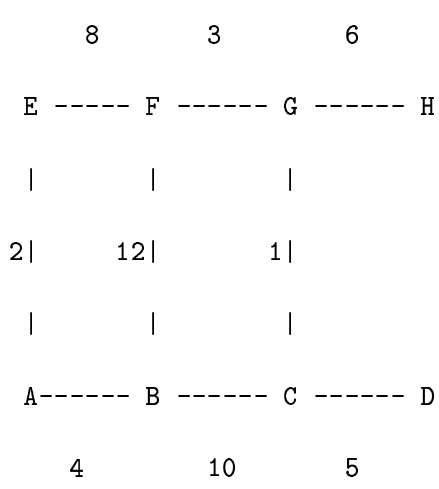
Computer Science 409, Spring 2001: Final Exam Solutions

1. [8 points: 4+4] Consider the following weighted graph:



   (a) List the first 5 edges added to the minimum spanning tree by Kruskal's algorithm.
   Solution: GC, EA, FG, AB, and CD

   (b) List the first 5 edges added to the minimum spanning tree by Prim's algorithm starting at vertex A.
   Solution: EA, AB, EF, FG, and GC.

2. [7 points: 1+3+3] Suppose you have started running Dijkstra's algorithm on an undirected graph $G(V, E)$ starting at a. Vertices a, b, and c have been completed (i.e., are no longer in the priority queue), and have distances 0, 2, and 5, respectively. Vertices d, e, f, and g are still in the queue; these vertices have current distances (i.e. d values) of 11, 5, 17, and ∞, respectively. Edges (d, f), (e, f), (e, g), and (g, f) have weights 3, 10, 3, and 3, respectively. Edges (d, e) and (d, g) have weight ∞ (i.e, they don't exist).

   (a) Which vertex is processed next?
   Solution: You proceses vertex e next.

   (b) After processing that vertex, what will be the new d values of d, e, f, and g?
   Solution: The new values for d, e, f, and g will be 11, 5, 15, and 8, respectively.

(c) For each vertex in the graph, what is the value of d (i.e., the shortest distance) when the algorithm terminates?

Solution: For a, it's 0, for b, it's 2, for c and e, it's 5, forr g, it's 8, for d and f, it's 11.

3. [8 points] You have 400 quarters, 1,000 dimes, 2,000 nickels, and 10,000 pennies. Describe a greedy algorithm for making change for any amount less than \$100 using the least number of coins. (No need to write detailed code; just explain how the algorithm works.) Prove that your algorithm uses the least number of coins in making change.

Solution: If you need to make change for n cents (convert dollars and cents to cents), then first use as many quarters as you can ($\lfloor n/25 \rfloor$, to be exact, then as many dimes as you can for the remaining amount (the remaining amount is $n - 25\lfloor n/25 \rfloor$, so you can use ($\lfloor (n - 25\lfloor n/25)/10 \rfloor$ dimes), and then pay the remaining amount in pennies. Here's the code. In the code, I use n_1, n_2, n_3, and n_4 for the number of quarters, dimes, nickels, and pennies used when using the minimal number of coins to make change; n is the amount of the change you need to make.

```
1  n_1 ← 0;
2  n_2 ← 0;
3  n_2 ← 0;
4  while n > 0 do
5      if n ≥ 25 then n_1 ← n_1 + 1; n ← n − 25
7        else if n ≥ 10 then n_2 ← n_2 + 1; n ← n − 10
7             else if n ≥ 5 then n_3 ← n_3 + 1; n ← n − 5
8        else n_←n; n ← 0
```

