Math 1060 Class Notes

"Ordinary writing has a certain redundancy to it. It needs redundancy, because English (lovely language though it is) can never capture a complex idea with perfect precision. In any phrasing, some shade of meaning is lost or obscured. A subtle, complicated thought must be illuminated from many angles before the reader is able to sift reflections from reality, or tell the shadows from the thing casting them. Thus, typical prose is full of pleasing repetition—paraphrase, caveats. You can skim, and even if you miss a few details, you'll walk away with the gist.

Math is different. Unlike English, mathematical language is built to capture ideas perfectly. Thus, key information will be stated once and only once. Later sentences will presuppose a perfect comprehension of earlier ones, so reading math demands your full attention. If your understanding is holistic, rough, or partial, then it may not feel like any understanding at all."

–Ben Orlin[1]

Introduction

These notes are for students in my MATH 1060 class. They are not a substitute for attending class, as they may include material not covered in class, and class may cover material not included in the notes. Ultimately, the material that will be tested over is the material presented in class. These notes are merely to provide extra help to those who may have a hard time following the in-class material at speed.

These notes are also organized slightly differently than the material in the textbook, and sometimes material may show up in different places than it does in the textbook. Of course, there is also material that I have intentionally left out and material that I have decided to include in addition to the material from the textbook.

[1] http://mathwithbaddrawings.com/2015/03/17/the-math-major-who-never-reads-math/#more-3263

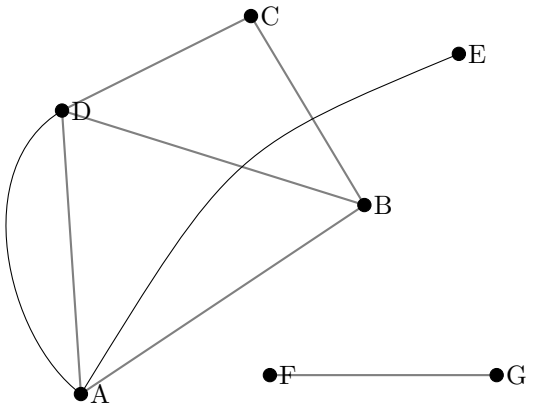
1 Chapter 1

1.1 Graph Theory

Graph theory is the mathematical theory that deals with *graphs*. These are not the same graphs that you are probably familiar with, which display the values of a function in the Cartesian plane. Instead, the meaning of the word graph in this context is quite different.

Definition: A graph is a finite set of points called *vertices* and a finite set of connections between those points called *edges*. In this class, we'll also assume that edges have to be drawn between distinct vertices, i.e. we can't have an edge connecting a vertex to itself (this type of edge is called a *loop*).

Example: The graph H shown below has seven vertices and eight edges. Note that the vertices have been given the names A, B, C, ..., G, but the edges have not been given names. Oftentimes, we can simply refer to the edges by the vertices they connect, e.g. the edge connecting C and B can be referred to as the edge BC. However, in this particular example there are two edges connecting A and D, so we can't simply refer to an edge as edge AD because we don't know which of the two that name refers to. In this case, we could label them AD_1 and AD_2.



There are a few more things to note about this example. Note that not every place where two lines representing edges cross is a vertex.

Definition: A path is a connected sequence of edges that starts at a vertex and ends at a vertex.

Definition: A circuit is a path that starts and ends at the same vertex.

Definition: The *valence* of a vertex is the number of edges that connect to that vertex.

Definition: A graph is *connected* is for every pair of vertices X_1 and X_2 in the graph, there exists a path starting at X_1 and ending at X_2.

1.2 Modeling Using Graphs

The focus of this class is on using mathematical concepts to model real-world scenarios. Graphs have a large number of modeling applications, such as street networks, airplane networks, bordering countries, among others.

1.3 Euler Circuits

Definition: An *Euler path* is a path that covers every edge in a graph exactly once.

Definition: An *Euler circuit* is a circuit that covers every edge in a graph exactly once. Thus, it is an Euler path that starts and ends at the same vertex.

