Scheduling Orals

A Thesis
Presented to
The Division of Mathematical and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Emily Osborne

December 2022

Approved for the Division
(Computer Science)

Charles McGuffey

Acknowledgements

My time at Reed has been such a long and complicated journey. It has been the best and worst times of my life, and I am endlessly grateful for every person who has played a role in my time here. I want to thank all of my friends who have given me their love and support throughout this endeavor. Your influence has dramatically shaped who I have become in the last four years and I am grateful for every second our stories have overlapped so far.

To my roommates, leaving the messy home we made together is by far the saddest part of being done at Reed. I love you all. Kai Hakomori, you were my first friend at Reed and you've always been there for me. A friend recently described love as someone feeling like you have solid ground in someone. You are my solid ground without which I could not stand. Thank you. Liam Ryan O'Flaherty, you are such a delight to be around. All this time we have been fighting to steal the shine, but the real shine is the friendship we forged in the battle :p . Alice Leask, I am so happy we got to become friends. Your passion for the things you love is constantly inspiring me. You approach everything with intensity including friendship and I am so grateful because I think it means we will be friends for a long time to come.

To Alice Barker, there is no way I would have tried computer science if you hadn't gotten me to sign up. No way I would have kept going if you hadn't encouraged me. No way I would have passed all my math classes if you hadn't been my tutor. Without your influence there is no way this thesis would be in Computer Science and I am so grateful for your support.

To my Advisor, Charlie McGuffey, thank you for being so patient with me during this thesis process. I really felt like you were on my team during orals and I appreciate your support.

To the Hackett family, you are my home away from home. I have honestly never felt more supported and welcomed than I do by all of you. Your generosity and sense of community is so inspiring and has shaped who I want to be.

To Sarah, my confidant, sister, and built in best friend, don't get kidnapped.

To Mom and Dad, this whole journey could not have gotten started without your love and support. You have always put me first and sacrificed a lot to prioritize my

education. I also know that Reed was not always your first choice for me but you were willing to support me wholeheartedly no matter where I wanted to go or what I wanted to do and I am so grateful for that. I love you both.

List of Abbreviations

CS	Computer Science
SMP	Stable Marriage Problem
NRMP	National Residency Matching Program
GCP	Graph Coloring Problem
FF	First Fit
LDO	Largest Degree Ordering
WP	Welsh Powell
IDO	Incidence Degree Ordering
DSATUR	Degree of Saturation
RLF	Recursive Largest First

Contents

List of Figures

Abstract

This thesis addresses the problem of how to schedule orals. First, I address how we should fairly match students with advisors for their theses. Then, I address how we can find which orals don't conflict with each other and can therefore be scheduled in the same time slot. I have also written the C++ code to implement this new method of scheduling orals so it can be used by the school.

My introduction section will further introduce the problem at hand, why it hasn't been solved already, and what other important applications there are for the matching and scheduling algorithms discussed in this thesis.

Chapter One will introduce the reader to some computer science basics that are a prerequisite to understanding the rest of this thesis. If one is already familiar with the computer science field, then they can likely skip Chapter One. I hope this section will make my thesis more accessible to readers who are not already familiar with the field of computer science.

Chapter Two will address the problem of matching students and thesis advisors. It will explain how the National Residency Matching Program algorithm works and why I have chosen to implement it. Namely, its stable matchings, strategyproofness, and runtime.

Chapter Three will address the matter of finding which orals can be scheduled simultaneously. I handle this by building a conflict graph of the meetings and then finding a Graph Coloring of this graph. Finding an optimal graph coloring for this type of graph is NP complete, so I employ an approximation algorithm to estimate an answer in polynomial time. This algorithm is called sequential coloring with a Largest Degree Ordering of the nodes.

Chapter Four will address my C++ code implementation of this project. I hope this chapter can be a bridge between the theory of my solution and my actual code. This chapter will also look at the runtime of my code on loads of different sizes.

The final chapter, Chapter Five, is a conclusion that will wrap up and address potential future work that can be done to extend this project.

Introduction

The problem we are looking to solve with this thesis is how to match students with thesis advisors and schedule their orals. Orals at Reed college, for those who don't know, are meetings where students defend their theses in front of a board. This board includes the student's thesis advisor, a second reeder from their department, a third reader from within their division but not their department, and a fourth reader from outside of their division. Each professor can be on several orals boards, but orals that have a professor in common cannot be scheduled at the same time because the professor cannot be in two (or more) meetings at once. This makes for a complicated web of connections.

The pairing of advisors and students in the Computer Science (CS) department is currently done in a meeting of the CS department professors. There is not a consistent method to the making of such pairings. Instead, they start with people who seem like obvious matches and then move on to ones that appear to be less obvious. Using a matching algorithm as a starting point for this kind of meeting would save professors some time and provide some guarantee of consistency and fairness. In a department meeting you have social dynamics at play, teacher preference is likely prioritized over student preference, and there is little or no transparency involved in the process. Using an algorithm would make the whole process more fair and transparent. My solution to the pairing problem is essentially to use the National Residency Matching Program algorithm. This algorithm is used for matching up medical students with residencies following graduation from medical school. The benefits to this algorithm include the following: it produces a stable matching; it slightly prioritizes student preferences; it is strategyproof for students, and it has a very reasonable runtime.

Next, I address the scheduling of orals. The scheduling of orals for the computer science department is currently not done by an algorithm but by the math department's Faculty Administrative Coordinator, Lisa Mickola. This works really well and has no fairness issues. The reason to change it would be to relieve Lisa of this task. At present, this is a complex problem that needs to be figured out anew every year; whereas the problem could be solved once via an algorithm that could be run not just this year but also in each successive year. My solution to the scheduling problem

is using a Graph Coloring algorithm called sequential coloring with largest degree ordering.

Of course, I am not the first person to think of using an algorithm for administrative scheduling problems. There are several software options that exist for solving this kind of problem already. When I asked at the registrar's office why this software is not already in use, I was told there was too high of a cost to switching. The software does have a price, but more burdensome than that is the task of inputting all of the necessary data into the required format and training people to use a new system. This small scale CS-department version of scheduling software that I have written would be easy for the CS department to switch to, and could be gradually scaled up to include more departments.

Ultimately, this project was successful in that the code written could be helpful in making the CS department more efficient. However, there is a lot more I would like to do to extend this project if I had the time. I discuss future extensions to this project in the conclusion in case any future Reed students would like to continue this project.

Chapter 1

Background

Chapter One is intended to serve as a reference to consult if and when a reader finds themself unfamiliar with a prerequisite concept at any point during their reading of this thesis.

1.1 Functions and Algorithms

An algorithm is a list of step-by-step instructions for how to complete a task. For example, if the task is making a sandwich, then the algorithm would be something like this: (1) lay out two pieces of bread, spreads, and sandwich contents; (2) put spreads on one of your two pieces of bread, (3) add the contents of the sandwich to the top of your first piece of bread, and (4) put the second piece of bread on top of the contents of the sandwich. Here, the algorithm has inputs and outputs. It has to take in bread, spreads, and sandwich contents as inputs as inputs. The inputs are the inputs are the items required for the algorithm to take place. The thing the algorithm produces is called an output; in this case, the sandwich would be an output.

When we implement algorithms in computer science we usually use "functions". A function is just a list of commands to which we give a name. We name functions so that we can define them once and then "call" or use them multiple times or with different inputs. The same way that it is easier to tell someone to "make a sandwich" than it is to tell them to do every step involved in making the sandwich, it is easier to ask the computer to do a predefined function than it would be to repeatedly tell it to perform every step required to perform that function.

We give the computer these instructions in a coding language instead of in english because english is not exact enough for a computer to understand. Often spoken languages can be interpreted multiple ways and their meaning relies on context. Computer languages are more precise. In this thesis I have written some code in a language

called C++, but in the written portion of the project I use what we call "pseudocode" to make the project more accessible.

1.2 Pseudocode

Pseudocode is a simplified version of computer code that is more like english. We often use pseudocode so we can show the steps of an algorithm clearly without requiring the reader to be familiar with a particular coding language. For the psuedocode in this project, I define functions with the following format:

```
def add_two_numbers(x,y):
      return (x + y)


add_two_numbers(2,3)
```

Figure 1.1: Pseudocode example

The term "def" indicates that I am defining a function. That is then followed by the function name. This function is called "add_two_numbers". The function name is followed by parentheses with variables representing the required inputs. There are two variables here, x and y, because this function needs you to input two numbers for it to add. The line that says "return (x + y)" is indented under the function definition because it is a part of this function. When we unindent all the way back to the left, we are done defining this function. The next line of code is "add_two_numbers (2,3)". This line is "calling" the function that we defined above. The computer doesn't execute the lines of code in a definition until that function is called. When the function is called we give specific inputs. Here the inputs are two and three so the function will add those numbers. The only function that happens without having to be called is the one named "main". When reading pseudocode later in this thesis, keep in mind that the computer will start from the function called "main" and then follow the instructions in the function from top to bottom.

