Math 203 – Review for Exam I

Your first exam covers Chapters 1-3 of your textbook. This review sheet is meant to help you study for the exam, but is not meant to be a substitute for studying from the book. In particular, I have tried to include all the topics we covered, but it is possible I accidently left something out. If you think I have left something out, tell me. It is not safe to assume that any omitted topics are not covered on the exam.

Good luck studying!

Graphs: You should be completely comfortable with the following concepts.
• Know what graphs, edges and vertices are.
• The *valence* of a vertex is the number of edges attached to the vertex.
• A *circuit* is any path in a graph that begins and ends at the same vertex.

One special type of circuit is an *Euler circuit.* This is a circuit which passes along each edge exactly once. We have a theorem which tells us precisely when it is possible to find an Euler circuit on a graph. It is called *Euler's Theorem* and it says that a graph has an Euler circuit if and only if it is connected and every vertex has even valence.

Here is an application using these concepts: You have some job to do, like collecting garbage, emptying parking meters, etc. To complete your job you need to travel down every road in town with as little backtracking as possible. (Quick question: When is it possible to finish the job without having to backtrack at all?) How do you solve this problem? Simple. Make a graph that represents the streets and intersections of the town. Then "Eulerize" the graph by adding as few edges as possible. Finally, find an Euler circuit in the modified graph and use this circuit to carry out your job. Another quick question: When you add an edge to the graph representing the town, what does that mean in terms of backtracking when you actually do the job?

More Graphs: You should also know the following terms.
• A *complete graph* is a graph which has an edge joining every pair of vertices.
• A *Hamiltonian circuit* is a circuit in a graph which passes through every vertex exactly once.

Although we have no theorem like Euler's Theorem for Hamiltonian circuits, it is easy to see that a complete graph always has a Hamiltonian circuit. In fact, if you just choose the vertices in any order, you can find a Hamiltonian circuit which visits the vertices in this order.
• A *weighted graph* is a graph with a number attached to each edge. In applications, this number can represent different things such as distance, cost in dollars, etc.
• The *Traveling Salesman Problem* is the problem of finding a minimal cost Hamiltonian circuit on a weighted (often complete) graph.

In a large graph, there are zillions of Hamiltonian circuits, so trying all of them (called the *Brute Force Method*) is not a practical way to solve a Traveling Salesman Problem, even with a computer. Unfortunately, there is no known method that finds the Hamiltonian circuit with least weight in a reasonable amount of time. Therefore, we usually try to make do with an algorithm that doesn't take too much time and finds a Hamiltonian circuit with "low" weight. We covered three algorithms (including Brute Force) for finding a low-cost Hamiltonian circuit.
•*Brute Force*: List all the possible Hamiltonian circuits on the graph, find the weight of each, and choose the best one. This method always gives the best possible solution, but it is highly impractical.
• *Nearest Neighbor*: Pick a starting vertex, and form a circuit which starts and ends at that vertex by always traveling along the edge coming out of your vertex that has the lowest weight, being careful to never visit a vertex twice until you've been to all of them once. This method is quick and easy, but might not give the best possible solution. It will usually give a solution which is pretty good, though.

• *Sorted Edges*: "Grab" edges until you have a complete circuit. The edge you grab will always be the one with the lowest weight that hasn't been grabbed yet, as long as grabbing it will not cause you to form a circuit that doesn't pass through all the vertices or cause a vertex to be touched by a third edge. Like the Nearest Neighbor algorithm, the Sorted Edges algorithm is quick and easy and gives a reasonable solution. It may not give the best possible solution.

Here is a typical application of these ideas: A person needs to travel to several locations, in no particular order. His or her goal is to choose an order in which to visit these locations that minimizes "cost". (Remember, we use the term cost loosely; it can refer to time, distance, money, etc.) Here are some appropriate specific examples:

• A regional manager needs to visit each store under her jurisdiction periodically.

• A phone company employee needs to empty the change from each phone booth in a city.

For another type of application, consider the situation in which one needs to program a machine to drill a bunch of holes in a metal plate. What is the relevant graph to consider in this example? What "cost" does one most likely want to minimize?

Yet more graphs: Here are some more graph related terms you need to know.

• A *tree* is a connected graph that has no circuits.

• A *spanning tree* on a graph is a subgraph that contains all of the vertices of the graph and is a tree. (Remember that a subgraph of a graph is a graph formed by some of the edges and some of the vertices of the original graph.)

• A *minimum-cost spanning tree* on a weighted graph is a spanning tree that has the lowest total cost – that is, a spanning tree such that the total of all of the numbers on the edges used is as small as possible.