2 Chapter 2

2.1 The Traveling Salesman Problem

The *Traveling Salesman Problem* is the problem of finding a minimum-cost Hamiltonian circuit on a weighted graph.

The Brute Force Algorithm: 1. Generate all possible Hamiltonian circuits. 2. Calculate the cost of each Hamiltonian circuit. 3. Pick the Hamiltonian circuit with the smallest cost.

The brute force algorithm is very slow when the number of vertices on a complete graph is large, but it also always gives us the correct solution to the problem.

The Nearest-Neighbor Algorithm: Given a starting vertex and a weighted graph: 1. From the current vertex, move along the least expensive edge that connects to a vertex that has not already been visited.

The nearest-neighbor algorithm is very fast compared to the brute force algorithm, but it is what is known as a heuristic algorithm, because it doesn't always solve the original problem. We hope that the Hamiltonian circuit generated by the nearest-neighbor algorithm is "good enough" even if it is not the efficient Hamiltonian circuit.

The Sorted-Edges Algorithm: Given a complete weighted graph

2.2 The Minimum-Cost Spanning Tree Problem

Kruskal's Algorithm:

3 Chapter 3

3.1 The Scheduling Problem

There are a wide variety of scheduling-type problems, but in this section we will focus on one in particular and variations on it. Vaguely speaking, the problem is this: given a set of tasks, which we'll refer to as a *job*, how can we "best" schedule the completion of these tasks? We can the things that complete the jobs *processors*. The processors can be people or machines. Before we state the problem precisely, we need to define some new concepts.

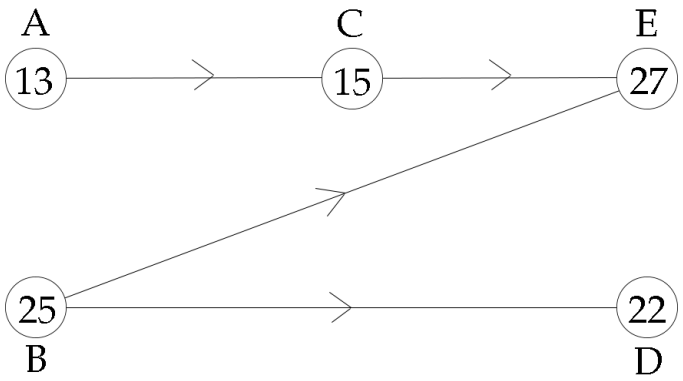
Definition: A *directed graph* is a graph where each edge can also be given a direction, indicated by an arrow on the edge.

Definition: An *order-requirement directed graph* (ORDG), also known as an order-requirement digraph, is a directed graph where the vertices represent tasks, and if, say, there is an arrow pointing from vertex A to vertex B, then that indicates that task A must be completed before task B can be started.

Example: An airplane's turnaround time depends on the completion of several tasks:

Task	Description of Task	Time to Complete
A	Unload Passengers	13 min
B	Unload Cargo	25 min
C	Clean Cabin	15 min
D	Load New Cargo	22 min
E	Load New Passengers	27 min

The order-requirement directed graph for this job is below:



This shows us that task C, cleaning the cabin, can only be started once task A, unloading the passengers, has been completed. Task E can only be started once tasks C and B have been completed. And task D can only be started once task B has been completed. The numbers on each vertex are the times to complete each task.

Definition: We say that a task is *ready* at a certain time t if all of its prerequisites given in the order-requirement directed graph are completed at time t and if that task is not already being worked on by a processor.

Now we can state the scheduling problem that we will be working with.

The Scheduling Problem: Given some number of tasks given in an order-requirement directed graph (ORDG), given some number of identical processors, and given a priority list of the tasks, how should we schedule the processors to work on the tasks?

