University of Washington
Department of Computer Science and Engineering
CSE 421, Fall 2006

December 15, 2006

Final Exam Solution, December, 2006

NAME: ______________________

Instructions:

- Closed book, closed notes, no calculators
- Time limit: One hour 50 minutes
- Answer the problems on the exam paper.
- If you need extra space use the back of a page
- Problems are not of equal difficulty, if you get stuck on a problem, move on.

1	/30
2	/30
3	/30
4	/30
5	/20
6	/20
7	/20
8	/20
9	/20
Total	/220

Problem 1 (30 points) Recurrences:

Give solutions to the following recurrences. Justify your answers.

a)

$$T(n) = \begin{cases} 5T(n/5) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

$O(n \log n)$. The work per level is n, and there are $\log_5 n$ levels.

b)

$$T(n) = \begin{cases} 2T(n/5) + n^{1/2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

$O(n^{1/2})$. The work per level is decreasing, so this is dominated by the work at the top level. The growth factor of work per level is $\frac{2}{\sqrt{5}} < 1$.

c)

$$T(n) = \begin{cases} 6T(n/5) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

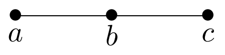
$O(n^{\log_5 6})$. The work per level is increasing, so the recurrence is dominated by the work at the leaves. The branching factor is 6, and the depth is $\log_5 n$, so the number of leaves is:

$$6^{\log_5 n} = (5^{\log_5 6})^{\log_5 n} = n^{\log_5 6}.$$

Problem 2 (30 points)Maximal Independent Set:

Let $G = (V, E)$ be an undirected graph. A subset I of the vertices is said to be independent if for all $u, v \in I$, $(u, v) \notin E$. A set of vertices M is a *maximal* independent set if M is not contained in any larger independent set, i.e., if $M \subset M'$ then M' is not independent. A set of vertices M is a *maximum* independent set if it is a largest independent set in the graph, i.e., M is independent, and if M' is any other independent set of G, $|M'| \le |M|$. In other words, a maximal independent set is an independent set that we cannot add any more vertices to without it ceasing to be independent, a maximum independent set is an independent set that contains as many vertices as possible.

a) Show that a maximal independent set is not necessarily a maximum independent set.



$\{b\}$ is a maximal independent set, while $\{a, b\}$ is a maximum independent set.

b) Show that a maximum independent set is a maximal independent set.

Let I be a maximum independent set. I could not be strictly contained in another independent set I', since I has maximum size for an independent set. Hence, I must also be maximal.

c) Give a polynomial time algorithm that finds a maximal independent set in a graph.

The idea is to use a greedy algorithm which builds an independent set one element at a time by adding an element to the set and discarding all of the neighboring elements.

```
*MaximalIndependentSet*($G = (V, E)$)
  $I := \emptyset$;
  $X := V$;
  **while** $|X| > 0$
    **choose** $v$ **from** $X$
    $I := I \cup \{v\}$;
    $X := X - \{v\}$;
    **for each** $u$ **in** $N(v)$
      $X := X - \{u\}$;
```

Problem 3 (30 points) Short Answer:

a) Compute the Fourier Transform of $1 + 2x + x^2$ at four points.

The n point Fourier Transform is to evaluate the function at the n-th roots of unity. For four points, this is evaluating at $\{1, -1, i, -i\}$, giving $\{4, 0, 2i, -2i\}$.

b) Why does Dijkstra's algorithm not work for graphs with negative cost edges.

Dijkstra's algorithm can fail with negative cost edges because encountering a negative edge can reduce the cost of a vertex that has already been processed.

c) What is Cook's theorem?

Broadly stated, Cook's theorem is that there is an NP-Complete problem. The specific version stated in class was that the Circuit Satisfiability problem is NP-Complete.

d) If you know problem X is NP-complete, and you want to show that problem Y is NP-complete, do you reduce X to Y, or reduce Y to X. Justify your answer.

Reduce X to Y. You want to show that you can use Y to solve X.

e) Name two minimum spanning tree algorithms. Which one could be called "Dijkstra's algorithm for minimum spanning trees".

