Algorithms
Computer Science 140 & Mathematics 168
Spring 2007
Homework 6b
Due Tuesday, February 27

- We are now covering material from Chapter 3 in our book.
- On this problem set, you are welcome to refer to anything that we covered in class by simply saying "in class we stated (or showed) that blah, blah, blah." This can save you some time and space.
- Exam 1 will be given out in class on Thursday, March 1 and will be due on Tuesday, March 6 at 5 PM in the CS office. The exam has no time limit. Here's what you may use on the exam:
  - Your own homework sets.
  - The solution sets that we've provided.
  - Class notes with your own markup on them. If you took notes on your laptop, you may use those.
  - Any additional notes that you prepare before the exam.

  No other materials may be used.

1. **[8 Points] Cycles in Directed Graphs.** Describe a simple algorithm for determining if a directed graph contains a cycle. The graph is assumed to be represented with an adjacency list. Explain why your algorithm works and carefully derive its running time. For full credit, your algorithm should be as fast as possible.

2. **[22 Points] Napquest Revisited!** A few weeks ago, you devised an algorithm for Napquest that finds the shortest path from one vertex to another in a very special kind of graph.

   Napquest is ready for more! Now they want you to design an algorithm that finds the length of the shortest path between any pair of vertices in an arbitrary directed graph in which every edge has a positive weight (distance) associated with it. By "shortest path" we do not mean the number of edges, we mean the total sum of the weights on the edges in the path!

The graph is represented by an adjacency matrix. However, instead of having 0's and 1's as entries in the matrix, the entries of the matrix specify the weight on a given edge. In particular, if there is a directed edge from vertex i to vertex j with weight d, then the entry in row i and column j of the matrix will be d. If there is no edge from vertex i to vertex j, we put the value ∞ in that entry of the matrix. (You may assume that our programming language supports arithmetic with ∞ so, for example, $\infty + 42 = \infty$ and ∞ is greater than any integer when a comparison is performed.)

So, the program has access to a $n \times n$ adjacency matrix, A, indicating the directed edges and their weights. Assume that matrix A is global, so we don't have to pass it in explicitly to our COMPUTE-DISTANCES function. Instead, COMPUTE-DISTANCES takes just 3 arguments as input: The start vertex, s, the destination vertex, d, and the maximum number of permitted edges p, in the path. The function then returns the length of the shortest path (in terms of total weight) that uses p or fewer edges. (What's with the "p or fewer edges business?" Read on!)

(a) Argue briefly that if there are n vertices in the graph, then the absolute shortest path between two vertices involves at most $n-1$ edges. Remember, all of the edge weights are positive.

(b) Now, give a recursive algorithm for COMPUTE-DISTANCES(s, d, p) where s is the start vertex, d is the destination vertex, and p is the maximum number of edges permitted. Note that when we invoked this algorithm, we will use $p = n - 1$, but your algorithm should work for any p. *Hint: Use it or lose it!*

(c) Now describe the dynamic programming formulation for making this recursive algorithm efficient. In particular, describe what the table looks like, what the cells represent, and the order in which you fill them in. Notice that your table should actually allow you to see the shortest path between every pair of vertices in the graph.

(d) What is the running time of your algorithm? Explain how you derived this.

(e) Napquest would actually like to know the distances between every pair of the n vertices. Notice that you've already done this! There's nothing to write here.

3. **[25 Points] Graph Diameter!** In this problem we will consider connected undirected acyclic graphs (graphs that have no cycles). The graphs are assumed to be represented with adjacency lists. The distance between two vertices in such a graph is just the minimum number of edges on a path between the vertices. The *diameter* of the graph is the maximum distance between all pairs of vertices. Make sure that you understand this definition well before proceeding!

   (a) Give a precise description of an algorithm for computing the diameter of an undirected acyclic graph. The description can be in English or high-level pseudo-code. (Think about efficiency. The amount of credit that you get for this problem will be dependent both on the correctness and the asymptotic run time of your algorithm.)

   (b) Explain briefly but convincingly why your algorithm is correct.

   (c) Carefully derive the asymptotic running time of your algorithm. Be sure to account for all of the work that your algorithm incurs. You will need to think about how the adjacency list representing the graph is used in the algorithm.