An if/else statement allows us to execute certain code if a certain condition is met. This is written as the word "if" followed by a condition in parentheses and a colon. The condition should be something that is true or false. If the statement is true then the computer will execute the code indented under the if statement. If the condition is false, then the computer will skip this code. The if statement can be used on its own or with an else statement. An else statement just looks like the word

"else" and a colon. The code indented under the else statement will only happen when the condition in the corresponding if statement is false.

```
if (1 < 2):
    print("hello)
else:
    print("goodbye")
```

Figure 1.2: if/else statement example

In this example, the condition is that one is less than two. This is always true so the computer will print hello and skip the code that says to print goodbye. If the condition were false, like one is greater than two, the computer would skip printing hello and just print goodbye.

There are also two types of loops that it is important to know. Loops are just parts of the code that repeat. The for loop has the word "for" followed by a number in parentheses and a colon. This means you should repeat the code indented under this line the number of times that it says in the parentheses. For example, if you start with the number zero, then you have a loop that says for 4 times, add the number one. You add the number one four times and you should end up with the number four. We usually use these loops to repeat steps for each item in a list.

The other important loop is called a while loop. This is like the if statement in that it has a condition. This loop will start with the word "while" and then a condition in parentheses, then a colon. If the condition is true, the computer will execute the code indented in the while loop. When it is done, it will check the condition again. While the condition is still true, the computer will execute the lines of code in this loop again and again. The loop will end when the condition is false. Generally, the condition depends on a value that is changed by the code inside the loop so the condition becomes false and the loop does not repeat forever.

1.3 Graphs

A graph G = (N,E) is a set of points that we call vertices or nodes and a set of edges which connect pairs of nodes. Many real world situations can be modeled by graphs. For example, the nodes could represent facebook accounts with lines joining accounts that are "friends; "or the nodes could represent airports with lines joining the ones

with direct flights between them. When we model these problems in graph form we get to apply all of the research and proofs that have already been done about graphs to our problem rather than starting from scratch. Deo (1974)

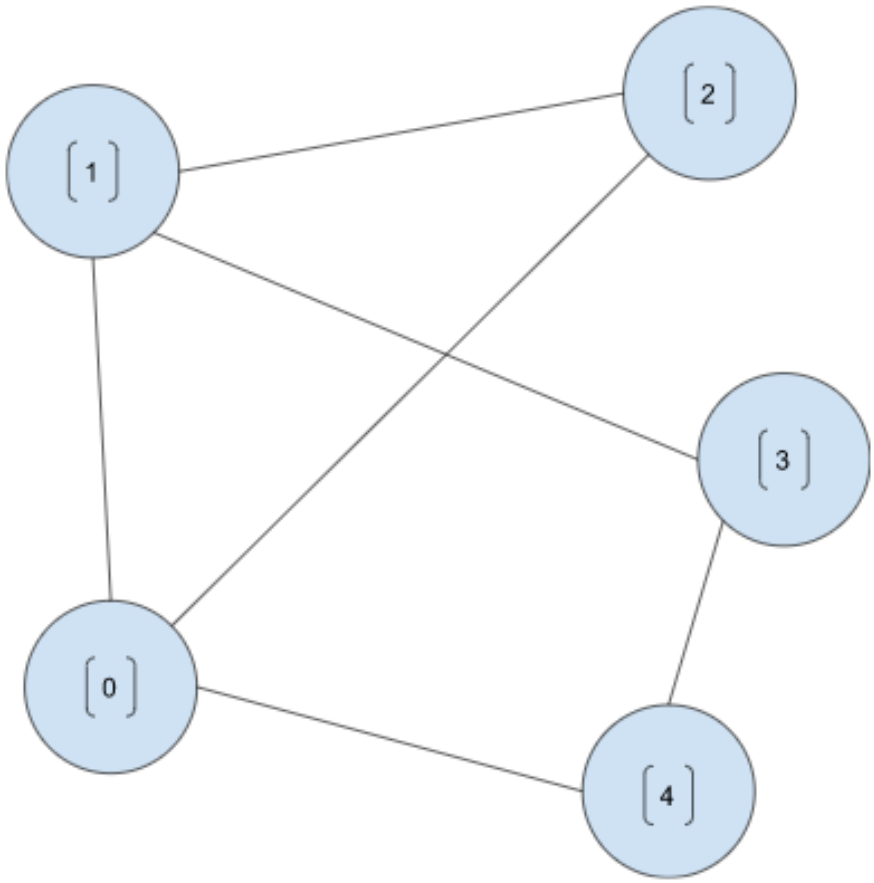

Figure 1.3: An example graph

In the example graph above, the nodes are the blue circles labeled 0 - 4. The edges are not numbered here but we can refer to them by the nodes they connect. For example, If I say the edge connecting 3 and 4, you know which edge I am referring to. The position in which I have drawn the graph does not affect how we interpret it. That is to say a graph is the same as long as its set of vertices and set of edges are the same. I could draw the edges longer or shorter or rearrange the vertices and it would not make a difference. The "degree" of a node is the number of other nodes to which it is directly connected. You could also say this is how many edges the node is attached to. For example, node 1 is connected to the nodes 0, 3, and 2, so it has a degree of three. Pictorial representations of graphs are helpful for understanding them, but in code we tend to represent graphs with numbers that represent which nodes are connected. The standard ways to do this are an adjacency list or an adjacency matrix. I use an adjacency matrix in my code. A matrix is a table of numbers. We refer to a number in a matrix by its row (how far down it is from the top) and then its column (how far over it is from the left). An adjacency matrix is an n by n matrix where n is the number of nodes. This means the matrix has n rows and n columns. Each entry in an adjacency matrix represents whether or not two nodes are connected by an edge. If I want to find if node i and node j are connected, I can look at row i and column j and see if there is a one or a zero in that spot. A one would mean the nodes are connected and a zero would mean they are not connected. We refer to the position in the matrix at row i and column j as matrix[i][j]. In the adjacency matrix, nodes are adjacent to themselves so matrix[i][i] will always equal 1 no matter the value of i. In the example graph above, nodes 1 and 3 are connected so matrix[1][3] = 1. However, nodes 1 and four are not connected so matrix[1][4] = 0. For this example graph, the complete adjacency matrix would look like this:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

1.4 Time Analysis

Time Complexity Analysis is how we estimate how long an algorithm will take to run. This is important because sometimes problems have solutions that work in principle but require too much time to actually be practical. Also, if there are multiple solutions that work, having an estimate of their runtime allows us to compare them and decide which one to use. In order to determine how much time an algorithm uses, we look at the number of steps that the algorithm takes on a particular size of

input. For example, if our algorithm is sorting a list of numbers, we need to find out how many steps are required to sort a list of that length. We can start by writing a pseudocode version of the algorithm. Then we build a runtime expression by adding together the number of steps it takes to get through the algorithm. What counts as one "step" is very loosely defined, but it might look like comparing two numbers or adding a number to a list. Each standard line of code counts as one step. When we get to a loop we think about how many times we would have to go through that loop. The number of steps within the loop is multiplied by the number of times the loop happens. When there are multiple branches where we can only go one way or another (e.g., if/else), we will only add the branch that has the longer run time since we can't possibly go through both branches. The number of steps will also depend on factors that are not related to the size of the input. For example, when we are sorting a list and our input is already sorted, it will likely take fewer steps than when our input is not sorted. Because of this we have best case, average case, and worst case analysis. This is where we look at the number of steps taken in the best case, average case, and worst case scenario. In this thesis, analysis will always be the worst case analysis. This means finding the most steps that we could possibly take on an input of size n. In the sorting example this might mean we consider the case where the input is in the exact opposite of sorted order. We generally do worst case analysis because it is more important to know if your algorithm has the potential to be unusably slow if you get an unlucky scenario than it is to know if it will be really fast in a lucky scenario. The reason we can so loosely define what counts as a step in the code is that we are going to use the expression for the number of steps for asymptotic analysis. Asymptotic analysis is just looking at things in terms of limits. We want to see how the runtime changes as n gets really big so we can categorize functions by their growth rate. This is part of why we do abstract analysis of these algorithms rather than writing code for them and then timing that code. Not only would timing evaluation depend on how well we wrote the code, the coding language we used, and the computer we ran it on; it would also be incredibly slow to run on big values of n. In asymptotic analysis there are basically three forms. Thomas H. Cormen (2009) An algorithm's Big O describes the upper bound on its runtime. An algorithm's Big Theta is an approximation of its runtime. And an algorithm's Big Omega describes the lower bound on its runtime. The formal definitions of these are as follows:

$g(x) \in O(f(x))$ iff there exists a positive real constant c and a positive integer n_0 such that $g(n) \leq cf(n)$ for all $n > n_0$

$g(x) \in \Theta(f(x))$ iff there exists two positive real constants c_1 and c_2 and a positive integer n_0 such that $c_1 f(n) \leq g(n) \leq c_2 f(n)$ for all $n > n_0$

$g(x) \in \Omega(f(x))$ iff there exists a positive real constant c and a positive integer n_0 such that $g(n) \geq cf(n)$ for all $n > n_0$

The most commonly used of these three is big O, so that is what I will be using in my thesis.However, big theta is a more accurate estimate of a function's runtime. To get a better idea of what runtimes mean, let's look at some big theta examples. If an algorithm's runtime is $\Theta(1)$, we say that algorithm is constant time. This would mean the amount of time the program takes to complete does not depend on the size of the input. If the runtime is $\Theta(n)$, then we call that linear time because it will form a straight line on a graph. This means the number of steps taken is directly proportional to the size of the input. I will often talk about things running in "polynomial time". This means that the algorithm's upper bound can be written as a polynomial expression of the size of the input. So the input size can be a base in the big Theta expression but it can't be an exponent. For example, $f(n) \in \Theta(n^2)$ is polynomial time, but $f(n) \in (\Theta 2^n)$ is not. What is considered an acceptable big Theta for an algorithm depends on the problem we are solving, but, generally speaking, taking longer than polynomial time is not acceptable unless you are only using very small inputs.

