

# Math 118 E1

## Final Exam Review

This semester we covered three major topics:

- Using algorithms to solve problems (Chapters 1, 2, and 3)
- Encoding information mathematically (Chapter 9)
- Logically examining different voting methods (Chapters 12 and 13)

The final exam will be cumulative, so it is important that you understand the material from each chapter. The purpose of this review guide is to remind you of the concepts and to give you the big picture; studying the details is up to you. Please find someone to help you if you do not understand something. I will be available throughout finals week if you need to ask me questions.

## 1 Chapter 1 – Euler circuits

Chapter 1 discusses finding efficient routes that cover all possible paths between different points. For example, a postal worker might want to deliver mail to all of the houses in a neighborhood without having to do any backtracking. First we want to know whether such a route exists, and then we would like to know if there is a good way to find one.

To answer these questions, we used a mathematical abstraction called a *graph*. A graph is simply a set of points (*vertices*) and lines that connect them (*edges*). For these problems, our edges represent the streets or sidewalks we would like to cover, and the vertices represent “intersections” where two edges meet. A solution to our problem is known as an *Euler circuit*.

By counting how many edges enter and leave a vertex, we can determine the *valence* of each vertex. Euler’s Theorem tells us when a (connected) graph has an Euler circuit, based on the valences of all the vertices. Then finding such a circuit simply involves making sure we don’t cut off part of the graph as we pick a route.

If an Euler circuit does not exist, we can still look for an optimal solution. This requires *eulerizing* the graph, or adding edges to make all the valences even. Then we may proceed as before, recognizing that added edges represent traveling back across an existing edge.

## 2 Chapter 2 – Hamiltonian Circuits

In Chapter 1, we focused on the edges of our graph. We wanted a path that begins and ends at the same point and covers each edge only once. In Chapter 2, we shift our focus to the vertices. Now we want a path that visits each vertex only once and returns us back to our starting place. These circuits are known as *Hamiltonian circuits*. In addition, by adding *weights* to the edges, representing the cost of traveling along that edge, we can look for the most efficient such circuit.

Generally speaking, it is a very hard problem to find a minimum-cost Hamiltonian circuit. We don’t even know if a Hamiltonian circuit exists for arbitrary graphs, but when each vertex is connected to every other vertex, we know at least one such circuit exists. These graphs are called *complete graphs*. In fact, for complete graphs with  $n$  vertices, we know that there are  $\frac{(n-1)!}{2}$  Hamiltonian circuits.

Unfortunately, even for complete graphs, finding the minimum-cost Hamiltonian circuit is a hard problem. The *method of trees* guarantees us a solution, but tends to take a long time. The number  $\frac{(n-1)!}{2}$  grows very quickly as  $n$  grows, so even for 25 vertices, modern computers cannot solve the problem. And even if they could, they probably could not find such a circuit for 50 vertices.

Instead of abandoning all hope, however, we use *heuristic algorithms* designed to find “pretty good” solutions most of the time. For finding minimum-cost Hamiltonian circuits, we have two such algorithms: *nearest-neighbor* and *sorted edges*. Each is relatively easy to do by hand, and tends to find pretty good solutions in most cases.

Another application of graphs is finding a way to ensure that all vertices are connected by some path. To cut down on costs, we will typically look for *trees*, graphs without circuits. A *spanning tree* of a graph is a tree within the graph that contains every vertex. Our goal is to find minimum-cost spanning trees.

Here we find that the heuristic algorithm that we use does indeed always find the optimal solution. So given a graph connecting several vertices, we can use *Kruskal's algorithm* to guarantee a minimum-cost spanning tree. While this algorithm is similar to the sorted edges algorithm, it is inherently different because the end result (a spanning tree) is different from that of the sorted edges algorithm (a Hamiltonian circuit).

Our last application of graphs involves order requirements diagrams: given a set of tasks to be completed, which ones cannot be started until some others are finished? We represent this ordering by a graph, where the vertices are the tasks, and our edges now have arrows indicating that a task needs to be completed before the next one begins. Finding the critical path tells us the earliest possible completion time for all of the tasks.

In summary, we have several problems that can be solved using algorithms. Some algorithms are inefficient, while others might not always guarantee the best solution. Knowing what input and output an algorithm expects can help you make sure you are using the correct one to solve your problem.

- **Method of trees**

- A *brute force* algorithm
- Input: a complete graph
- Output: the minimum-cost Hamiltonian circuit

- **Nearest-neighbor**

- A heuristic algorithm
- Input: a complete graph
- Output: a Hamiltonian circuit (not necessarily minimum-cost)

- **Sorted edges**

- A heuristic algorithm
- Input: a complete graph
- Output: a Hamiltonian circuit (not necessarily minimum-cost)

- **Kruskal's algorithm**

- A heuristic algorithm
- Input: a connected graph
- Output: the minimum-cost spanning tree

### 3 Chapter 3 – Scheduling

In Chapter 3, we continue with our list of tasks to be completed. Using our order requirements graph, how can we assign the jobs to different workers to get the work done quickly? This can be quite difficult in general, so we start with a few simplifying assumptions.