4. **[25 Points] 2SAT!** 2-Satisfiability, often called 2SAT, is a classic problem in logic that arises in applications in Artificial Intelligence and elsewhere. The problem goes like this: We are given a collection of n boolean variables $x_1, \ldots, x_n$. (That is each variable can take the value TRUE or FALSE.) In addition, we are given clauses where each clause comprises the disjunctions (logical OR) of exactly two literals, where a literal is a variable or its negation. We wish to find an assignment of TRUE or FALSE to each variable such that all of the clauses are satisfied. For example, here is an instance of 2SAT:

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

   Note that $\vee$ is the logical OR, $\wedge$ is the logical AND and $\overline{x}$ denotes the logical negation of variable x. Saying that we want to satisfy each of the clauses is exactly the same thing as saying we want to satisfy the conjunction (AND) of all of the clauses. In this example, we can satisfy the expression by making x_1 be TRUE, x_2 be FALSE, and x_3 be TRUE.

As we'll see in a few weeks, the evil twin of this problem, 3SAT, in which each clause contains the disjunction of exactly 3 literals is NP-complete: There is no known polynomial-time algorithm for that problem and there is strong evidence that suggests that no polynomial-time algorithm exists.

Amazingly, 2SAT is solvable in polynomial time and you'll show that here! In particular, in this problem, we will see that 2SAT can be solved using graph algorithms instead.

(a) Consider the following instance of 2SAT:

$$(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$$

Show that this instance is satisfiable by giving a satisfying assignment for the variables.

(b) Now consider the following instance of 2SAT:

$$(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$$

Explain briefly why this instance of 2SAT is not satisfiable.

(c) Given an instance of 2SAT, let's construct a corresponding directed graph as follows: For each variable x_i that appears in the 2SAT instance, construct a *pair* of vertices, one labeled x_i and the other labeled $\overline{x_i}$. For every clause of the form $(a \vee b)$ in the 2SAT instance (where a and b are variables which may or may not be negated), place a directed edge from vertex $\overline{a}$ to vertex b and also a directed edge from vertex $\overline{b}$ to vertex a. For example, if we had a clause $(x_1 \vee \overline{x_3})$ then $a = x_1$ and $b = \overline{x_3}$. Therefore, we would place a directed edge from vertex $\overline{x_1}$ to vertex $\overline{x_3}$ as well as a directed edge from vertex x_3 to vertex x_1. In this example, you should interpret the edge from vertex $\overline{x_1}$ to vertex $\overline{x_3}$ to mean "if $\overline{x_1}$ is true (that is, x_1 is false) then $\overline{x_3}$ must be true." Similarly, the edge from vertex x_3 to the vertex x_1 is interpreted as "if x_3 is true then x_1 must be true."

Construct this directed graph for the 2SAT instance in part (a). This graph should contain 4 vertices 6 edges. Notice that there is no path from vertex x_1 to vertex $\overline{x_1}$.

(d) Notice that in the graph you constructed there is a path from vertex $\overline{x_1}$ to vertex x_1. Clearly explain why *the existence of this path* implies that a satisfying assignment for this instance of 2SAT cannot have $\overline{x_1}$ be **true** (that is, x_1 cannot be **false**). Be very clear and precise about your reasoning here.

(e) Notice that there does *not* exist a path in this graph from vertex x_1 to vertex $\overline{x_1}$. Notice also that in your satisfying assignment for this instance, x_1 was set to **true**. Clearly explain why this graph tells us that x_1 should be assigned **true** if the instance has any hope of being satisfied.

(f) What does the graph tell you about the value that should be assigned to variable x_2? Explain.

(g) Now, construct the graph for the 2SAT problem in part (b). What property does this graph have that forces you to conclude that the 2SAT instance is not satisfiable? Be precise.

(h) Next, write down a conjecture that starts as follows: "A 2SAT instance is satisfiable if and only if the corresponding directed graph has the property that blah, blah, blah." Fill in the "blah, blah, blah" with mathematically precise language.

(i) Now, prove your conjecture. This is the main part of this problem (it's also worth most of the points!) and will take a few paragraphs to prove. Note that it is an "if and only if" proof, which means that there are two things to show. In this proof, the fact that two edges were introduced for each clause will be absolutely critical. You'll need to use this fact!

(j) Now, describe an algorithm to determine whether or not a 2SAT instance is satisfiable. What is the running time of your algorithm as a function of the number of variables and clauses? Explain the running time carefully. As long as the running time is polynomial in the number of variables and clauses, everything is fine!