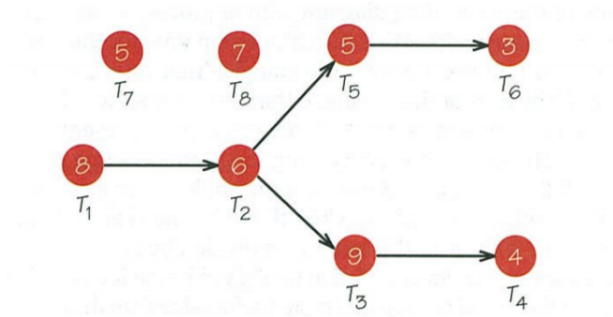
We make three assumptions for this problem:
1. The processors are all identical, so they can all do every task and each task will be completed in the same time across all processors.
2. Once a processor starts a task, it will work on it until the task is completed.
3. No processor can stay voluntarily idle, i.e. if there is at least one ready task, then the processor will start work on one of those ready tasks.

When we are given an ORDG, a number of processors, and a priority list for the tasks, then there is a simple algorithm to generate a schedule for completing the job.

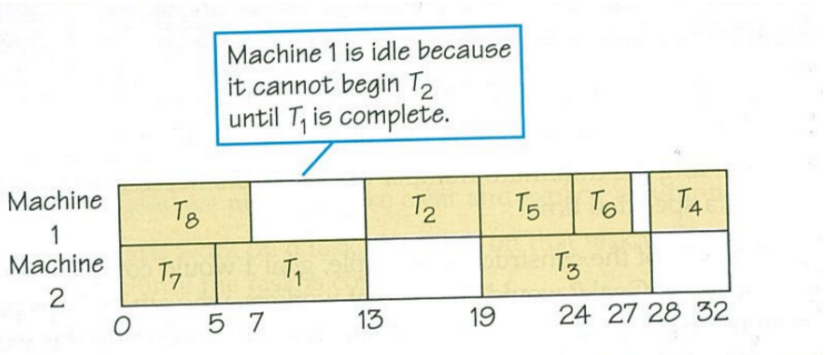
List-Processing Algorithm: At a given time, assign to the lowest-numbered processor the highest-priority task on the priority list that is ready at that time.

Example: Given two processors (call them P_1 and P_2), the order-requirement directed graph given below, and the priority list T_8, T_7, T_6, $\ldots$, T_1, use the list-processing algorithm to generate a schedule for the

job.



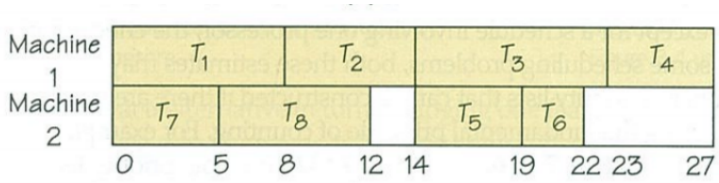
At time $t = 0$, the ready tasks are T_1, T_7, and T_8, so the highest priority ready task is T_8. So we assign task T_8 to processor P_1, because P_1 is the lowest-numbered available processor. Also at time $t = 0$, we assign the next highest priority task T_7 to processor P_2. When a task is finished, we have to check our ready tasks and the priority list to assign another job, so as soon as P_2 finishes task T_7 we see that the only ready task is T_1, so we assign that task to P_2. When P_1 finishes its task, however, there are no ready tasks, so P_1 must remain idle until P_2 completes task T_1. The finished schedule is given below.



When we are given the priority list, the list-processing algorithm will simply generate a schedule, but oftentimes we are more interested in what happens if we are allowed to change the priority list. Different priority lists can generate different schedules, some of which take longer than others to finish the job. Consider the following example.

Example: Given two processors P_1 and P_2, the order-requirement directed graph from the last example, and the priority list $T_1, T_2, T_3, \ldots, T_8$, use the list-processing algorithm to generate a schedule for the job.

The generated schedule is given below.



Notice that this schedule finishes the job in only 27 units of time, instead of 32 units of time like in the first example. How can we know if this schedule takes the least amount of time of all of the possible schedules that can be generated for this problem with different priority lists?

To answer this question, we need to know when we have a schedule that takes the minimum amount of time of all possible ones. So, given a problem, how can we tell if there cannot be a schedule that takes less time

than one we already have? There are two possible ways to know:

1. Length of *critical path*: A *critical path* in an order-requirement directed graph is the length of the longest path in the graph, where the length is determined by adding up the times shown on the vertices that lie on the path. Because no task can be started before its prerequisites are complete, we know that a job cannot be completed in less time than the length of the critical path.

