Algorithms
Computer Science 140 & Mathematics 168
Spring 2009
Homework 7b and 8a
Due Thursday, March 12 in class

- This is the last assignment before spring break. It combines assignment 7b with 8a and is due at the beginning of class on Thursday, March 12.
- Ran will be out-of-town and will therefore have no office hours on Monday, March 9.

1. **[30 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!

2. **[15 Points] Proof of Dijkstra's Algorithm.** Write the proof of correctness of Dijkstra's Algorithm in your own words - except now generalize it slightly to permit edge weights to be non-negative *but possibly 0*. Please use the proof technique that we developed in class. Note that there are other proofs of correctness but I would like you to use ours (it's simpler and more elegant than the proofs that often are presented in textbooks.)

   In your proof, please use the following notation from class: Let s denote the start vertex (we are trying to find the shortest path lengths from s to every other vertex) and let $\ell(u)$ denote the label on vertex u after a given iteration of the algorithm.

3. **[25 Points] Hurts' On-board Navigation System Re-revisited.** Hurts Car Rental has designed a new generation of alternative fuel vehicles. The new vehicles use a special fuel comprising a finely minced mixture of the "Algorithms" textbook, chocolate fudge Pop Tarts, Spam, Spam, Spam, and Mountain Dew. Due to this rather unusual fuel requirement, there are only certain cities in the country where the vehicles can be refueled. Thus, to get from the start city to the destination city, the driver must plan a route that ensures that the car can be refueled along the way. (Note that this problem differs in several important ways from the first Hurts problem on an earlier assignment of a few weeks ago. First, we no longer use fuel packs - we can now fill up with fuel like a normal car. Second, the graph is no longer a straight line - it's a general

graph. Third, we only care about minimizing the distance travelled, not about the cost of fuel.)

The on-board computer on such a vehicle contains a weighted directed graph in which the vertices represent cities, the directed edges represent one-way roads, and the weights on the edges represent the (strictly positive) lengths of the roads. The graph, of course, can have cycles!

The computer knows which select cities have filling stations. To use the navigation system, the user will enter the starting city, the destination city, and the range of the vehicle on a full tank. (You may assume that the starting city and the destination cities, being Hurts rental cities, both have filling stations - and that the car starts with a full tank of fuel.) The computer will respond with the *shortest* route from the starting city to the destination city that ensures that the vehicle doesn't run out of fuel or determines that no such route exists.

You should assume that there is some constant C such that every vertex has in-degree and out-degree (number of entering edges and number of exiting edges, respectively) at most C.

(a) **Describe** and **analyze the running time** of an efficient algorithm for determining such a path. There are several ways to do this problem, but if you break this problem up into a couple intermediate steps you will probably find that the important problem of proving correctness (in the next part of the problem) is relatively easy.

(b) *Prove* that your algorithm finds an optimal solution. Be careful and precise. A very short but rigorous proof is possible if your algorithm was designed carefully.

For full credit, your algorithm must be both provably correct and fast. Your algorithm may be built up of several algorithms which we have seen in class. You may apply any algorithms that we have seen in class and need not re-prove their correctness. However, if you use an algorithm that is at all different from one that we've examined in class, you must indeed prove its correctness

4. [**10 Points**] **Forrester's MST Algorithm!** Professor I. M. A. Forrester of the Pasadena Institute of Technology has developed a new algorithm for the minimum spanning tree (MST) problem on completely connected graphs (graphs in which there is an edge between every pair of vertices). It is based on the divide-and-conquer paradigm and it goes like this: Given a *completely connected graph* $G = (V, E)$, partition the set of vertices V into two sets V_1

and V_2 whose sizes are equal (or differ by at most 1). Let E_1 be the edges incident only on vertices in V_1 and let E_2 be the edges incident only on vertices in V_2. Recursively solve the MST problem on the two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge that crosses the cut (V_1, V_2) and use that edge to unite the two spanning trees found by the recursive calls.

Professor Forrester did not prove the correctness of his algorithm. Your job is to either carefully prove the correctness or give a counterexample that shows that the algorithm is not correct.

5. [**20 Points**] **Barůvka's Algorithm!** In class we mentioned that the first algorithm for computing minimum spanning trees was published by the Czech mathematician Otakar Barůvka in 1926 and was used for laying out electrical networks in Czechoslovakia (actually, in a part of Czechoslovakia known as Bohemia - but that detail will probably not make a large impact in your solution to this problem). The algorithm is given on the next page:

```
A = {};  \\ Comment:  A is a subset of a minimum spanning tree
Consider the n  vertices in the graph as n connected components;
while A contains fewer than n-1 edges
{
   for each connected component C
   {
      Find the least weight edge (u,v) with one vertex in C and
        one vertex not in C;
        Indicate that edge (u,v) is "chosen" but do not add it yet to A;
   }
   Add all "chosen" edges to A;
   Compute the new connected components;
}
return A \\ Comment:  This is intended to be a MST!
```

Notice that if C_1 and C_2 are two different connected components before we begin the for loop, then inside the for loop the algorithm will choose the least weight edge coming out of component C_1 and also the least weight edge coming out of C_2. The edge chosen by C_1 might join C_1 and C_2 into a new connected component, but this new connected component will not be discovered until the for loop has ended! In other words, both C_1 and C_2 will each get an opportunity to choose the least weight edges coming out of their components!

(a) Give a counter-example that shows that Borůvka's Algorithm doesn't work!! (You might find it useful to use the fact that some edges in the graph may have the same weight.) Show your counter-example graph and explain carefully why Borůvka's Algorithm would not compute a minimum spanning tree in this case.

(b) Now assume that no two edges in the graph have the same weight. By the OPTIONAL BONUS PROBLEM below, such a graph has exactly one minimum spanning tree (you may just use this fact here, although you are encouraged to prove it in the bonus problem!). Under this assumption, prove that Borůvka's Algorithm is correct.

(c) Why doesn't your proof from part (b) work if some edges in the graph have the same weights?

(d) How could Borůvka's Algorithm be modified slightly to work in the most general case that edge weights are not necessarily distinct? Explain briefly why this modification preserves the correctness of the algorithm.

(e) Describe an implementation of Borůvka's Algorithm that is as fast as you can make it. The description should be in clear English, but it should indicate the data structures that would be used to support the algorithm and give a careful derivation of the asymptotic worst-case running time using these data structures.

6. [**Optional 10 Point Bonus Problem**] Let G be a connected undirected graph in which each edge has a distinct edge weight (that is, no two edges have the same weight). Prove that there is a *unique* minimum spanning tree in the graph. (This problem does not require that you know any more graph theory than we have already used in class.)