First, we assume that all of our workers are equal – everyone is capable of doing every task, and everyone works equally hard. Second, we assume that no one is lazy. If there is a task that can be done, and someone is available to do it, they will start working on it. Third, our tasks need to be given by an order requirements graph. This just ensures that we know in advance what jobs depend on the others. Finally, we want a priority list that is independent from the order requirements.

When there is more than one job that can be started, this list will give us a way to decide which task to start next.

Given these assumptions, we may now use the *list processing algorithm* to determine which workers will do which jobs. Here we choose the next ready task and assign it to the lowest-numbered available processor. This continues until all the tasks have been completed. If this is done in the same amount of time as the earliest completion time, we say that the schedule is *optimal*.

There are two scenarios which differ slightly from this. In one case, we realize that our priority list may have been determined somewhat artificially, and that another priority list could actually result in a more efficient schedule. One way to determine such a priority list is *critical path scheduling*. To create this list, we add tasks one at a time, taking the head of the critical path at each stage.

In the other case, we have a set of independent tasks – there are no arrows on our order requirements graph. Now our critical path is not a good measure of efficiency, so we add up the total time for all the tasks and divide by the number of processors to find out the earliest possible completion time.

Instead of starting with a fixed number of processors and trying to find the quickest schedule, we can instead set a “deadline” and try to find the fewest number of workers required to finish the job by this deadline. Another way of thinking about this problem is *bin packing*: we have several items of certain sizes, and we have bins of fixed capacity. How can we put the items in the bins, using the fewest number of bins?

Depending on whether we have all of the items available to us when we start, we can use a number of algorithms to solve the bin packing problem. *Next Fit* (NF), *First Fit* (FF), and *Worst Fit* (WF) all specify how to find the bin for each item. If we sort the items in order from largest to smallest before we add them to the bins, we will often get better results. These variations are referred to as NFD, FFD, and WFD. To determine the optimal solution, we can add all the item sizes and divide by the bin capacity.

Note that in any of these problems, it might not actually be possible to do as well as the optimal solution. This optimum just lets us know that we can stop trying to better once we reach it.

Our last scheduling problem involved trying to avoid conflicts. This *conflict scheduling* can be used when trying to arrange final exams or determining emergency radio frequencies for nearby towns. The first step is to find a way to represent our scheduling problem. We use a chart to show all the possible conflicts. This does not lend itself well to finding a solution, however, so we convert the chart into a graph, where the edges represent conflicts. Now we simply need to determine colors for the vertices, with no two adjacent vertices being the same color.

## 4 Chapter 9 – Encoding information

Chapter 9 taught us how to put interesting information into identification numbers. We can add a digit to check for possible typographical errors, we can use numbers to specify manufacturers and publishers, and we can capture the basic “sound” of a last name for driver’s licenses.

Check digits are computed using the digits in an identification number. Each digit is multiplied by some number, called a *weight*, and then these are all added together. Then we just need to find out what to add to get to a certain multiple, and this is our check digit. Some common uses of check digits are the following:

- Money orders – no weight; check digit is remainder when dividing by 9
- Traveler’s checks – no weight; check digit is added to get a multiple of 9
- Bank identification numbers – weight (7, 3, 9); check digit is last digit of the sum using this weight
- Codabar – weight (2, 1); add in the number of odd-position digits greater than or equal to 5; check digit is added to get a multiple of 10
- UPC – weight (3, 1); check digit is added to get a multiple of 10

- ISBN – weight (10, 9, 8, . . . , 1); check digit is added to get a multiple of 11 (use  $X = 10$ )

Identification numbers sometimes encode more than just check digits. U.S. Postal zip codes contain information telling where the address is. Social security numbers identify the state in which they are issued. Perhaps the most common use of encoding information is the UPC and ISBN schemes. In a UPC, the first digit represents the type of the product, the next set of five digits represents the manufacturer, and the next five digits are assigned by the manufacturer to identify the product. A similar scheme is used for ISBN, taking into account the language where the publisher is located.

The Illinois state driver’s license numbers contain a lot of information about the driver. The first four digits are computed using the *Soundex algorithm*. This algorithm attempts to extract information about how the name sounds (based on the English language). By using this method, computers can search for slight variations in spellings. The next three digits encode the first and middle initials. The last five digits contain the holder’s birth date and gender.