Kruskal and Prim. Prim's algoirhm has the same structure as Dijkstra's algorithm, so it could be called "Dijkstra's algorithm for minimum spanning trees".

f) If you have n points in the plane, what is the run time of the fastest known algorithm for finding the closest pair of points.

$O(n \log n)$ by Divide and Conquer.

Problem 4 (30 points) Longest Path Problem:

For each of the following problems sketch an efficient algorithm or give an NP-completeness proof. You may draw on results from class or the textbook. (Note: In this problem, path length is the number of edges.)

a) Longest path in a directed acyclic graph. Given vertices s and t, find a maximum length path from s to t.

   Compute a topological sort of the vertices. Then assign to each vertex a distance from s that is one plus the maximum distance of all of its predecessors to s.

   An alternate (but less efficient approach), is to assign all edges a distance of -1, and then use the Bellman-Ford algorithm to compute shortest paths.

b) Path of length K. Given a directed graph, vertices s and t, and an integer K, find a path of length exactly K between s and t.

   This can be done with a dynamic programming algorithm, which records that there is a path of length j from s to v if there is a path of length $j-1$ from s to u and an edge from u to v.

   K phases of the Bellman-Ford algorithm (with edges assigned a length of one) does not work for this, since a single phase of Bellman-Ford can extend a path by more than one edge.

c) Longest simple path. Given a directed graph and vertices s and t, find the longest simple path between s and t. (Recall that a simple path is a path with no repeated vertices.)

   This is NP-Complete. The reduction is from the Hamiltonian Path problem (covered on the last day of class).

Problem 5 (20 points) Non-adjacent LCS:

The sequence $C = c_1, \ldots, c_k$ is a *non-adjacent subsequence* of $A = a_1, \ldots, a_n$, if C can be formed by selecting non-adjacent elements of A, i.e., if $c_1 = a_{r_1}, c_2 = a_{r_2}, \ldots, c_k = a_{r_k}$, where $r_j < r_{j+1} - 1$. The non-adjacent LCS problem is given sequences A and B, find a maximum length sequence C which is a non-adjacent subsequence of both A and B.

This problem can be solved with dynamic programming. Give a recurrence that is the basis for a dynamic programming algorithm. You should also give the appropriate base cases, and explain why your recurrence is correct.

The recurrence for $Opt(i,j)$ is giving the length of the longest common non-adjacent subsequence of $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$. The key idea is that we match a_i and b_j, then we cannot match a_{i-1} or b_{j-1}. This alters the recurrence from the LCS problem by making the dependency on $Opt(i-2, j-2)$ if there is a match. The recurrence becomes:

$$Opt(i,j) = \begin{cases} \max(Opt(i-2, j-2) + 1,\ Opt(i-1, j),\ Opt(i, j-1)) & \text{if } a_i = b_j \\ \max(Opt(i-1, j),\ Opt(i, j-1)) & \text{if } a_i \neq b_j \end{cases}$$

To handle the base cases appropriately, we need to ensure that we the don't access indices that are out of bounds. The most convenient way to do this is just to define $Opt(0,j) = Opt(i,0) = Opt(-1,j) = Opt(i,-1) = 0$ so we can avoid special cases in the recurrence.

Problem 6 (20 points) Mod K Subset Sum:

The mod K subset sum problem is: given a set of integers $S = \{s_1, \dots, s_n\}$ and an integer K, does there exist a non-empty subset S' of S such that $\sum_{s \in S'} s = W$ with $W \bmod K = 0$.
Give a dynamic programming algorithm for solving the mod K subset sum problem. Your algorithm should return the set S' if it exists.

The basic idea is that we use an array $S[0 \dots K-1]$ to keep track of the mod K sums. (Using an array of size K can be a considerable savings over solving the full subset sum, and then testing each of the set sizes divisible by K). The value $S[j]$ is used to tell us if there is a subset that has size (mod K) of j. To allow us to reconstruct the set we store the index of an element (instead of just using a boolean). The code uses an auxilliary array to avoid a bug where multiple copies of the same element are used in the sum.