It remains to show that this algorithm gives change using the minimal number of coins. Suppose n_1', n_2', n_3', and n_4' are the number of quarters, dimes, nickels, and pennies used when making chnage using the minimal number of coins. Note that $n_2 \leq 2$, since if $n_2 \geq 3$, we can replace three of the dimes by a nickel and a quarter, thus making change using fewer coins. Similarly, $n_3 \leq 1$, since if $n_3 \geq 2$, we can replace two of the nickels by a dime. Finally, $n_4 \leq 4$. Moreover, if $n_2 = 2$, then $n_3 = 0$, otherwise we can replce two dimes and a nickel by a quarter. But that means we have less than 25 cents altogether using n_2' dimes, n_3' nickels, and n_4' pennies, which means that $n_1' = n_1$. We also have at most less than 10 cents in nickels and pennies, so $n_2 = n_2'$. Finally, we have less than 5 cents in pennies, so $n_3 = n_3'$. It follows that $n_4 = n_4'$, since $n = 25n_1 + 10n_2 + 5n_3 + n_4 = 25n_1' + 10n_2' + 5n_3' + n_4'$.

Remark: Everyone got the greedy algorithm right (so everyone got at least 4 on the question). However, no one really gave a careful enough proof of correctness. The greedy algorithm wouldn't work if we had only quarters, dimes, and pennies (but no nickels). For example, suppose we had to return 30 cents. Then using the algorithm above, we would use one quarter and 5 pennies. It's clearly better to use three dimes. Whatever argument you give better fail if you don't have nickels. Almost all the arguments actually given worked perfectly well in this case too.

4. [9 points: 1 + 3 + 3 + 2]

(a) What is a *flow network*.

Solution: A flow network is a graph where each edge is labeled by a positive number called a *capacity*. There are also two distinguished nodes called the *source* and the *sink*. [Remark: a flow network is *not* a flow.]

(b) What is a *flow* in a flow network?

Solution: Given a flow network $G = (V, E)$ with capacity function c, a flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ satisfying three properties:

- $f(u, v) \leq c(u, v)$ (capacity constraint)
- $f(u, v) = -f(v, u)$ (skew symmetry)
- $\sum_{v \in V} f(u, v) = 0$ if u is neither the source nor the sink.

(c) Given a flow f in a flow network G, what is the residual network G_f?

Solution: If $G = (V, E)$ is a flow network with a capacity function c, let $c_f(u, v) = c(u, v) - f(u, v)$. G_f is the flow network with vertices V, edges $E_f = \{(u, v) : c_f(u, v) > 0$, and capacity c_f.

(d) What is an *augmenting path* with respect to f (where f is a flow in flow network G).

Solution: An augmenting path with respect to f is a path in G_f.

5. [8 points]

(a) Suppose we wanted to represent a large binary tree as a graph. Is it better to store it as an adjacency list or as an adjacency matrix? Why?

Solution: An adjacency list is better, since has at most two edges going out (if it's a directed tree) or three if it's not. Thus, less storage is needed to represent the tree using an adjacency list.

(b) What is difference between amortized time and expected time?

Solution: With amortized time we consider the average cost per operation if many operations are performed. [Remark: a lot of the answers here were pretty far off ...] With expected time, we consider the average (expected) running time of the algorithm assuming a distribution over the inputs.

(c) True or false: if $A \leq_P B$ and A is NP-hard, then B is NP-hard. Why?

Solution: True. if A is NP-hard, then all problems in NP can be reduced to A. Since $\leq_P$ is transitive, then all problems in NP can be reduced to B, so B is NP-hard.

(d) True or false: If $f(n) = O(g(n)$ and $h(n) = O(g(n)$, then $f(n) = O(h(n))$.

Solution: False. $O(g(n))$ just says that $g(n)$ is an upper bound. For example, if $h(n) = n$, and $f(n) = g(n) = n^2$, then $f(n) = O(g(n))$ and $h(n) = O(g(n))$, but $f(n) \neq O(h(n))$.

6. [10 points: 2+2+6] Suppose that $L \subseteq \{0, 1\}^*$.

(a) Carefully define L^*.

Solution: $L^* = \epsilon \cup L \cup L^2 \cup \ldots$, where ϵ is the empty string, and L^k consists of all strings of the form $x_1 x_2 \ldots x_k$, where $x_1, \ldots, x_k \in L$.