## 5 Chapter 12

Next we turn to analyzing voting systems. Chapter 12 discusses situations in which every voter and every candidate is considered equal. For two voters, this yields the *majority rule* system, being completely fair in some sense. However, when there are three or more candidates, majority rule can no longer be relied upon.

While there are several different voting systems, there are some conditions we would like to have for determining the outcome of our elections. These fairness conditions go by the terms *CWC*, *IIA*, *Pareto*, and *Monotonicity*. In order to more accurately analyze an election, we need to know all of a voter’s preferences, not just the top choice. For this we use *preference lists*.

*Plurality* is a voting system in which only the first choice of each voter is considered. This system is currently used to select the United States president, and can fail the CWC in some cases.

*Borda Count* takes the entire set of preferences into consideration by assigning points to each candidate based on where they are ranked. The Associated Press and USA Today college sports polls use the Borda Count. It is known to be able to fail the IIA.

*Sequential Pairwise Voting* seeks to eliminate problems with the CWC by having candidates go head-to-head. An *agenda* is used to determine the order of the head-to-head rounds. Unfortunately, even though it satisfies CWC, this voting method fails the other three criteria, most notably the Pareto condition.

The *Hare System* seeks to eliminate least-preferred candidates until a winner is determined. A method similar to plurality is used to rank the candidates, and the ones with the least number of votes are eliminated. The process is repeated until everyone (possibly only one candidate) has the same number of votes. While this system sounds effective, it can fail Monotonicity.

One inherent problem in these voting methods is the susceptibility to *insincere voting*. Because of the above mentioned flaws, a voter can affect the outcome of an election in their favor by not voting the way they actually prefer. This insincerity makes it difficult to interpret the ballots.

An alternative approach is used in *approval voting*, where a voter simply lists all of the candidates of which they approve, without the need to rank them. This attempts to eliminate some of the insincerity.

Unfortunately, Kenneth Arrow’s *Impossibility Theorem* states that it is impossible to come up with a voting system for three or more candidates that satisfies all four of the criteria all of the time. We can still hope to find better solutions, however, by making the failure less likely to actually happen.

## 6 Chapter 13 – Weighted Voting

In the last chapter, we looked at a situation in which not every voter is equal. Instead, each voter is given a certain number of equal votes, and a *quota* of votes necessary to win is established. This

reflects situations like the Electoral College and corporate stockholders, where the number of votes each member of the system has is based on some external factors (like population or ownership).

Groups of voters who decide to vote together are known as *coalitions*. Depending on the way in which it is formed, a coalition may be *winning*, *losing*, *blocking*, or some combination of those. If a single person can form a winning coalition, they are referred to as a *dictator*; if they form a blocking coalition, they are said to have *veto power*.

Deciding who actually has the ability to influence votes can be harder than simply counting the number of votes each person has. Two different methods can be used to find the amount of influence a voter has: the *Banzhaf Power Index* and the *Shapley-Shubik Index*.

The Banzhaf Power Index uses the notion of a *critical voter* to determine power. If a coalition is a winning coalition, and a voter leaves the coalition, making it a losing coalition, the voter that left is considered critical. Counting how many times a voter is critical will give the Banzhaf Power Index. Recall that it is necessary to double all of the numbers in this index.

On the other hand, the Shapley-Shubik Index is based on the idea of a *pivotal voter*. If we draw a spectrum of opinions on a matter, and place each voter where they lie on that spectrum, we end up with a *permutation* of the voters. We add the voters one at a time to form a coalition, starting with the voter most in favor of the measure. The first voter in a permutation that makes the coalition a winning coalition is called pivotal. The number of times a voter is pivotal determines the Shapley-Shubik Index.

When we look at these weighted voting systems, we want to know how the power is balanced. For a set number of voters, there are usually only a few different ways we can distribute the power. If two voting systems have the same winning coalitions, they are said to be equivalent. The Banzhaf Power Index can be used to figure out whether two systems are equivalent. We can also use *minimal winning coalitions*, in which every voter is critical. These coalitions completely determine the weighted voting system.

## 7 Sudoku

Finally we discussed logic puzzles and the importance of analytical thinking. While puzzles like *Sudoku* might not appear to have “real-world” significance, they train our brains to think in an orderly fashion.

Sudoku puzzles involve placing numbers into squares, following some easy rules. For a 2 by 2 puzzle, the numbers 1, 2, 3, and 4 are placed into the squares in such a way that every row, every column, and every block contains exactly one of each number. The puzzles begin with a few numbers placed in the boxes, and solving the puzzle involves adding the remaining numbers.

As the puzzles get larger, they can become more difficult. However, with practice, 3 by 3 puzzles such as those found in newspapers can be done relatively easily.