots, v_n\}$. Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of colors. Each vertex v_i is colored with the first available (the lowest index) color c_j that is not used to color any of the neighbors of v_i with index lower than i. From our in class exercise, we learn that the greedy algorithm does not always give the chromatic number of the graph. Hence, we will randomize our order of the vertices to verify our result and do not claim that our result is the lowest possible number of colors.

For our model, the input was the neighbor matrix and the output was the a dictionary showing the coloring result, the specific color assigned to each vertex.

No outside resources allowed!

We adapted an algorithm from Wikipedia, which works as follows:

1. The `first_available` function serves as a helper function that takes in a list of colors (with each color represented as an integer) and returns the smallest non-negative integer not in the given list.

```
def first_available(color_list):
    # Return smallest non-negative integer not in the given list of colors.
    color_set = set(color_list)
    count = 0
    while True:
        if count not in color_set :
            return count
        count += 1
```

2. In the `greedy_color` function, we first initialise a dictionary which will store colors (represented as integers) as keys, and the nodes (modules) of that color as its values.

3. We then added the additional dictionary `color_count` to store the colors as keys and the number of nodes with that color as its value - this allows us to later limit each color to only have a maximum of 29 modules, as we only have 29 classrooms.

```
def greedy_color(G, order):
    color = {}
    color_count = {}
```

4. For each node in the order that we had specified, we then use a `for` loop to perform the following:

   (a) Obtain a list of colors used by the neighbors of the node

```
# Continuing in `greedy_color`

    for node in order:
        # Obtain list of colors used by neighboring nodes
        used_neighbor_colors = [color[nbr] for nbr in G[node]
                                if nbr in color]
```

   (b) Use our helper function `first_available` to determine the color of the current node.

```
        # Continuing in `for` loop
        color[node] = first_available(used_neighbor_colors
```

   (c) We then increase the count of the color returned in `color_count` by 1.

```
        used_color = color[node]

    if used_color in color_count:
        color_count[used_color] = color_count.get(used_color) + 1
    else :
        color_count.setdefault(color[node],1)
```

   (d) We use a `while` loop to help us keep track if the selected color's `color_count` exceeds 29. If it does, the this means that the current node cannot take on this color as there as not enough classrooms. Within the `while` loop, we then reduce the count of the selected color by 1 (back to 29). The selected color is then added to the list of neighbor colors, and then use our helper function `first_available` again to find the next available color for our current node. If the selected color's `color_count` is below 29, it then exits the 'while' loop, assigning the node the selected color.

```
    while (color_count.get(used_color) > 29) :
        used_neighbor_colors.append(used_color)
        color_count[used_color] = color_count.get(used_color) - 1
        color[node] = first_available(used_neighbor_colors)
        used_color = color[node]
        if used_color in color_count:
            color_count[used_color] = color_count.get(used_color) + 1
        else :
            color_count.setdefault(color[node],1)
```

5. Finally, after we complete the `for` loop, we return the dictionary `color`.

```
# Continuing in `greedy_color`
    return color
```

3.4.2 Randomizing Orders for Greedy Coloring Algorithm

As previously mentioned, greedy algorithm is affected by the ordering of vertices. As such, we implemented a function `random_color` to randomize the order of vertices for a given number of times. The random order will be passed on to the second parameter for the greedy coloring function that we previously defined. This allowed us to compare the number of colors required for all randomly generated orders. At the end of

Where is it?

this process, we inspected the results using a histogram, which shows the smallest and largest number of colors required.