1.5 P and NP

In the previous section we were classifying algorithms by the time they take, in this section we are classifying problems by the amount of time their solutions take. There are many problems where we can find solutions in polynomial time, but there are also many problems where we haven't found such solutions. This could be because polynomial time solutions don't exist or it could be that we haven't found them yet. We categorize these problems into classes based on their complexity, the most relevant for this thesis being P and NP. Sipser (2006) P stands for polynomial time and NP stands for non-deterministic polynomial time. P is, formally, the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. What that means for our purposes is, if a problem is in P, we can find an answer for it in polynomial time. NP is the class of languages that have polynomial time verifiers. If something is in NP, we don't know how to find the solution in polynomial time, but if you have the solution you can verify that it is correct in polynomial time.

Chapter 2

Matching

In this chapter we want to find a way to pair up students with thesis advisors in a way that is fair for both the students and the advisors. Pairing up students and professors initially seems like a simple task. One could just divide the students up into equal sections and assign each group to a professor. Alternatively, you could make the professors like team captains at recess who get to take turns picking their top student from the pool of unpaired students. Or you could just assign every student to the professor they most want to be paired with.

These examples of matching algorithms bring us to the requirements for solving this problem well. Each student should end up paired with one professor, and we should be able to limit how many students end up with one professor. This would allow us to evenly distribute students if that is appropriate or allow professors to agree to taking on more or fewer students depending on their workload at the time. We also want to consider both the student and professor preferences to end up with pairings they are both happy with. In order to get student and professor preferences every student and every professor will submit a google form where they rank all of their potential matches from first to last.

In matching problems there are a few standard metrics to know. A matching is "envy free" if no student or professor would like to switch their match for someone else's. This would mean that everyone gets their top pick. This is not a reasonable standard to hold a matching algorithm to because it usually isn't possible. There is also a standard that we call "justified-envy free" or "stable." A matching is stable if no two elements prefer each other to their matched partners. If we are pairing up men and women, for example, a matching is stable if there are no men and women in separate couples who prefer one another to the partner they are matched with. sta (2012)

2.1 Stable Marriage Problem

For more information on stable matching let's look at the stable marriage problem. The stable marriage problem (SMP) is the problem of finding a stable matching between two equally sized groups of men and women given a list of each person's preferences. This problem was researched surprisingly recently, in the late 20th century by David Gale and Lloyd Shapely. The two mathematicians came up with the Gale-Shapley algorithm that proved that it is always possible to solve the SMP and make all marriages stable. Huang (2006)

In the first round, each unengaged man proposes to the woman he prefers most. Then each woman replies "maybe" to the offer she most prefers and "no" to any other offers. She is then provisionally "engaged" to the suitor she most prefers so far. In each subsequent round, first each unengaged man proposes to the most-preferred woman to whom he has not yet proposed regardless of whether the woman is already engaged. Then each woman replies "maybe" if she is currently not engaged or if she prefers this man over her current partner. In this case, she rejects her current partner and he goes back to proposing to other women. All matches are provisional and may change until everyone is paired. This process is repeated until everyone is engaged.

This algorithm will always terminate, everyone will end up married, and all of the marriages will be stable. However there is not just one way to get a stable matching. There may be several different stable matchings depending on people's preferences. For example, suppose we have three men (A,B,C) and three women (X,Y,Z) with the preference lists:

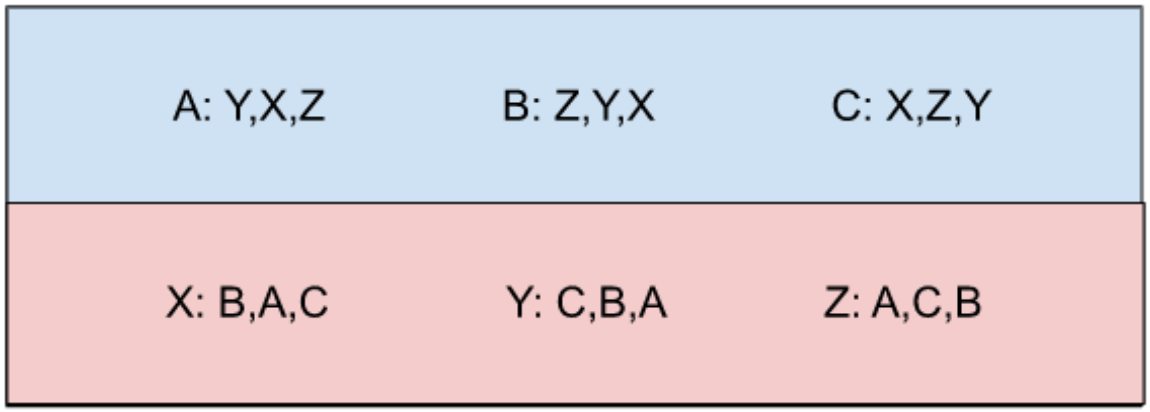

Figure 2.1: Stable Marriage example preference lists

There are three stable solutions to this matching problem. First, the men get their first choice and women get their third so the matchings are AY, BZ, CX. Second, all participants get their second choice so the matchings are AX, BY, CZ. Third, women

get their first choice and men get their third so the matchings are AZ, BX, CY. I think the second solution is the most fair, but there doesn't seem to be a standard metric in the math world that says why this stable matching would be better than the others. I suppose one could just say that it distributes the envy more evenly. In the Gale-Shapley algorithm described above you will always end up with the solution that prioritizes the men's preferences (the first solution). We could easily swap the gender roles and end up with the solution that prioritizes the women's preferences (the third solution). However, there is not a prevalent algorithm that will give us the second, more evenly balanced solution.

2.2 National Residency Matching Program

In 1984, Alvin E. Roth observed that essentially the same algorithm had already been in practical use since the early 1950s in the National Resident Matching Program. This algorithm is used to match medical students to residency positions, and this is the algorithm we are using to match students with professors. The real difference between the stable marriage question and the residency matching question is that the stable marriage problem is strictly for matching groups of equal sizes. There cannot be more men than women or more women than men. The residency matching question doesn't have this requirement. In residency matching there are generally more students than there are positions available so not every student gets matched. In our case there will be more positions available than students. That might seem counterintuitive because there are more students than professors but we can think about it as matching students with positions that are associated with a professor. The number of positions available is determined by what the professor lists as the maximum number of students they are willing to take on. Let's look at this algorithm in the context of matching students and professors.

The matching algorithm is "student-proposing." This means that it attempts to place a student with the professor that they most preferred according to their ranking list. If the student cannot be matched with their first choice professor because that professor is matched with other students who that professor prefers, then the algorithm attempts to place the student with their next best choice of professor. This continues until the student is tentatively matched or all of the professor choices have been exhausted without finding a match. We do not have to worry about a student not having a match because unlike the residency matching situation that this algorithm is used for, in our situation there are always more available spots than there are students. A tentative match will happen if the professor has an unfilled position available or if the student we are attempting to match is ranked more highly by the professor than one of their current tentative matches. If this is the case then the

lowest ranked student in the professors tentative match list will be bumped to make room for this new match. We will then attempt to match the bumped student with their next ranked professor. These tentative matches become final when all students rank order lists have been considered and everyone is matched. Just like the Gale Shapley algorithm, this algorithm will always terminate and the matchings will all be stable. E Peranson (1995)

2.2.1 Strategyproofness

An interesting metric to consider with this algorithm is its strategyproofness. An algorithm is strategyproof if each agent involved does not benefit from misrepresenting their preferences. In the NRMP algorithm the student can not get a better match by misrepresenting their preferences. This algorithm is even group-strategy proof for students meaning that no group of students can coordinate a misrepresentation of their preferences so that they all get a better match. It is possible for some groups of students to misrepresent their preferences in such a way that some of them are better off and others retain the same match. Of course, any strategy would require having everyone's preference lists which no student would have anyway. Professors, on the other hand, do have incentive to cheat. Professors can use individual strategy in misrepresenting their preferences to get a better match. I am satisfied with this level of strategyproofness because no stable matching procedure exists in which it is always the best strategy for all parties to be honest about their preferences. In fact, many of the other methods of finding a stable matching, like maximum weight matching, do not share this student strategyproof quality. University (a)

2.2.2 Time Complexity

Now let's talk about the time complexity of this algorithm. The runtime of the algorithm will depend on the number of students, which we will call n, and then number of professors which we will call m. The order of the rankings will also impact the run time, but we can just assume the worst case ordering for this analysis.