The code for filling in the array is:

```
for $i := 1$ to $n$
  for $j := 0$ to $K - 1$
    if $S[j] = 0$ and $S[(j - s_i) \bmod K] \neq 0$
      $T[j] := i$;
    else
      $T[j] := S[j]$;
  $S = T$;
```

When the array is filled in, the algorithm tests is $S[0]$ is non-zero. If it is, a set can be reconstructed by tracing back through the array, picking out elements until the cell 0 is reached for a second time.

Problem 7 (20 points) Capacity Reduction for NetFlow:

Let $G = (V, E)$ be a flow graph with maximum flow f. Let e be an edge with capacity c. Suppose that the edge has it's capacity reduced to $c-1$. Describe an $O(n+m)$ time algorithm that computes a new maximum flow f', starting from the flow f for the modified flow graph. Justify the correctness of your algorithm.

Let e be the edge that has its capacity reduced from c to $c - 1$.
If $f(e) \leq c - 1$ we don't need to do anything.
If $f(e) = c$ we need to reduce the flow by one unit along e. We do this by finding an $s - t$ path through e comprising of edges with positive flow. This can be done in $O(n+m)$ time using breadth first search. We reduce the flow along this path by one unit and have a valid flow for the graph.
The resulting flow might, or might not be optimal. We look at the residual graph. If there is an $s-t$ path with augment along this path, and have an maximum flow. If the residual graph did not have an $s - t$ path, we had a maximum flow. Note that at most one augmentation is necessary, since we are within one of a maximum flow value.

Problem 8 (20 points) Vertex cut:

Let $G = (V, E)$ be a directed graph with distinguished vertices s and t. Describe an algorithm to compute a minimum sized set of vertices to remove to separate s and t. Your algorithm should identify the actual vertices to remove (and not just determine the minimum number of vertices that could be removed).

We build a flow graph and then use the minimum cut in the graph to determine the set of vertices to remove. We then split each vertex v, other than s or t into vertices v_{in} and v_{out} with a unit capacity edge from v_{in} to v_{out}. The vertex s is replaced by s_{out} and t is replaced by t_{in}. The edge (u, v) is replaced by an edge (u_{out}, v_{in}) with infinite capacity.

We compute a maximum flow between s_{out} and t_{in}. Let S be the set of vertices reachable from s_{out} in the residual graph by paths of positive capacity. We say that a vertex v is a cut vertex if v_{in} is in S, but v_{out} is not in S. The cut vertices are a minimum set of vertices to remove to separate the graph.

Problem 9 (20 points) Currency Conversion:

A group of traders are leaving India, and need to convert their Rupees into various international currencies. There are n traders and m currencies. Trader i has T_i Rupees to convert. The bank has B_j Rupees worth of currency j. Trader i is willing to trade as much C_{ij} of his Rupees for currency j. (For example, a trader with 1000 rupees might be willing to convert up to 700 of his Rupees for USD, up to 500 of his Rupees for Japaneses Yen, and up to 500 of his Rupees for Euros).

Assuming that all traders give their requests to the bank at the same time, describe an algorithm that the bank can use to satisfy the requests (if it can).

We express this problem as a network flow problem. The idea for the network flow problem is that we construct a flow graph, where the fluid is Rupees and the allocation of flow shows how this is transfered into different currencies.

The graph has vertices $x_1, \ldots, x_n$ for the traders and $y_1, \ldots, y_m$ for the currencies. The source s is connected to the vertices $x_1, \ldots, x_n$, and the vertices $y_1, \ldots, y_m$ are connected to the sink t. There are edges from the vertices $x_1, \ldots, x_n$ to the vertices $y_1, \ldots, y_m$.

An edge (s, x_i) has capacity T_i, the edge (x_i, y_j) has capacity C_{ij} and the edge (y_j, t) has capacity B_j. A flow corresponds to converting money into the different currencies. The capacity constraints ensure that the traders and banks don't exceed their amounts of currency, and that the traders get appropriate currencies. If there is a flow of $\sum_{i=1}^{n} T_i$ then all traders exchange all their money. The maximum flow clearly can't exceed this, so we compute a maximum flow in this graph.