We defined the `random_color` function with two parameters, G, denoting the graph to be colored, and I, denoting a given number of iterations of randomization.

```
def random_color(G,I):
```

In the function `random_color` , we first initialized the variables that will be useful later for storing and updating values based on the results computed in the greedy coloring algorithm.

```
    bestCount = 200000000
    worstCount = (-1)
    bestColoring = {}
    worstColoring = {}
    counts = []
```

This `for`-loop achieved the aim of randomizing the order of coloring. Given the number of iterations we desire, we can generate a random permutation for the list of vertices, by using the numpy module `np.random.permutation`. We then can pass the variable G (a dictionary variable) and this random order (a list variable) to the predefined function `greedy_color`. The returned value would be the specific coloring. To compute the number of colors used, we simply find the maximum index. Since the index starts from 0, we added 1 to this value and stored it in the variable `count`. We then compared `count`, the current "chromatic number" as computed by the greedy algorithm, with the best result, `bestCount` and the worst result `worstCount` so far. If the current value is better/worse, `bestCount`}/`worstCount` will get updated, respectively. The list variable `counts`, however, will store results from all random orders.

```
for i in range(I):
    random_order = np.random.permutation(list(G.keys()))
    coloring = greedy_color(G,random_order)
    count = max(list(coloring.values())) + 1
    counts.append(count)

    if count < bestCount:
        bestCount = count
        bestColoring = coloring
    if count > worstCount:
        worstCount = count
        worstColoring = coloring
return counts, bestCount, worstCount
```

Lastly, in order to visualize the performance of the randomized greedy coloring algorithm, we plotted the histogram of the values stored in the list varibale `counts` during the coloring algorithm.

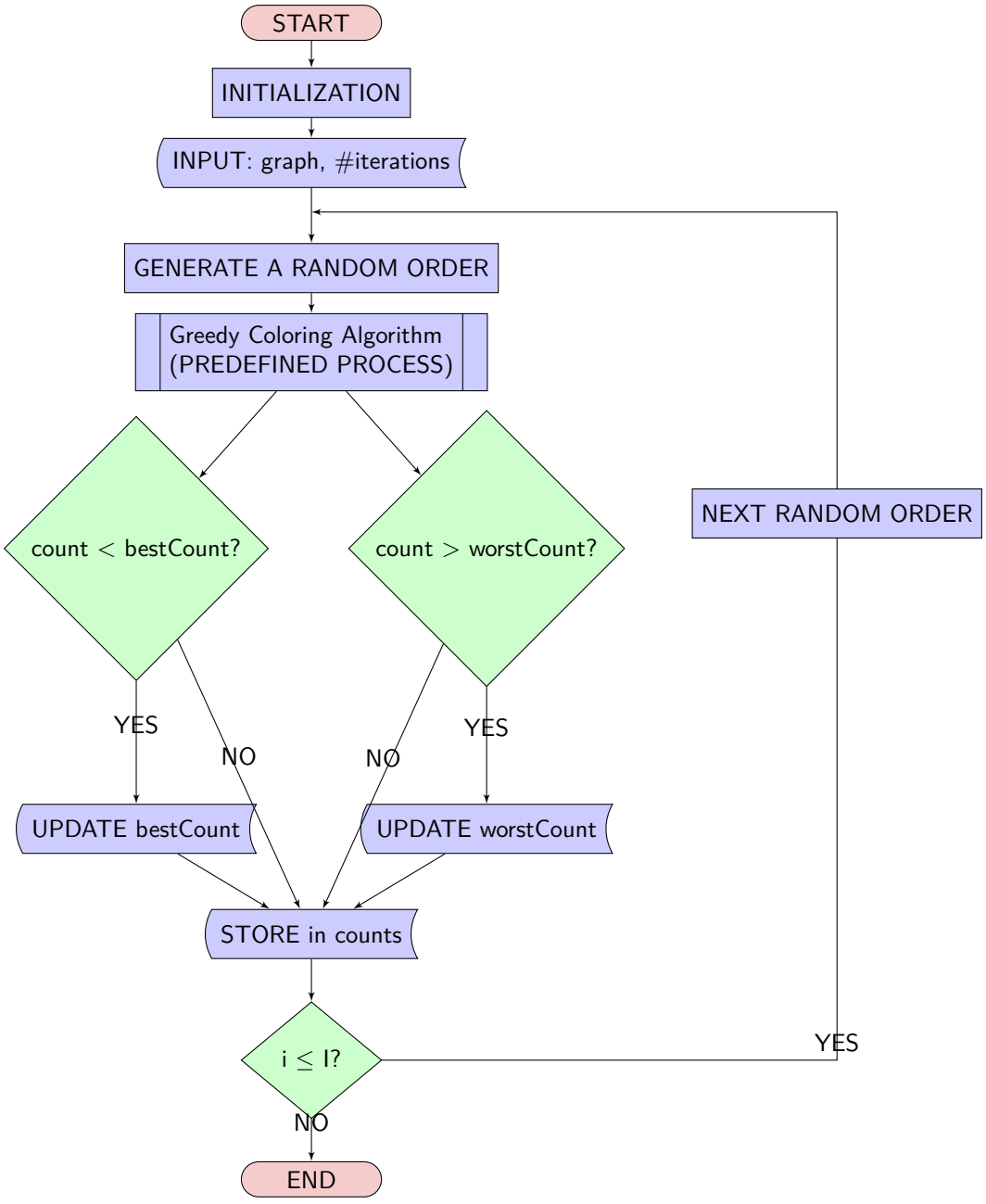
Where?

```
from collections import Counter

counts, _ , _= random_color(dic, 10000)
number_of_unique_values = len(Counter(counts).keys())

plt.hist(counts,bins = number_of_unique_values)
plt.show()
```