Let $f(n)$ be the maximum number of steps that our algorithm uses on any input of length n. At first glance, it is clear we have a for loop of size m inside a for loop of size n so we might think $f(n) = (n * m)$. But there are two factors complicating this. First, when we are trying to match a student with a professor that does not have any spots available we need to check a list of all the students they are matched with so far. Second, if the student matches and bumps another student, this could lead to a whole series of other bumps. For example, if everyone has the same favorite professor and our student order is opposite of the professor's rankings of them, we could match the first student, then the second student could bump the first, then

```
def main(n,m):
      for (n students):
            if(student is already matched): skip to next student
            for (m professors):
                  if (professor has empty space):
                        add student

                  else:
                        if (student is better ranked by this professor
                        than one of their current matches):
                              match this student and professor and go back to
                              finding a match for the student we just bumped
```

Figure 2.2: The pseudocode for the NRMP.

the third student could bump the second who could bump the first again and so on. There seems to be an inverse relationship between number of possible bumps and list of matched students. In order for each match to disrupt all previous matches every time, each professor must only have one available spot. That would mean the list of matched students to check would be length 1. On the other hand, if the list of matched students you had to check was as long as possible it would be length (n-1). That would mean the professor had matched with every student except this one before running out of spots. That necessarily means that only one bump happens that whole time. I want to look at the number of steps taken in each of these situations because I imagine that one of these edge cases will be the worst case.

First, let's imagine that we have n professors who are each accepting one student and the rankings are such that every match bumps every previous match. The first time we match the first student, they only need to check one professor of m. Then when they are bumped they check one and two. Then one, two and three. This sum continues up until m. The sum of numbers from 1 to m = $(m^2 - m)/2$. If this number of bumps were the same for each student then the number of step would be n times the sum from 1 to m. But the second student will be bumped one fewer times than the first. The third will be bumped one fewer times than the second. So really the number of steps should be the sum from 1 to n times the sum from 1 to m. Because the number of professors and students is equal in this scenario n = m so we can just write this expression in terms of n. So $f(n) = ((n^2 - n)/2)((n^2 - n)/2)$. This simplifies to $f(n) = (n^4 - 2n^3 + n^2)/4$

In the second scenario, we only have one bump but we have to search an (n-1)

length list. In this case we only have to search that list once because the last student is the first one where the professor ran out of spots. We also know that every other student must have gotten their first choice, otherwise some of them would be with other professors. So we have n (the number of students) times 1 (the number of professors they each had to look through + (n-1). I am adding the (n-1) instead of multiplying because we only had to do that once. Here $f(n) = n$ which is clearly not the worst case scenario so the first scenario must be the worst. This leads me to the conclusion that $f(n) = O(n^4)$.

Chapter 3

Graph Coloring

In this problem, we now have matched up students and professors and the next challenge is scheduling orals. In order to do this we need to group meetings that have no conflicts with one another and then assign those groups to time slots. We can solve this problem with graph coloring.

3.1 What is Graph Coloring?

As explained in the background section, a graph G = (N,E) is a set of points that we call vertices or nodes (N) and a set of edges which connect pairs of nodes (E). In this thesis, a "graph coloring," is really shorthand for a proper vertex coloring. A k-vertex coloring of G is an assignment of k colors: 1, 2, ..., k, to the vertices of G. In other words, if you color in all of the vertices of the graph, that is called a coloring. We represent the number of colors you used with the variable k. A coloring is "proper" if no two distinct adjacent vertices have the same color. That means that in a "proper coloring," no vertices that are connected by an edge are allowed to have the same color. So when I say "a graph coloring," I mean a labeling of the graph's vertices with colors such that no two vertices sharing the same edge have the same color. A subset of vertices assigned to the same color is called a color class. College

Graph coloring as a problem first came up because cartographers were thinking about how many colors were needed to color a map so that no two regions sharing a common border were the same color. This problem's history in map making is why we talk about it in terms of colors and not some other kind of labeling. Francis Guthrie, a South African mathematician who lived in the mid 1800s was the first to postulate that four colors would be enough to color any map in such a manner. This became known as the Four Color Problem and was passed around the math community at the time. A mathematician named Alfred Kempe published a paper that claimed

to have solved the four color problem and for a decade the problem was considered solved. Kempe was elected a Fellow of the Royal Society and later the President of the London Mathematical Society for his work. Then, about ten years later, Percy Heawood published a paper pointing out that Kempe's proof didn't work. But he had used Kempe's ideas to prove a five color theorem: that every planar map can be colored with no more than five colors. It wasn't until the 1970s that mathematicians Kenneth Appel and Wolfgang Haken finally proved the four-color theorem using a computer. This was the first major computer-aided proof. Graph coloring has been studied as an algorithmic problem since the 1970s.

The reason we are still studying Graph Coloring algorithms is that it is an np-complete problem. This means that we don't have a way to find the best coloring of an arbitrary graph in polynomial time. There are certain types of graphs that we can optimally color in polynomial time, but not most of them. The "best" or "optimal" coloring is one that uses the smallest number of colors possible to color a graph G. The smallest number of colors that can color graph G is called its chromatic number and is written as X(G). We also do not have a polynomial time algorithm for finding the chromatic number of a graph as this problem is np-hard. However, we do know the upper and lower bounds for the chromatic number based on the type of graph, number of nodes, and the degree of those nodes. The maximum chromatic number for any graph is the maximum degree of its vertices + 1. Again, the degree of a node is how many other nodes it is connected to. So the worst case scenario coloring-wise is that you go to color a node and every single node it is attached to is already assigned a unique color. You cannot assign this node any of those colors because it is adjacent to them, so you assign it the next available color. This means using degree + 1 colors. If this worst case scenario happens on the node with the largest degree, we get the upper bound on the chromatic number: maximum degree + 1 colors.

3.1.1 Examples

We care about finding polynomial time algorithms for graph coloring because it can be used to solve many types of real world problems. Graph coloring is applicable to fields such as chemistry, transportation, medicine, software design, register allocation, networks, economics, physics, etc. Let's look at a few examples to get a sense of how graph coloring applies to so many fields. University (b) Briggs (1992)

In chemistry, for example, let's say a company manufactures chemicals to sell, but certain combinations of these chemicals would cause explosions if brought into contact with each other. The company would probably want to partition its warehouse into separate compartments so that incompatible chemicals will be kept away from each other as a precaution. But how many compartments should they use and which chemicals should go in each compartment? We could make a graph where each node

is a chemical and then connect the nodes of the incompatible chemical pairs. Then we can color this graph. Any incompatible chemicals will be adjacent in the graph and therefore will not be the same color. So the number of colors used tells us how many compartments are needed and the chemicals that are assigned the same color should be in the same compartment.

In computer science, graph coloring can be used for optimizing register allocation. In processing code there are always lots of objects/variables to assign to registers. We could start by assigning each object to a separate symbolic register. Then we can perform a live range analysis. A variable is "live" from the time that it is defined until the last time the value is used. The information about when the variable is live could be used to construct an interference graph where nodes in the graph represent the symbolic registers and the edges tell us which symbolic registers are live at the same time. If two variables are live at the same time they cannot be put in the same register simultaneously. The coloring of this interference graph will tell us how many registers are needed and the nodes of the same color can be allocated to the same register.

Another real world problem that can be solved this way is allocating radio frequencies to the towers in a location. Towers that are located close together so that their transmissions will overlap need to transmit different frequencies. If we make a graph where the nodes are towers and they are connected if they are close enough for their transitions to overlap, then a coloring of this graph will give us independent sets such that every tower with the same color can be using the same frequencies without causing problems.

3.2 Applying Graph Coloring to our Scheduling Problem

Now let's apply graph coloring to our orals scheduling. In order to find groups of meetings that don't conflict with one another, we can build a "conflict graph." This is a graph where each meeting is represented by a node. One meeting has a student and two professors. This could be extended to four professors in the future. However, currently, in the CS department, students are matched with their advisor and their second reader and then they find their third and fourth readers on their own. So only using two professors does model the current problem that Lisa is facing every year. Next, we add edges connecting any nodes that share a professor. So, if a professor is in multiple meetings, those nodes will all be connected by edges. This is an example of a graph that might be generated with just 5 students and 5 professors.