2. Total time of tasks divided by number of processors: if we were not constrained by the structure of the tasks given in the order-requirement directed graph or by the fact that only one processor can work on a task at a time, we could always have every processor working towards the completion of the entire job. In this case, the time to complete the job would be the total time of all of the tasks divided by the number of processors. With the constraints we have, we can only do worse than this, so it will take at least as long as the total time of all of the tasks divided by the number of processors to complete the job.

So, for the examples that we have done, we *know* that the second example is in fact the optimal schedule (finishes the job in the minimum amount of time) because the length of the critical path in the order-requirement directed graph is exactly 27 units of time.

3.2 The Coloring Problem

The Coloring Problem: Given a map of some region with some number of connected countries, how can we pick a color for each country so that:
1. No country is the same color as another country that borders it, and
2. We use as few colors as possible?

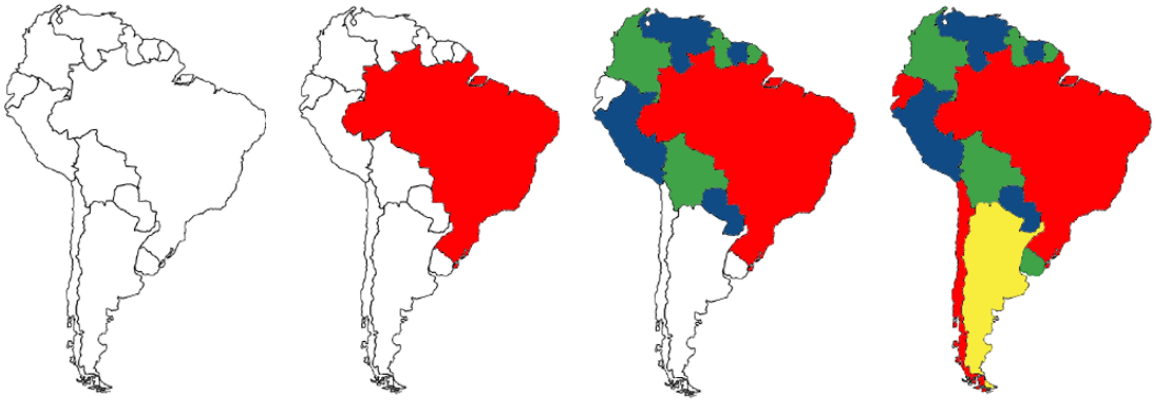
Example: Given the below map of South America, how can we go about trying to color it with as few colors as possible?

First, let's pick a color for Brazil, say red. Because there are countries that border Brazil, we can't color the whole map with one color. Then because any country that borders Brazil also borders another country that borders Brazil, we see that we can't color the map with two colors.

What about three colors? We can try to go around the border of Brazil and alternate two other colors (say green and blue) for each country that borders Brazil. This works well until we get to Argentina. Because

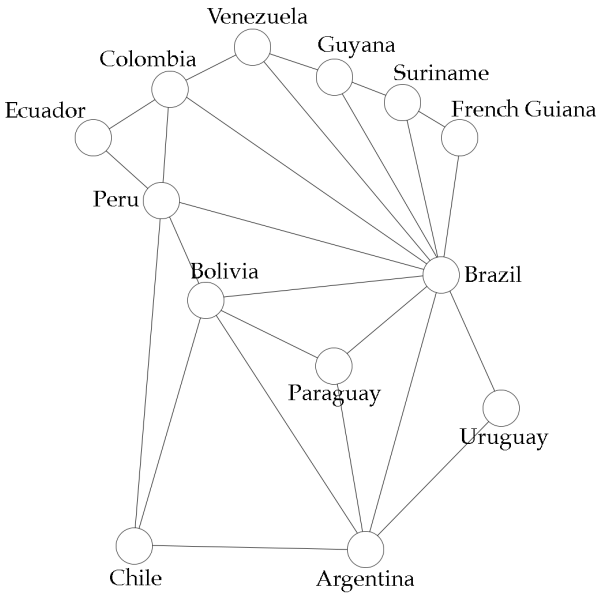
Argentina borders a red country, a green country, and a blue country, we can't color it with any of the three colors we're using. Therefore we can't color this map with three colors.