Our entire modeling process can be illustrated in the flowchart below.



3.5 Analyzing the Results

In this section, we will discuss the results of the model.

3.5.1 Without Common Curriculum Modules

We first solve the model using a naive ordering of the modules based on the order in the csv file which orders the modules based on major. This gives us the following result:

It is still not clear why the CC modules needed to be separate.

You need to restrict the number of rooms if your are not including the CC mods.

```
The number of colours is 10
The number of modules in each colour:
29
29
27
18
13
10
7
4
3
2
The total number of modules is 142
```

Figure 11: Composition of the modules across colors

We then further compared the coloring generated from the specific order where the the vertices were grouped together by major with the colorings generated from random orders. To do this, we plotted the histogram below showing the distribution of the number of colors used in 1000000 input orders. We can see that the order by major is already an optimal order for the greedy coloring algorithm because it is the same as the `bestCount`.

How does this compare to non-greedy coloring? Did you find a lower bound as well?

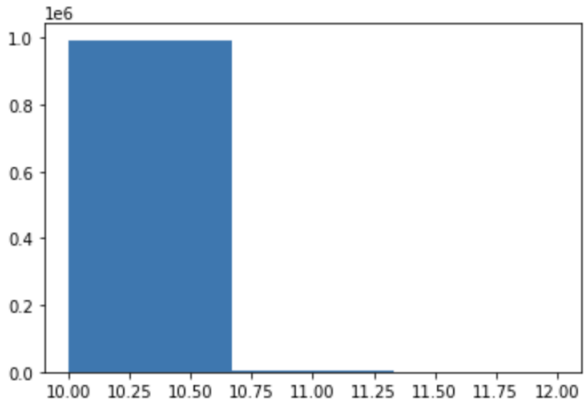


Figure 12: Histogram of number of colors required across 1000000 iterations

It is reasonable to conclude that 10 is likely the best outcome using the greedy coloring approach. (This does not mean however, that 10 is the chromatic number, as it may be still possible for a different order, or for a different approach to greedy coloring, to obtain a better coloring outcome.)

Since each colour represents 1 time slot, to answer research question 1, we are unable to remove any slots from the current timetable and the maximum number is 0.