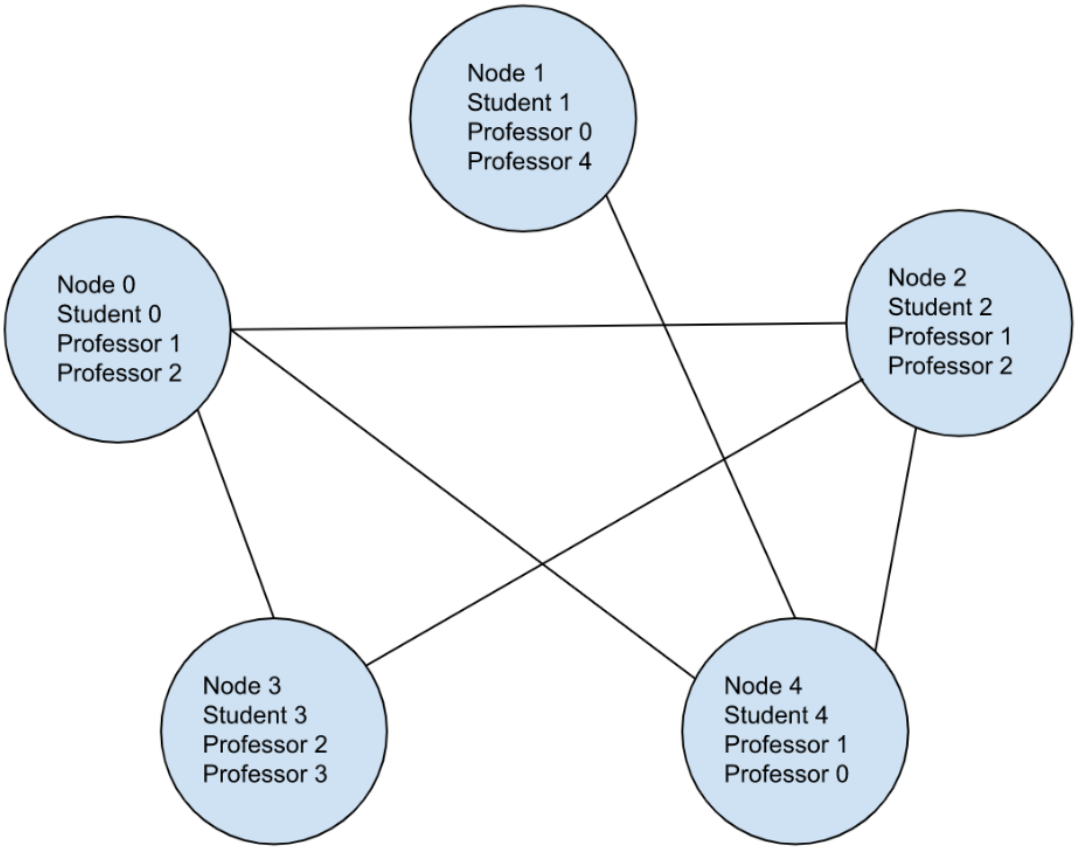

Figure 3.1: Example Conflict Graph

In my code, this graph would be represented with an adjacency matrix which would look like this:

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

From here the goal is to find a coloring of this graph. Because all of the vertices that have conflicts are directly connected (i.e. adjacent), no conflicting meetings can end up being the same color. Therefore, once we find a coloring, we can schedule all meetings of the same color at one time. So now lets look at how to find that coloring.

3.2.1 Sequential Coloring

In a perfect world we would like our algorithm to give us a coloring with exactly the chromatic number of colors in polynomial time, but this does not seem possible with the type of conflict graph our problem generates. As I mentioned earlier, there are some types of graphs that we can color optimally in polynomial time, but our graphs do not seem to fall into any of those categories. However, it is possible to get a good-enough coloring in polynomial time with a greedy coloring algorithm called the sequential coloring algorithm. A greedy algorithm is one that makes the optimal choice at each step in an attempt to find the optimal way to solve the entire problem. In the sequential coloring algorithm, we iterate through the nodes and check if they can be assigned to the first color option. This depends on if a node they are adjacent to has already been assigned that color. If a neighbor already has that color, we try the next color, if not, we use the color. Once a node has been assigned a color we move on to the next node until all of them are colored. Sysło (1989)

This is an approximation algorithm, meaning it can find an answer within some margin of error from the optimal solution. Our coloring just has to be good enough that k is less than our number of available time slots. This method is often mislabeled as “backtracking” online. The difference is that backtracking incrementally builds candidates for solutions and abandons a candidate as soon as it determines that it can not lead to a valid solution. Backtracking can be applied to problems that work with the concept of a “partial candidate solution” and have a quick test of whether it can possibly be completed to a valid solution. This requires one to define a specific number of colors that would count as a valid solution. In our case, we could use the number of time slots as the number of colors we are looking for and backtracking would find us a viable solution if it exists. When deciding which algorithm to use I went with sequential because it is faster and simpler while also being effective. But

now, looking back, I think backtracking would have been a better choice. Runtime is not really an issue for the scale we are looking at and it is nice that backtracking is not just likely to find a good solution, but is guaranteed to find one if it exists.

3.2.2 Node Order in Sequential Coloring

An important factor in how well this algorithm works is the order in which we color the nodes. If we follow the algorithm on an example, you can see how choosing the wrong order results in needing more colors. Again, the algorithm is just: go through the nodes in order, assign each node to the lowest color that is not already being used by a connected node.