However, if we color Argentina another color (yellow), then we can finish coloring in the map. Thus, we can color the map of South America with four colors.



To better understand the problem, let's model it using graphs. We represent every country as a vertex, and draw edges between those countries that share a border.

Now if we look at the graph, notice that Paraguay and its neighbors form a complete graph with four vertices, and thus we need at least four colors to color that section. If Paraguay were removed from the graph, we could color the map with only three colors.



Can every map be colored with four colors? If you look at various other maps, you might notice that seemingly every map can be colored using only four colors, but how do we know that there isn't some map that could be drawn that would take more than four?

For example, why couldn't we take a complete graph with five vertices and make a map that is represented by that graph? It turns out there there is no map *on a flat surface* that can be represented by a complete graph with five vertices.

In fact, there no maps on a plane that require more than four colors, but the reasons why this is true are so complicated that they can only be "understood" by computers!

Theorem: (The Four Color Theorem) Every map on a plane can be colored with no more than four colors.

Graphs that can represent maps on a plane are called *planar graphs*, and can be distinguished by being able to be drawn without any edges crossing. Be careful though! There are ways to draw planar graphs that have edges crossing. Try to "untangle" a graph before deciding it is not planar. So we can state the theorem as:

Theorem: (The Four Color Theorem) Every planar graph can be colored with no more than four colors.

The Coloring Problem: (General Statement) Given a graph, how can each vertex be colored so that no two vertices connected by an edge are the same color?

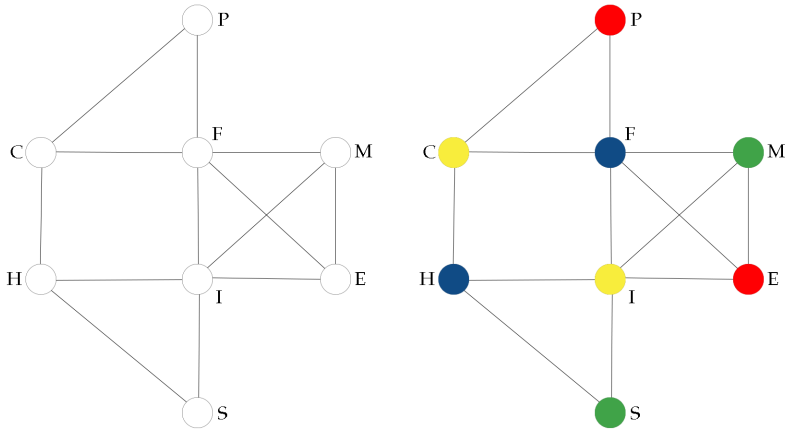
The coloring problem has more general applications than just cartography. We'll look at an application to another type of scheduling problem.

Suppose a college is scheduling final exams for its eight classes and wants to schedule them so that no student has two or more exams scheduled at the same time. The table below gives the different classes that have at least one student in common an "X."

	F	M	H	P	E	I	S	C
French (F)		X		X	X	X		X
Math (M)	X				X	X		
History (H)						X	X	X
Philosophy (P)	X							X
English (E)	X	X				X		
Italian (I)	X	X	X		X		X	
Spanish (S)			X			X		
Chemistry (C)	X		X	X				

If we model this problem as a graph with the classes as vertices and edges given between classes with at least one student in common, then the problem becomes "how can we classify each vertex so that no two vertices that share an edge are given the same classification?" Notice that this is the exact same as the coloring problem, with a "classification" being a color.

To add another layer to the problem, we can try to minimize the number of rooms needed for the exams. Because a color represents a certain exam time, if we have three vertices with a particular color, that time slot needs three rooms. The below solution to the question shows that we can solve this problem with only two rooms and four time slots.


P
C
F
M
H
I
E
S
P
C
F
M
H
I
E
S