Fall 2009 - 91.503 - Algorithms

Computer Science Department
University of Massachusetts Lowell
Lowell, MA 01854


Name: ________________________________________________________________

1._______  2._______  3._______  4._______  5._______  6._______  7._______

Total: __________/50

Exam Time: 1h & 15m. Each problem is worth 10 points. 50 pts total. Choose any 5 problems. If needed, use the back of each page.

1. **Dynamic Programming.** Recall the *rod-cutting problem*: given a rod of length $n$ inches and a table of prices $p_i$ for $i = 1, 2, \ldots, n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling the pieces.

   (5 pts) provide a pseudo-code algorithm for a dynamic programming (polynomial-time) solution of the problem.

   (5 pts) Modify your pseudo-code algorithm so that you maximize the revenue in the case where each cut has a fixed positive cost $c$. Explain what this does *in your own words*.

   **Hint.** The best solution for the first part is the Bottom-Up-Cut-Rod algorithm on p. 366 of the third edition. For the second part, one can modify this algorithm so that each time a cut appears, $\max(q, p[i] + r[j - i])$ is diminished by $c$, with no such diminishing if $i = j$, since no cut is required.

2. **Greedy Algorithms.**

   (3 pts) Describe the *fractional knapsack problem*.

   (3 pts) Define *greedy choice property*.

   (4 pts) Prove that the fractional knapsack problem has the greedy choice property.

   **Hints.** The *Fractional Knapsack Problem* is defined on p. 426 of the third edition. The *Greedy Choice Property* is defined on p. 424. It would have been helpful to say what it meant in the case of the Fractional Knapsack Problem, since you have to use it in the last part.
Sketch of proof. Assume the knapsack has volume \( V > 0 \). Assume the items \( i = 1, \ldots, n \) are ordered by decreasing value per unit volume, with \( w_i \) denoting the quantity of item \( i \). Denote the value per unit volume by \( \theta_i \). The greedy strategy is then: *take as much of item 1 as you can. If that does not fill the knapsack, take as much of item 2 as you can. Continue this way until you either have filled the knapsack or run out of items to put in it.* Since the decision is made at each point, based only on the current available choices, this is a greedy strategy. Does it apply? Assume not. There are two possibilities (the detail may be boring...)

1. The knapsack was filled on the first try, all with \( \theta_1 \) material. Any other strategy would involve a volume \( v_1 < V \) of material 1, and a volume \( V - v_1 > 0 \) filled by materials of type \( j > 1 \). An upper bound on the value in this latter case is \( v_1 \cdot \theta_1 + (V - v_1) \cdot \theta_2 < V \cdot \theta_1 \), contradicting optimality.

2. The knapsack required two or more materials to be filled, so the greedy strategy has \( w_1 \cdot \theta_1 + v_2 \cdot \theta_2 + \ldots \). Assume this strategy does not lead to an optimal solution. We can also assume it breaks down at the first choice (otherwise start at the first \( i \) for which it breaks down). Let \( v_i = w_1, v_j, j = 2, \ldots, n \), denote the volumes in the greedy strategy, let \( \bar{v}_j, j = 1, \ldots, n \) denote the volumes in the optimal strategy. So the optimal strategy has \( \bar{v}_1 < w_1 < V \). In the optimal strategy, replace as much of \( \bar{v}_j \) (where \( j \) is the smallest index item greater than 1 appearing in the optimal strategy) as possible by the remaining \( w_1 - \bar{v}_1 \) of item 1. We have just increased the value of the load, thus violating the hypothesis that we had an optimal solution.

3. Amortized Analysis. A sequence of \( n \) operations is performed on a data structure. The \( i^{th} \) operation costs \( i \) if \( i - 1 \) is an exact power of 2, and 1 otherwise.

(4 pts) Describe the *accounting method* of analysis.

(4 pts) Use the accounting method of analysis to determine the amortized cost per operation in the data structure given above.

(2 pts) What data structure does this apply to? Explain briefly.

Hints. For the first part, see p. 456. For the second part, see the discussion on p. 465. The result of the summation near the bottom, tells you that assigning 3 to each insertion, taking 1 to pay for the insertion and leaving 2 to take care of table expansion is the correct step. The data structure is the dynamic table - which does what and how?

4. Sets.

(3 pts) Define *rank* and *union by rank*.

(7 pts) Give a sequence of \( m \) Make-Set, Union and Find-Set operations, \( n \) of which are Make-set operations, that takes \( \Omega(m \log n) \) time when we use union by rank only. Explain your reasoning.