(b) If $L_1 = 0, 01$ and $L_2 = 1, 10$, what is $L_1 L_2$?

Solution: $L_1 L_2 = \{01, 010, 011, 0110\}$.

(c) Show that if L is decidable in polynomial time, then so is L^*. (Hint: use dynamic programming. Again, there's no need to write detailed code.)

Solution: Given $x = x_1 \dots x_n$, let $x^0 = \epsilon$, $x^k = x_1 \dots x_k$ and let $x^{j,k} = x_j \dots x_k$ (for $1 \leq j \leq k$. The idea of the algorithm is to check if $x^k \in L^*$. Note that $x^{k+1} \in L^*$ iff there is some $j \geq 0$ such that $x^j \in L^*$ and $x^{j,k+1} \in L$. With this in mind, here is the algorithm. In the algorithm T is a table such that $T(j) = 1$ iff $x^j \in L^*$.

```
1  n ← |x|
2  for k = 0 to n do
3      if k = 0 then T(k) ← 1 else T(k) ← 0
4      for j = 0 to k do
5          if T(j) = 1 and x^{j,k} ∈ L then T(k) ← 1
6  return T(n)
```

Running time: Suppose there is an algorithm to decide if $y \in L$ that runs in time $O(|y|^m)$. The algorithm goes through the loop at most n^2. On each iteration through the loop, it may compute with $y \in L$ for some y with $|y| \leq |x|$. Thus, the computation of whether $y \in L$ takes at most $O(|x|^m)$ steps. Thus, the running time of the algorithm on input x is $O(|x|^{k+2})$. As for correctness, it clearly suffices to show that by induction on k that after the $(k+1)$st iteration, $T(k') = 1$ iff $x^{k'} \in L^*$, for all $k' \leq k$. For the base case, $x^0 = \epsilon \in L^*$ and $T(0)$ is set to 1 on the first iteration of the algorithm. Suppose the induction hypothesis hold for $k-1$. I now prove the inductive step. Notice that the only value of T that changes in the $(k+1)$st iteration is $T(k)$. Clearly $x^k \in L^*$ iff there is a prefix x^j of x^k such that $x^j \in L^*$ and $x^{j,k} \in L$. So, by line 5 of the algorithm, if $x^k \in L^*$, then $T(k)$ is set to 1 in the $(k+1)$st iteration. On the other hand, if $x^k \notin L^*$, the inner loop never sets $T(k)$ to 1; it remains at 0. Since the algorithm returns $T(n)$, which is 1 iff $x in L^*$, we are done.

7. [8 points: 2 + 1 + 5]

(a) What is a *Hamiltonian path*?

Solution: A Hamiltonian path in a graph $G = (V, E)$ is a path that goes through each vertex in V exactly once.

(b) What is a directed acyclic graph?

Solution: A directed acyclic graph is a directed graph $G = (V, E)$ that has no cycles; that is, there is no directed path of the form $(v_0, v_1, \dots, v_k)$ where $v_k = v_1$.

(c) Show that the Hamiltonian-path problem can be solved in polynomial time on directed acyclic graphs. (Again, you don't have to write detailed code. Just explain in English how the algorithm works.) Prove that your algorithm is correct. (You may use any algorithms discussed in class. Hint: think about topological sort.)

Solution: Suppose $G = (V, E)$ is a directed acyclic graph. First step: topologically sort the vertices in G. (This can be done in polynomial time; we discussed an algorithm in class.) Let $v_0, \dots, v_k$ be the list of vertices in topologically sorted order. Then check if

$(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$. (Clearly this can be done in linear time.) Clearly if $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, k-1$, then the graph has a Hamiltonian path, namely $(v_0, \ldots, v_k)$. Conversely, if the graph has a Hamiltonian path, say $(v_{i_0}, \ldots, v_{i_k})$, where $i_0, \ldots, i_k$ is some permutation of $1, \ldots, k$. We must have $i_j < i_{j+1}$ for all $j = 0, \ldots, k-1$, since the only edges in G from a lower-numbered edge in the topological sort to a higher-numbered edge. The only way that this can happen is if $i_j = j$, that is, the Hamiltonian path is $(v_0, \ldots, v_{k-1})$, in which case $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, k-1$.