3.5.2 With Common Curriculum Modules

Similarly, with the addition of Common Curriculum modules, we first solve the model using a naive ordering of the modules based on the order in the `csv` file which orders the modules based on major. This gives us the following result:

```
The number of colours is 10
The number of modules in each colour:
29
29
29
25
26
18
9
11
10
2
The total number of modules is  188
```

Figure 13: Composition of the modules across colors

Again, we then further compared the coloring generated with the colorings generated from random orders. This is done by plotting a histogram of the frequencies of the number of colors required across 1,000,000 iterations:

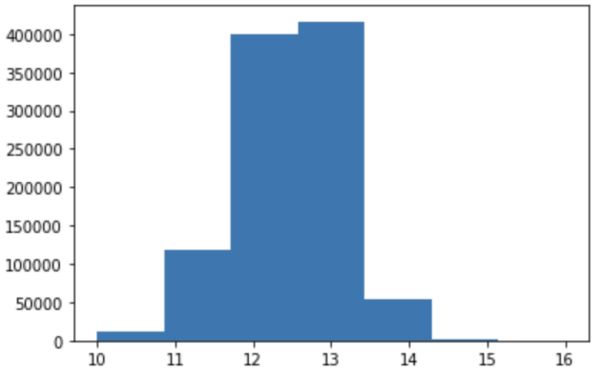


Figure 14: Histogram of number of colors required across 1000000 iterations

We find that the `bestCount`is still 10, showing that the order by major is already an optimal order. This may perhaps be not surprising as it is unlikely that the number of colors required will be less than 10 given our earlier findings without the addition of the Common Curriculum modules.

An interesting point in the histogram, however, is that the histogram seems to follow a normal distribution. This further points to the initial order by major as an optimal order.

Hence, to answer research question 2, we are unable to remove any slots from the current timetable and the maximum number is 0. However, it is worth noting that in our assumption, we have made Wednesday a free day, which is not the case in reality. Thus, our answer suggests that it is possible to leave Wednesday as a free day while still having all of the current modules.

3.5.3 Run-time and Other Observations

Here we comment on other aspects of our model.

Firstly, the model is time efficient because it took a very short run time to generate the coloring.

To obtain a valid coloring using a specified order, the model (with Common Curriculum modules) took 1.07 ms.

For a total of 1000000 iterations with random orders, the model (with Common Curriculum modules) took 9min 37s.

This means that our model could prove to be useful to the Registry because it can produce valid results within a time constraint.

Secondly, the model is applicable to the context. Our model is suitable because our assumptions are reasonable and hence is useful for scheduling the Yale-NUS modules.

3.6 But... Can We Do Better?

It is somewhat unsatisfying to find out we cannot remove any time slots. As an extension, we can examine our assumptions made earlier to assess the plausibility of further reducing the number of time slots required by allowing some modules to take place in the same time slot.

Specifically, our question is: What is the probability that one more time slot can be removed, if a given number of originally clashing modules are now allowed to take place in the same time slot?

This question also helps the Registry to assess if intentionally shifting modules is able to reduce the total number of time slots required.

3.6.1 Introducing Flexibility

In a realistic scenario, the assumption "Within the same major, 1000/2000 level modules do not clash, while 3000/4000 modules do not clash." which we have made earlier in the model above does not always hold. That is, some modules within the same major may be allowed to take place at the same time with other modules of a similar level. For example, students major in MCS are asked to choose a specific track of focus. Thus, a student taking YSC2229 Introductory Data Structures and Algorithms will not need to take YSC2232 Linear Algebra, so these two 2000-level modules can actually take place concurrently.

To investigate if this could in turn result in a fewer number of time slots needed, we thus introduce flexibility by randomly allowing originally clashing modules to take place in the same time slot with the exception that:

1. The same professor does not teach both of the selected modules.
2. Common Curriculum modules are not selected. (This is because sessions for Common Curriculum modules usually require all groups in the same session to take place at the same time slot.)

3.6.2 Implementation

1. To do this, we first define a helper function `remove_random_values`. The function picks a module at random, and removes a neighboring module at random. If the chosen module does not have any neighboring modules, or if the chosen module is a Common Curriculum module, we will pick another module at random instead.