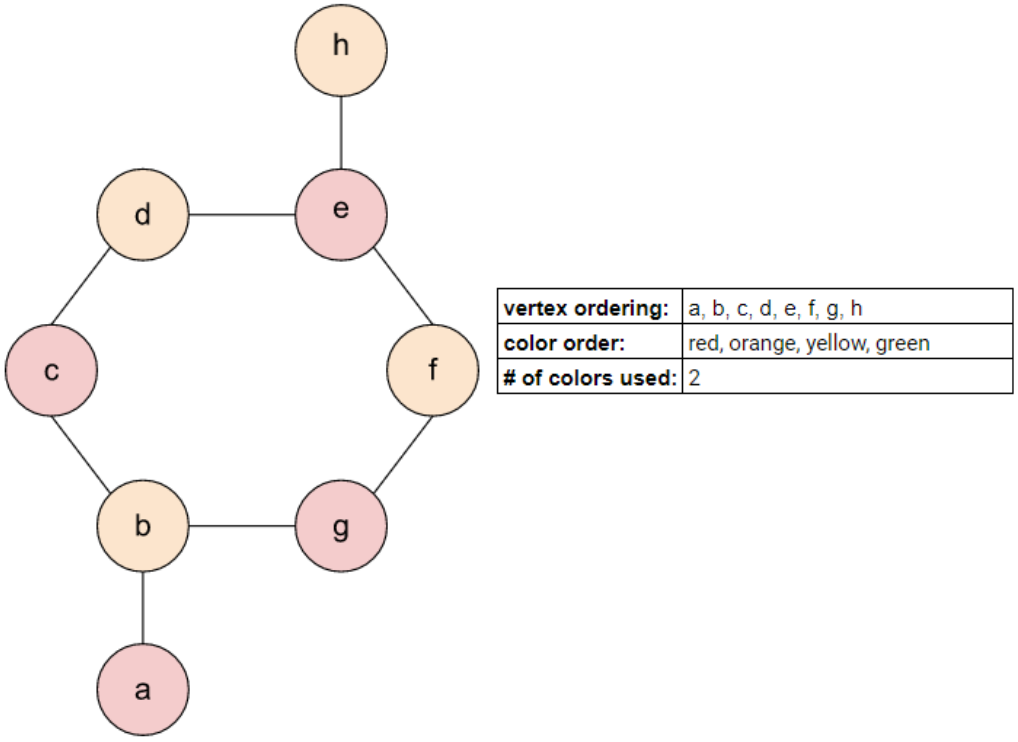

Figure 3.2: Part 1 of 2: An example of node order impacting coloring

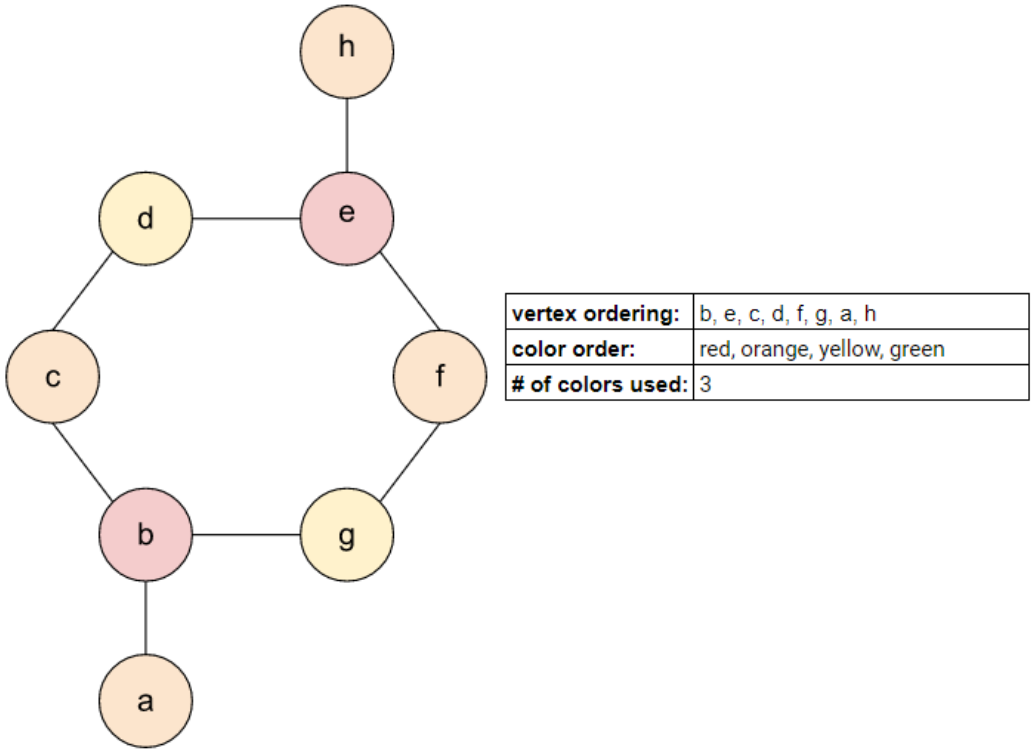

Figure 3.3: Part 2 of 2: An example of node order impacting coloring

There are many algorithms for choosing node order. "A Performance Comparison of Graph Coloring Algorithms," by Murat Aslan and Nurdan Akhan Baykan, tested the prominent algorithms for choosing node order in graph coloring in a way that was super helpful and informative. The algorithms they looked at were: First Fit (FF), Largest Degree Ordering (LDO), Welsh and Powell (WP), Incidence Degree Ordering (IDO), Degree of Saturation (DSATUR) and Recursive Largest First (RLF). The results they found were essentially that all of the algorithms except FF were sufficient for solving the Graph Coloring Problem. For most graphs they all came up with a similar coloring and WP was the fastest, followed by LDO. Then for a couple types of graphs RLF and DSATUR were able to use fewer colors, but they did take longer. The algorithm that I implemented was Largest Degree Ordering (LDO). This method is just putting the nodes in order from largest to smallest degree. This is usually helpful because the nodes with the largest degree being colored last can lead to the worst case coloring with maximum degree + 1 colors. It helps to get the nodes that are most likely to cause conflicts colored and out of the way first. This was simple to implement and, as you will see in the tables, very competitive with the other algorithms. I won't explain the rest of the algorithms but this source does a good job of that so I would recommend reading about them if you are interested. Aslan & Baykan (2016)

In the tables below, which are taken directly from the Aslan and Baykan paper (so don't blame me for the spelling mistake), "V represents the number of the vertices, E is the number of the edges, Den. represents density (D= (2*E)/(V*(V-1)), Best/X(G) means chromatic number or the best known number, R represents the number of colors that algorithms have found, T is computation time in seconds. The algorithms are written in the programming language Matlab R2010a. For experiments we used a Laptop computer. It has Intel Core i5 2.20 GHz processor and 8 GB DDR3 RAM."

They tested these coloring methods on different types of benchmark graphs provided by DIMACS. The tables show results of these algorithms on Mycielski and SGB graphs, Queen graphs, CAR graphs, Random and Flat graphs, Register Allocation graphs, and Leighton graphs.

Graph	V	E	Den.	Best / X(G)	RLF		DSATUR		WP		LDO		IDO		FF	
					R	T	R	T	R	T	R	T	R	T	R	T
myciel3	11	20	0.33	4	4	0.0004	4	0.0023	4	0.0001	4	0.0002	4	0.0009	4	0.0001
myciel4	23	71	0.27	5	5	0.0009	5	0.0077	5	0.0002	5	0.0005	5	0.0028	5	0.0003
myciel5	47	236	0.21	6	6	0.0024	6	0.0255	6	0.0003	6	0.001	7	0.0085	6	0.0006
myciel6	95	755	0.17	7	7	0.0075	7	0.0876	7	0.0004	7	0.0024	7	0.0309	7	0.0016
myciel7	191	2360	0.13	8	8	0.0293	8	0.3254	8	0.0006	8	0.0068	8	0.1366	8	0.0054
miles1000	128	3216	0.39	42	42	0.1065	42	1.2942	43	0.0016	43	0.0123	43	0.7013	44	0.0121
miles1500	128	5198	0.63	73	73	0.3158	73	2.6877	73	0.0024	73	0.0219	73	1.6033	76	0.022
miles500	128	1170	0.14	20	20	0.025	20	0.3249	20	0.001	20	0.0055	20	0.136	22	0.0049
miles750	128	2113	0.26	31	31	0.0522	31	0.7112	32	0.0013	32	0.0083	31	0.3443	34	0.0081
anna	138	493	0.05	11	11	0.0135	11	0.1231	11	0.0006	11	0.0037	11	0.0457	12	0.0026
david	87	406	0.11	11	11	0.0076	11	0.1026	11	0.0005	11	0.0025	11	0.0346	12	0.0017
homer	561	1629	0.01	13	13	0.3404	13	0.5412	13	0.0024	13	0.0232	13	0.246	15	0.0183
huck	74	301	0.11	11	11	0.0062	11	0.0745	11	0.0005	11	0.0021	11	0.0244	11	0.0014
jean	80	254	0.08	10	10	0.006	10	0.0616	10	0.0004	10	0.002	10	0.0198	10	0.0013
games120	120	638	0.09	9	9	0.018	9	0.1537	9	0.0006	9	0.0039	9	0.0577	9	0.0032

Figure 3.4: The results and computation times for Mycielski and SGB graphs.

Graph	V	E	Den.	Best/χ(G)	RLF		DSATUR		WP		LDO		IDO		FF	
					R	T	R	T	R	T	R	T	R	T	R	T
queen5_5	25	160	0,51	5	**5**	**0,0012**	**5**	0,0332	7	0,0003	7	0,0006	7	0,0109	8	0,0005
queen6_6	36	290	0,45	7	**8**	**0,0028**	9	0,0634	9	0,0003	9	0,0010	10	0,0218	11	0,0008
queen7_7	49	476	0,40	7	**9**	**0,0045**	11	0,1090	12	0,0005	12	0,0016	12	0,0461	10	0,0013
queen8_12	96	1368	0,30	12	**13**	**0,0202**	14	0,3851	15	0,0007	15	0,0045	15	0,1779	15	0,0041
queen8_8	64	728	0,36	9	**11**	**0,0081**	12	0,1783	13	0,0005	13	0,0024	15	0,0923	13	0,0020
queen9_9	81	1056	0,32	10	**12**	**0,0154**	13	0,2853	15	0,0007	15	0,0036	15	0,1312	16	0,0030
queen10_10	100	2940	0,59	11	**13**	**0,0215**	14	0,4351	17	0,0009	17	0,0052	17	0,1876	16	0,0044
queen11_11	121	3960	0,54	11	**14**	**0,0345**	15	0,6300	17	0,0009	17	0,0072	18	0,2964	17	0,0063
queen12_12	144	5192	0,50	13	**15**	**0,0550**	16	0,9163	19	0,0010	19	0,0100	20	0,4604	20	0,0092
queen13_13	169	6656	0,47	13	**16**	**0,0800**	17	1,3226	23	0,0013	23	0,0134	22	0,6869	21	0,0125
queen14_14	196	8372	0,44	16	**17**	**0,1227**	19	1,8408	25	0,0015	25	0,0170	24	1,0488	23	0,0169

Figure 3.5: The results and computation times for Queen graphs.

Graph	V	E	Den.	Best/χ(G)	RLF		DSATUR		WP		LDO		IDO		FF	
					R	T	R	T	R	T	R	T	R	T	R	T
1_Fullins_4	93	593	0,14	5	5	0,0069	5	0,0869	5	**0,0003**	5	0,0021	6	0,0298	11	0,0016
1_Fullins_5	282	3247	0,08	6	6	0,0612	6	0,5026	6	**0,0007**	6	0,0104	7	0,2167	14	0,0098
1_Insertions_4	67	232	0,10	5	5	0,0058	5	0,0258	5	**0,0002**	5	0,0013	5	0,0090	5	0,0008
1_Insertions_5	202	1227	0,06	6	6	0,0314	6	0,1527	6	**0,0005**	6	0,0055	6	0,0569	6	0,0038
1_Insertions_6	607	6337	0,03	7	7	0,3575	7	1,2288	7	**0,0023**	7	0,0344	7	0,6011	7	0,0308
2_Fullins_3	52	201	0,15	5	5	0,0027	5	0,0237	5	**0,0002**	5	0,0010	5	0,0076	10	0,0008
2_Fullins_4	212	1621	0,07	6	6	0,0329	6	0,2116	6	**0,0006**	6	0,0060	6	0,0796	14	0,0052
2_Fullins_5	852	12201	0,03	7	7	0,8307	7	3,2494	7	**0,0044**	7	0,0703	7	1,8256	18	0,0763
2_Insertions_4	149	541	0,05	5	5	0,0188	5	0,0621	5	**0,0004**	5	0,0036	5	0,0230	5	0,0021
2_Insertions_5	597	3936	0,02	6	6	0,3721	6	0,6322	6	**0,0023**	6	0,0276	6	0,2985	6	0,0211
3_Fullins_3	80	346	0,11	6	6	0,0060	6	0,0380	6	**0,0003**	6	0,0017	6	0,0134	12	0,0012
3_Fullins_4	405	3524	0,04	7	7	0,1476	7	0,5354	7	**0,0012**	7	0,0157	8	0,2370	17	0,0150
3_Fullins_5	2030	33751	0,02	8	8	10,3646	8	18,3988	8	**0,0254**	8	0,3786	9	11,5821	22	0,4600
3_Insertions_3	56	110	0,07	4	4	0,0033	4	0,0125	4	**0,0002**	4	0,0010	4	0,0047	4	0,0006
3_Insertions_4	281	1046	0,03	5	5	0,0731	5	0,1288	5	**0,0007**	5	0,0073	5	0,0498	5	0,0048
3_Insertions_5	1406	9695	0,01	6	6	3,6966	6	2,3609	6	**0,0128**	6	0,1101	7	1,2766	6	0,0996
4_Fullins_3	114	541	0,08	7	7	0,0107	7	0,0610	7	**0,0004**	7	0,0026	7	0,0216	14	0,0020
4_Fullins_4	690	6650	0,03	8	8	0,5299	8	1,2976	8	**0,0031**	8	0,0386	8	0,6677	20	0,0387
4_Fullins_5	4146	77305	0,01	9	9	89,5661	9	85,9533	9	**0,1131**	9	1,6550	9	55,6602	26	2,0289
4_Insertions_3	79	156	0,05	4	4	0,0065	4	0,0180	4	**0,0002**	4	0,0015	4	0,0068	4	0,0009
4_Insertions_4	475	1795	0,02	5	5	0,2527	5	0,2467	5	**0,0015**	5	0,0188	5	0,1009	5	0,0103
5_Fullins_3	154	792	0,07	8	8	0,0220	8	0,0922	8	**0,0005**	8	0,0036	8	0,0332	16	0,0029
5_Fullins_4	1085	11395	0,02	9	9	1,8848	9	2,9627	9	**0,0080**	9	0,0874	9	1,6530	23	0,0923