Unlike the situation where you're trying to find a minimum cost Hamilton circuit, there is an algorithm which is quick and easy and which will always give the minimum-cost spanning tree for a weighted graph. It is called *Kruskal's algorithm*, and it is very similar to the Sorted Edges algorithm for finding a reasonable Hamiltonian circuit. The idea is to list the edges by weight, with the edge at the top of the list having the lowest weight. Starting at the top, pick an edge from the list. Add this edge to the tree you are building if it does *not* cause a circuit to arise. Then move on to the next edge on the list and repeat the process. When you have gone through the whole list, you will have formed a minimum-cost spanning tree.

A typical application of minimum-cost spanning trees arises when you want to build a phone network. The vertices represent places you need to hook up to the network and the edges represent phone lines you can build. The weights of the edges represent the costs of building the corresponding line. A minimum-cost spanning tree in this example allows you to connect everybody to the network as cheaply as possible.

Even more graphs! Here are just a few more terms.

• A *digraph*, or a *directed graph*, is a graph in which the edges have directions assigned to them. They can represent one-way streets, for example, or they can represent an order in which some tasks (represented by vertices) need to be done.

• An *order-requirement digraph* is a digraph with a number attached to each vertex. Think of the vertices as representing tasks that need to be done, the numbers as representing the amount of time needed to complete the given task, and the edges as telling you which tasks must be done before which others. That is to say, if there is an arrow from vertex A to vertex B, then task B cannot be begun until task A has been completely finished.

• The *critical path* on an order-requirement digraph is the longest (i.e., the most time consuming) path on the graph. Its length is the total amount of time needed, if infinitely many workers are available, to do all the tasks in the project represented by the order-requirement digraph.

Scheduling Problems While finding a critical path will tell you the minimum amount of time needed to complete a project "in a perfect world", sometimes our world is not perfect. Suppose, for example that we don't have infinitely many workers. Instead, we have a certain number of them (machines, computers, people, etc.). For simplicity's sake we assume that each perform at exactly the same rate. We also have an order-requirement digraph. Each task can be performed by any of the processors. The goal is to build a schedule with the earliest possible completion time. As with the traveling salesman problem, it is impossible in practice to check every possible schedule to see which is best, and therefore it is usually impossible to find a best possible schedule in practice. Instead, we must make do with heuristic algorithms that give "pretty good" schedules. The algorithm we discussed was called the List Processing Algorithm.

• The *List Processing Algorithm* is used to schedule tasks on a finite number of processors. The idea is that we assign the first available task to the first available processor. The order in which tasks are assigned is given by an order-requirement digraph and by an ordered list of tasks. When a processor becomes available, we set it to work on the first task in the list which is currently "ready", as determined by the order-requirement digraph. We continue until all tasks are assigned. Notice that it is possible that a processor may need to rest (or be idle) if there are no tasks available for it to work on.

• If a priority list is not already given to use, we can use *Critical Path Scheduling* to come up with one.

• In some cases, there is no order-requirement digraph to work with because all the tasks are *independent*. This means that all tasks are ready to be done at the beginning of the project. In this case, you look only at the ordered list of tasks when deciding how to schedule the tasks on the processors. If we use Critical Path Scheduling on independent tasks, the list we get just lists the tasks in decreasing order of time they take to complete.

Bin Packing One situation where bin packing techniques apply is in figuring out how much lumber to buy to build bookshelves – this is explained well on pages 102-103 of your text. Another would be the following. Suppose you have several differently sized items to pack into boxes. You can always go out and buy more boxes, but you want to use as few as possible, so you want to fill them as full as possible. How do you decide which items to put in which boxes?

   In fact, there is again (as in the case of the Traveling Salesman Problem or Scheduling Problems) no algorithm for finding the best bin packing which can be done in any reasonable amount of time. (You could try all possible packings, but this would take a very long time.) Instead, we have six "heuristic" algorithms to choose from. Each of these gives a reasonably good packing in a short amount of time. For each of the first three, assume you are given an ordered list of sizes for your items which you need to pack in your bins.

• The *next fit* algorithm says: Put items into the first bin until the next item doesn't fit. Close off that bin forever and open up another one. Fill it for as long as you can, then close it off and open a third. Keep doing this until all items are packed.

• The *first fit* algorithm says: Put items into the first bin (Bin 1) until the next item doesn't fit. Then open a new bin (Bin 2), but don't close off the first one. Put each item from your list into the lowest-numbered bin in which it will fit, opening new bins only when an item won't fit into any of the bins you've already opened.

• The *worst fit* algorithm begins like the first fit one. However, instead of putting each item from the list into the lowest-numbered bin in which it will fit, put it into the bin (which is already opened and) which has the most room left in it.

   There are also *decreasing* versions of all these algorithms. These are just like the three discussed above, but first you re-order your list of items from biggest to smallest, so that you pack the big items first and the small items last.