   The function is defined as such:

```
def remove_random_values (fun_dic) :
    pick_random_key = random.choice(list(fun_dic.keys()))
    while (fun_dic[pick_random_key] == []) | (len(pick_random_key) == 11):
        pick_random_key = random.choice(list(fun_dic.keys()))
    values_from_random_key = fun_dic[pick_random_key]
```

```
    pick_random_value = random.choice(values_from_random_key)
    values_from_random_key.remove(pick_random_value)
    fun_dic[pick_random_key] = values_from_random_key

    return fun_dic, pick_random_key, pick_random_value
```

2. We then use a `for` loop to calculate how many neighboring modules we have to remove to obtain a better coloring count compared to our results above (10). The result for each iteration is sorted in the list `edges_removed`.

   For each iteration of the `for` loop,

   (a) We reset the dictionary by making a fresh copy of the dictionary `dic_cc` of the modules and their neighboring modules used earlier to obtain the result of a colors required count of 10.

   (b) We then reset the colors required count to 10.

   (c) Using a `while` loop,

      i. We make use of the helper function `remove_random_values` to obtain a modified dictionary with one value randomly removed.

      ii. We also make use of another `while` loop to ensure that the removed module does not breach the assumption that an instructor cannot teach two modules in the same time slot.

      iii. We then calculate the number of colors now required with the modified dictionary (the exit condition for the while loop is when the number of colors is 9 or less). It should be noted that this calculation is done using the greedy colouring model with the order arranged by major (which we have discussed above to be likely optimal).

   (d) Finally, we then count the number of edges remaining. We deduct the original number of edges from this number, and then append it to our list `edges_removed`.

   The code is written as such:

```
edges_removed = []

for i in range(100000) :
    modified_dic_cc = copy.deepcopy(dic_cc)
    min_color = 10

    while min_color > 9 :
        modified_dic_cc, random_key, random_value =
        remove_random_values (modified_dic_cc)

        while (str(data_cc["Instructor"][data_cc["Module Code"] == random_key])
        == str(data_cc["Instructor"][data_cc["Module Code"] == random_value])):
        # If the instructor is the same, we put the key and value back
            modified_dic_cc[random_key].append(random_value)
            modified_dic_cc, random_key, random_value =
            remove_random_values (modified_dic_cc)
            # print("Instructor same, picking another random node")

        modified_diclist_cc = list(modified_dic_cc.keys())
        coloring_cc = greedy_color(modified_dic_cc, modified_diclist_cc)
```

```
        colors_to_mod_cc = {}
        for key in coloring_cc:
            colors_to_mod_cc.setdefault(coloring_cc[key],[]).append(key)

        min_color = len(colors_to_mod_cc)

    count = 0

    for key, value in modified_dic_cc.items():
        if isinstance(value, list):
            count += len(value)

    edges_removed.append((2413 - count))
```

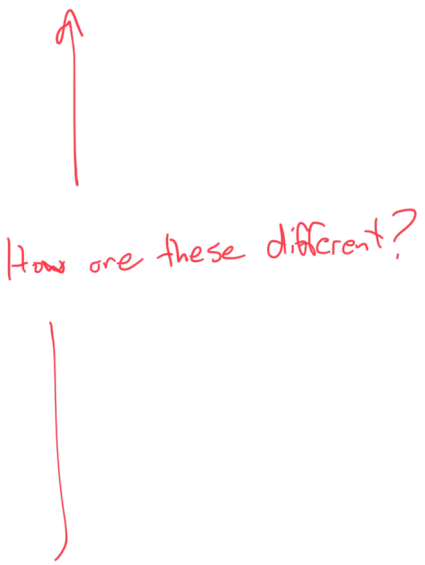
3.6.3 Results

We ran the `for` loop through 100,000 iterations and appended the number of neighboring modules removed in total to the list `edges_removed`.