Figure 3.6: The results and computation times for CAR graphs.

Graf	V	E	Den.	Eniyi/ χ(G)	RLF		DSATUR		WP		LDO		IDO		FF	
					R	T	R	T	R	T	R	T	R	T	R	T
DSJC125_1	125	736	0,09	5	**6**	**0,0135**	**6**	0,0846	7	0,0005	7	0,0032	7	0,0320	8	0,0024
DSJC125_5	125	3891	0,50	17	**21**	**0,0468**	22	0,6111	23	0,0011	23	0,0079	25	0,2966	26	0,0074
DSJC125_9	125	6961	0,89	44	**49**	**0,1811**	51	1,4215	53	0,0019	53	0,0162	54	0,7575	56	0,0154
DSJC250_1	250	3218	0,10	8	**10**	**0,0665**	**10**	0,4791	11	0,0011	11	0,0142	12	0,2086	13	0,0097
DSJC250_5	250	15668	0,50	28	**35**	**0,4661**	37	4,9399	41	0,0025	41	0,0371	40	3,0145	43	0,0394
DSJR500_1	500	3555	0,03	12	**12**	**0,2863**	13	0,5829	13	0,0023	13	0,0237	13	0,2609	15	0,0199
R250_5	250	14849	0,48	65	71	0,7803	**68**	**4,6108**	70	0,0034	70	0,0439	69	2,7790	79	0,0478
flat300_20	300	21375	0,48	20	**38**	**0,7199**	42	8,5125	44	0,0032	44	0,0554	45	5,3253	47	0,0609
flat300_26	300	21633	0,48	26	**39**	**0,8435**	41	8,5990	45	0,0030	45	0,0566	48	5,5501	45	0,0610
flat300_28	300	21695	0,48	28	**38**	**0,8624**	42	8,6611	45	0,0030	45	0,0564	48	5,5324	46	0,0613

Figure 3.7: The results and computation times for Random and Flat graphs.

Graph	V	E	Den.	Best/χ(G)	RLF		DSATUR		WP		LDO		IDO		FF	
					R	T	R	T	R	T	R	T	R	T	R	T
fpsol2_i1	496	11654	0,09	65	65	0,9869	65	3,1791	65	**0,0044**	65	0,0646	65	1,8096	65	0,0552
fpsol2_i2	451	8691	0,09	30	30	0,5217	30	1,9960	30	**0,0024**	30	0,0442	30	1,1139	30	0,0409
fpsol2_i3	425	8688	0,10	30	30	0,5184	30	1,9752	30	**0,0022**	30	0,0427	30	1,0739	30	0,0407
mulsol_i1	197	3925	0,20	49	49	0,1299	49	0,6347	49	**0,0021**	49	0,0153	49	0,2924	49	0,0137
mulsol_i2	188	3885	0,22	31	31	0,1171	31	0,6423	31	**0,0015**	31	0,0145	31	0,2899	31	0,0133
mulsol_i3	184	3916	0,23	31	31	0,1164	31	0,6189	31	**0,0015**	31	0,0143	31	0,2805	31	0,0134
mulsol_i4	185	3946	0,23	31	31	0,1243	31	0,6328	31	**0,0015**	31	0,0145	31	0,2994	31	0,0130
mulsol_i5	186	3973	0,23	31	31	0,1253	31	0,6286	31	**0,0015**	31	0,0145	31	0,2900	31	0,0128
inithx_i1	864	18707	0,05	54	54	2,7427	54	6,7614	54	**0,0066**	54	0,1337	54	4,2802	54	0,1266
inithx_i2	645	13979	0,07	31	31	1,4014	31	4,2319	31	**0,0037**	31	0,0839	31	2,5214	31	0,0800
inithx_i3	621	13969	0,07	31	31	1,3034	31	4,1724	31	**0,0035**	31	0,0819	31	2,5577	31	0,0780
zeroin_i1	211	4100	0,18	49	49	0,1427	49	0,6636	49	**0,0020**	49	0,0157	49	0,3188	49	0,0139
zeroin_i2	211	3541	0,16	30	30	0,1062	30	0,5390	30	**0,0014**	30	0,0136	30	0,2504	30	0,0124
zeroin_i3	206	3540	0,17	30	30	0,1150	30	0,5439	30	**0,0014**	30	0,0134	30	0,2530	30	0,0123

Figure 3.8: The results and computation times for Register Allocation graphs.

Graph	V	E	Den.	Eniyi/χ(G)	RLF		DSATUR		WP		LDO		IDO		FF	
					R	T	R	T	R	T	R	T	R	T	R	T
le450_15b	450	8169	0,08	15	17	0,3071	**16**	1,7589	18	0,0025	18	0,0348	18	0,9585	22	0,0337
le450_25a	450	8260	0,08	25	**25**	**0,3502**	**25**	1,7952	26	0,0029	26	0,0367	25	1,0172	28	0,0355
le450_25b	450	8263	0,08	25	25	0,3583	25	1,9924	**25**	**0,0028**	25	0,0371	25	1,0341	27	0,0355
le450_25c	450	17343	0,17	25	**28**	**0,7839**	29	5,9978	29	0,0034	29	0,0626	31	3,6658	37	0,0674
le450_5c	450	9803	0,10	5	**5**	**0,2226**	10	2,4336	12	0,0020	12	0,0352	12	1,3233	17	0,0375
le450_5d	450	9757	0,10	5	**6**	**0,2315**	12	2,4073	14	0,0025	14	0,0362	13	1,2504	18	0,0382

Figure 3.9: The results and computation times for Leighton graphs.

3.2.3 Runtime of Sequenital Coloring with LDO

The coloring algorithm pseudocode looks like this:

```
def check():
      for (n meetings):
            if adjacent to a node of this color return false
      return true


def main():
      for (n meetings):
            for ( n colors):
                  if(check()):
                        assign this meeting to this color
                        exit the color loop and move to the next meeting
```

Figure 3.10: Sequential Coloring pseudocode.

For the sake of readability I did not include the ordering of the nodes in the pseudocode, but that is a part of the actual code.

At first glance we can see that this code has a for loop of size n (the check function), inside a for loop of size n (the colors for loop), inside a for loop of size n (the meetings for loop). This gives us a $O(n^3)$.

In trying to be more precise in estimation I considered the fact that it is not possible to have to go through n colors to find a match each time. Even if every node requires a new color, the first node will only have to check one color, then the next will check two, and so on until the last node has to check all n colors. So instead of $n * n * n$, it is really more like $(1 + 2 + ... + n) * n$. This is equal to $((n^2 - n)/2) * n$ which still simplifies to $O(n^3)$.

3.3 Assigning Colors to time slots

Next, we need to assign colors to time slots. Orals time slots are 90 minutes. There are 10 minute breaks between meetings, and an hour break for lunch. Meetings go from 9:00am to 6:00 pm. This means there are 5 time slots per day. In the fall semester, there are only two days of orals: Thursday and Friday. In the spring semester, there are five days of orals: Monday through Friday. This means that there are 10 time slots in the fall and 25 time slots in the spring.

The number of colors in our graph coloring should not be greater than the number of time slots. This is very unlikely to happen on real world numbers but is theoretically possible. If this becomes a problem we will want to switch to backtracking instead of sequential coloring because if a k coloring exists, backtracking is guaranteed to find it. If it still cannot get the number of colors to be less than or equal to the number of time slots, the problem is not possible to solve.