[Remark: you can't compute a Hamiltonian path using DFS, as many people tried to do.]

8. [11 points: 2 + 1 + 3 + 5] The *low degree spanning tree problem* is as follows: Given a graph G find a spanning tree where the maximum degree of each node is as small as possible.

   (a) Convert this optimization problem to a decision problem.
   Solution: The decision problem version is: does G have a spanning tree in which each node has degree at most k.

   (b) What is the formal language corresponding to the decision problem?
   Solution: $\{(G, k) : G$ has a spanning tree where each vertex has degree at most $k\}$.

   (c) Show that the decision problem is in NP.
   Solution: Guess a spanning tree (that is, guess the edges in the spanning tree). Then check that (a) there are no cycles in the edges you've chosen (i.e., it really is a tree); (b) all the nodes are connected, and (c) each vertex has degree k. This can all be checked in polynomial time. [Although you didn't have to do it, here's how: start at any node, and do a breadth-first search. Keep track of all the nodes you've seen. If the successor of a node is a node you've already seen, there's a cycle. Make sure that no node is left out of the search – otherwise the graph isn't connected. Finally, make sure that no node has more than k successors. Note that you did have to check that the edges you've chosen form a spanning tree. You can't just assume that they do.]

   (d) Show that it is NP-hard. (Hint: try reducing the Hamiltonian cycle problem to it. It's easy!)
   Solution: Notice that a graph has a Hamiltonian path iff it has a spanning where each node has degree ≤ 2. Clearly a Hamiltonian path is a spanning tree where each node has degree ≤ 2. Conversely, suppose you have a spanning tree where each node has degree ≤ 2. It must be a Hamiltonian path. If there were any branching, there would be some node with two successors and a parent, and it would have degree 3. Thus, G is a graph with a Hamiltonian path iff $(G, 2)$ is an element of the language described in (b).

9. [12 points] We need a data type with the operations Insert, GetMax (report the maximum element and delete it from the data structure), and ReportMin (report the minimum element without deleting it from the data structure). For each set of requirements listed below, describe a data structure for the ADT that fits the requirements and explain how the operations are implemented in that data structure. (Note: n is the number of elements that have been inserted.)

(a) All operations take worst-case time $O(\lg n)$.

Solution: Use two heaps, one with the the maximum element at the top, and one with the minimum elemenet at the top. (Both heaps store all the elements in the data type.) Inserting an element means inserting it into each heap. That takes $O(\lg n)$, since that's what insertion takes in each heap. Finding the max can be done in constant time (since it is the root of the first heap); then deleting the maximum element can be done in $O(\lg n)$ time for each heap. ReportMin can actually be done in constant time (since we just look at the minimum element in the second heap). Alternatively, you can use one heap with the maximum element on top, and keep a pointer to the minimum element. Every time you insert a new element, you need to update the pointer. One point was deducted if you used a BST or a skip list instead of a heap, since these data structures only have *expected* worst-case running time of $O(\lg n)$.

(b) GetMax and ReportMin each take worst-case time $O(1)$. (Note that there is no requirement on Insert.)

Solution: Use a sorted doubly-linked list. GetMax is easy (find the element at the right-hand end and delete it). ReportMin is equally easy (find the element at the left-hand end). Insertion takes time $O(n)$, since we have to go along the whole list to find the right place to put it the new element.

(c) Insert and ReportMin each take worst-case time $O(1)$. (Note that there is no requirement on GetMax.)

Solution: Use a singly-linked list with the minimum element at the head. To insert an element, just compare it to the left-most element. If it is less than the left-most element, then put it at the left end. Otherwise, put it just past the leftmost element. This clearly takes constant time. Finding the miminum also clearly takes constant time (just look at the leftmost element). GetMax take time $O(n)$ (since you have to run along the whole list to find the maximum element).