In the table below, the left column indicates the a bound of number of neighboring modules removed to achieve a lower number of colors required (9). The right column indicates the number of iterations out of a total of 100,000 that meets the bound specified in the same row.

neighboring ~~Modules~~ Classes Removed	Count
<= 10	3563
<= 20	9109
<= 30	14610
<= 40	19191
<= 50	23041
<= 60	26337
<= 70	29568
<= 80	32918
<= 90	36537
<= 100	40594
> 100	59406

For a complete overview, we also plotted the distribution of the number of neighboring modules removed for 100,000 iterations in a histogram:



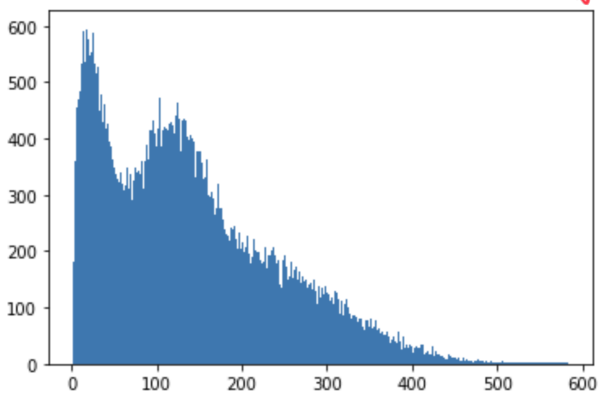


Figure 15: Histogram of successful reductions after edges removed

The x-axis of the Figure 16 is the number of edges removed, while the y-axis shows how many runs out of the 100,000 iterations result in a reduced number of time slots.

To interpret Table 3.6.3, we can consider a plausible scenario where Registry requests each major to make an accommodation for a single module within their major to take place in the same time slot as an originally clashing module.

This means that a total of 19 neighboring modules will be removed (considering languages as separate majors as well).

```
array(['MCS', 'PHY SCI', 'LIFE SCI', 'ARTS HUM', 'HISTORY', 'LIT',
       'PHILO', 'ANTHRO', 'ECONS', 'GA', 'PPE', 'PSYCH', 'URBAN',
       'SOC SCI', 'ES', 'CHINESE', 'LATIN', 'SANSKRIT', 'SPANISH'],
      dtype=object)
```

Figure 16: List of Majors

Reading the above table, even if we were to consider that one of the majors removed an additional neighboring module (bringing the total up to 20), only in 9109 out of 100,000 iterations (approximately 9.1%) were the numbers sufficient to achieve a lower number of colors required.

This allows us to draw three key conclusions:

Firstly, our earlier model with the assumption ""Within the same major, 1000/2000 level modules do not clash, while 3000/4000 modules do not clash." is likely to be applicable to the current context at Yale-NUS. Even though our model may be 'stricter', allowing some modules (<= 20) to take place in the same time slot is unlikely to affect the total number of colors required.

Secondly, from this result, the Registry can understand that requesting every major to allow a module to take place in a previously clashing time slot is unlikely to be worthwhile in the context of reducing the total number of time slots required.

Thirdly, this could, however, still be useful to the Registry if it is an imperative requirement to reduce the number of time slots. An alternative approach, for instance, by presenting the 9109 possible colorings to each affected major to choose from, can be considered.

4 Conclusion

In our project, we first solved a small-scale, hypothetical scheduling problem in Question 1. The process of finding the solution provided us with the theoretical insight into the application of graph coloring in solving scheduling problems.

Applying the same method to Question 2, we formulated and implemented a working, time-efficient, and applicable computational model using greedy coloring algorithm in the context of scheduling the Yale-NUS modules. We found with our original assumptions, it was not possible to remove the number of time slots. But since we found that we can schedule all classes with our current assumptions, it suggests that it is indeed possible to make Wednesday a free day for all.

We further experimented with relaxing the constraints of the model and considered its impact on the time slots required.

We believe our model can be modified for real applications.

References

[1] "Greedy Coloring." *Wikipedia*, Wikimedia Foundation, 12 April 2020.

[2] Miklós Bóna, *A Walk Through Combinatorics*, May 2011.