If the number of colors in our graph coloring is less than but close to the number of time slots, we can just directly assign each color to a time slot. If the number of colors in our graph coloring is significantly less than the number of time slots, we can divide some or all of the color classes in half and assign each half to a time slot. Spreading the meetings out like this could be preferable if we have a shortage or meeting rooms. I have been told by the CS faculty that a shortage of meeting rooms is not a problem that the department ever has, so I did not really consider that as a factor.

Chapter 4

Results

In this chapter, I hope to help the reader understand the code that I have written. The code is all put together in one document in "big_doc.cpp", but it is also broken up into smaller files so one can more easily test it and see how it works. I will break this down into three sections: the matching, building the conflict graph, and coloring the conflict graph.

The matching portion of this code is contained in the document "matching_algorithm.cpp" with help from the document "util_functions.cpp". The input for matching is the number of students, number of professors, number of students each professor is willing to advise, and the student and professor rankings of one another. If the number of students is n and the number of professors is m, the teachers' rankings of students is represented by an m x n matrix. This is basically m lists of n students. Each of the m lists is a different professor's list of preferences. The numbers in the list represent the student number and their position in the list represents the professor's ranking of the student. The students' rankings of professors are in an n x m matrix. Just like in the other matrix, there are n lists of m professors and each list is a different student's rankings of the professors. The numbers represent which professor is being referred to and the order of the list represents the student's ranking of that professor. These 5 variables are the only things that need to be changed in order to change the input. These variables are all grouped together so that you can copy paste the test case inputs into this spot to try out different numbers of professors and students. The rankings in the test files are randomly generated with the file "code_to_generate_test_data.cpp". This section of the code results in student-professor matches in the form of a vector of vectors. This is a series of lists of the students each professor is paired with.

The next step is building the conflict graph. This starts with the code from "add_third_prof.cpp". The starting point for this code is the vector of vectors that represents the matches made in the first section. This code adds a second reader to each of these matches at random and then puts them in an n x 3 matrix. Then,

"matches_to_adj_matrix.cpp" takes over from there. This code iterates through the matches checking if any meetings have the same professors. It uses this information to build an adjacency matrix that represents our conflict graph. This code also keeps track of the degree of each node with an array as it builds the adjacency matrix.

Next, we need to color the graph. This starts with using the list of node degrees that we made in the last section to find the order that we want to color the nodes in. This happens in "ordering.cpp". Then, "new_coloring.cpp" traverse the nodes in that order and sequentially color them.

As we were expecting from this code, there are no guarantees that the coloring will be the lowest possible or that the runtime will be optimal, but it does seem to work well in practice with numbers that would make sense for this problem. When testing this code on six professors and thirty students, roughly the size of the CS department, the code ended up generating colorings between 10 and 15 colors. To get an idea of the practical runtime for the code I have written I did some timed tests as well. The code that I used for testing the time is the "big_doc_time_testing.cpp" file in the "time_testing" folder. This file is just like the "big_doc.cpp" file where I combine all of the code except that I have gotten rid of the print statement because they are not relevant to actual use of this code and are quite slow. This folder also contains the data that I used for the various sized tests.

My tests are run on an MSI GS65 Stealth 8SE. Which is a thin gaming laptop with 8 cores. Each result is an average from three timed runs. On a test size of 4 professors and 6 students the runtime was 5.13e-05 seconds. On a size of 6 professors and 30 students, about the workload we would expect if we were using it to schedule orals for the Computer Science department, the runtime was 0.0001042 seconds. Then on a workload of 161 teachers and 1449 students, a workload roughly based on the number of professors and students at Reed college, the runtime was 5.23647 seconds. Then in an attempt to find at what point the code starts taking an unreasonable amount of time to run, I tried a workload of 10,000 students and 1,000 professors. At this point my editor (atom) started crashing because it couldn't handle matrices of that size. I switched to a notepad++ editor and was able to run it with a runtime of 1719.5 seconds. This is about 28 minutes. This is a long time, but not actually that ridiculous for a program that is doing a scheduling problem of this scale.

Conclusion

I think this thesis is useful in that it can actually be put into use to improve the Reed computer science department. Using the NRMP algorithm for matching up students and advisors would make the process more fair and transparent, and using the graph coloring algorithm to schedule orals would make that process more quick and efficient. That being said, there is a lot that could be done to extend this thesis.

There are some simple changes that would need to be made to finish the implementation so that this code can be used by the Computer Science department. That would look like making the digital forms that students and professors would use to rank one another. Then, one could read the data from the resulting csv file (spreadsheet) into the data structure that it needed for the beginning of my code. Also, my code currently goes from the matching step to adding a random second reader for the sake of starting the graph coloring problem. However, in a final implementation we would either need an algorithm to match the student to a second reader or would need to manually input each student's second reader before running the graph coloring algorithm. It would also be relatively easy to extend this to include the third and fourth reader if we wanted to. Currently, in the CS department, students are just matched with an advisor and a second reader. The third and fourth readers are found by the student. However, it might save us all some emailing back and forth if we were to automate that process. Finally, one could very easily write the code that maps the colors to time slots as described in that section of my thesis.

In changing the scale of this project one might want to change the algorithm used for ordering the vertices in the sequential coloring algorithm. If speed is a priority over the number of colors used I would suggest using the Welsh Powell algorithm, and if minimizing the number of colors used is important and time is less of a priority I would suggest using the Recursive Largest First algorithm. Alternatively, if one needs to find out if coloring with a specific number of colors is possible, I would recommend using backtracking instead of sequential coloring.

Further extensions to this project could include taking into account time slots that people have personal scheduling conflicts with. This would look like the form that takes student and professor rankings having a question where you can indicate

certain time slots that don't work for you. Then this information could be taken into consideration when assigning colors to time slots. One could also include taking indifference into consideration in the ranking process and matching algorithm. The stable marriage problem with indifference has been studied and a version of that could pretty easily be used to add indifference as a factor in our matching. Also, rather than having a cap for the number of slots that each professor has available, as in my current code, one could treat the number of students a teacher would like to advise as another preference in this constraint satisfaction problem. It could get more and more expensive to add matches to one professor but still possible if it is a good enough match. I do not know exactly how one would solve this, but I imagine it could be done with a variation of maximum weight matching. Also, in the current implementation of my project, I imagine the number of students and the number of available slots will be somewhat close so I didn't look at balancing the workload when there are many more slots available than there are students, but one could consider that factor in a future extension. One potential way to do this would be artificially limiting the number of positions that professors have available.

More complicated extensions of this project could look like dealing with the matching and scheduling involved in scheduling classes for the school. This would look similar to this project, but would have to include many more factors such as available classrooms and their capacities, professor schedules, student schedules, required courses vs. optional classes, etc. This would be a complicated problem but also very interesting and useful.

Further work that is likely beyond the scope of a thesis but would be very cool might include thoroughly analyzing the conflict graphs that arise from this kind of problem to see if they have any properties that might make a more efficient coloring possible.

While this thesis isn't saying anything about algorithms that hasn't been said before in other papers, I hope the code can be a contribution to the Computer Science department. If they choose to use it, I think it could improve their advisor student matchings and help them schedule orals more easily.

Bibliography

(2012). Stable matching: Theory, evidence, and practical design.

Aslan, M., & Baykan, N. A. (2016). A performance comparison of graph coloring algorithms. *International Journal of Intelligent Systems and Applications in Engineering*, *4*(Special Issue-1), 1–7.

Briggs, P. (1992). *Register Allocation via Graph Coloring*. Rice University.

Chen, X., Ding, G., Hu, X., & Zang, W. (2012). The maximum-weight stable matching problem: duality and efficiency. *SIAM Journal on Discrete Mathematics*, *26*(3), 1346–1360.

College, W. (n.d.). Graph coloring. `https://www.whitman.edu/mathematics/cgt_online/book/section05.08.html`. Accessed on 2022-11-29.

Deo, N. (1974). *Graph Theory with Applications to Engineering Computer Science*. Dover Publications, Inc.

E Peranson, R. R. (1995). The nrmp matching algorithm revisited: theory versus practice. Volume 70, Issue 6, p477-484.

Huang, C.-C. (2006). Cheating by men in the gale-shapley stable matching algorithm. In Y. Azar, & T. Erlebach (Eds.), *Algorithms – ESA 2006*, (pp. 418–431). Berlin, Heidelberg: Springer Berlin Heidelberg.

Sipser, M. (2006). *Introduction to the Theory of Computation*. Thomson Course Technology.

Sysło, M. M. (1989). Sequential coloring versus welsh-powell bound. *Discrete mathematics*, *74*(1-2), 241–243.

Thomas H. Cormen, R. L. R. C. S., Charles E. Leiserson (2009). *Introduction to Algorithmss*. The MIT Press.

University, C. (n.d.a). Examining the national resident matching program. https://blogs.cornell.edu/info4220/2015/03/06/examining-the-national-resident-matching-program/. Accessed on 2022-11-29.

University, C. (n.d.b). Register allocation via graph coloring. https://www.cs.clemson.edu/course/cpsc827/material/Optimization/Register%20Allocation%20via%20Graph%20Coloring.pdf. Accessed on 2022-11-29.