Hints. For union by rank, see p. 569.

1. There are \( n \) Make-Sets for a number of operations = \( n \).

2. If we perform the unions as \( \text{Union}(x_1, x_2), \ldots, \text{Union}(x_{n-1}, x_n) \) (assume \( n \) a power of two, but it is not crucial), followed by
Union\((x_1, x_2, x_3, x_4), \ldots, \text{Union}(x_{n-3}, x_{n-2}, x_{n-1}, x_n)\), and so on, until we have a single set, the number of operations is \(n - 1\), each with constant cost. Note that the union by rank has given us a tree of rank (and thus depth) \(\log n\). Why?

3. Take \(m > 3n\). Each of the \(m - 2n + 1 > m/3\) **Find-Set** operations can take up to \(\log n\) time. Thus \(\Omega(m \log n)\).

5. **Graphs, Flows and Cuts.**

   (2 pts) Define bipartite graph.

   (2 pts) Define bipartite matching.

   (2 pts) Describe the Ford-Fulkerson algorithm. What does it compute?

   (6 pts) Run the Ford-Fulkerson Algorithm on the flow network below to obtain a maximum bipartite matching. Show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5, and in R top to bottom from 6 to 9. For each augmentation pick the augmenting path that is lexicographically smaller (e.g., \(s_{16}t \leq s_{28}t\)).

   **Hints.** For bipartite graph and bipartite matching see p. 732. For Ford-Fulkerson, see p. 724. The construction of the solution goes as follows

   Find an augmenting path (joining \(s\) and \(t\)). There are several possible, but the only one satisfying the condition is the path \(s_{16}t\). Since we assume all capacities to be 1, the residual network just reverses the edges \((s, 1), (1, 6), (6, t)\). We now construct a second augmenting path. The lexicographic constraint forces us to choose \(s_{28}t\). Again, reverse the edges to construct the residual graph. On the third pass we have the augmenting path \(s_{37}t\). Constructing the residual graph (reverse the last set of edges) gives us a graph with no paths from \(s\) to \(t\). We are done. The Maximal Match: \((1, 6), (2, 8)(3, 7)\).

6. **Graphs, Flows and Cuts.** Let \(G = (V, E)\) be a flow network with source \(s\) and sink \(t\). Let \(f\) be a flow on \(G\).

   (2 pts) Define flow. Use either definition (from either edition): the rest of the solution must remain consistent with your definition.
(2 pts) Define cut.

(2 pts) Define $|f|$, the value of the flow.

(4 pts) Sketch the proof that the net flow across $(S, T)$ is $f(S, T) = |f|$.

**Hints.** See textbook. The crucial pieces of the fourth part are given by 26.11 on p. 722 and the first equation containing $f$ (why?). The second edition has a shorter proof.

7. **Graphs, Flows and Cuts.** Use Johnson’s algorithm to find the shortest paths between all pairs of vertices in the graph below.

(6 pts) Show the value of $h$ and $\hat{w}$ computed by the algorithm - this needs the Bellman-Ford algorithm. Show the intermediate steps.

(4 pts) Complete the computation - just for the path from 6 to 5, with full application of Dijkstra’s algorithm started from 6. Show the intermediate steps.

Add a distinguish vertex, and name it 0. Connect it with all the other vertices via edges of weight 0. You will get:
You now need to use Bellman-Ford to find the shortest distances from 0 to every other vertex. See the text for pseudo-code for the algorithm. The main part, for us is:

Bellman-Ford(G, w, s)
   Initialize-Single-Source(G, s)
   for i = 1 to |G.V| - 1
      for each edge (u, v) ∈ G.E
         Relax(u, v, w)

Where

Relax(u, v, w)
   if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u

Take the edges in the following order (you may end up with different intermediate values depending on the order of the edges): (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (1, 5), (2, 1), (2, 4), (3, 2), (3, 6), (4, 1), (4, 5), (5, 2), (6, 2), (6, 3).

We have several passes from the initial state (the second line gives the results after all the (0, v) edges have been relaxed):

<table>
<thead>
<tr>
<th>0.d = 0</th>
<th>1.d = ∞</th>
<th>2.d = ∞</th>
<th>3.d = ∞</th>
<th>4.d = ∞</th>
<th>5.d = ∞</th>
<th>6.d = ∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.d = 0</td>
<td>1.d = 0</td>
<td>2.d = 0</td>
<td>3.d = 0</td>
<td>4.d = 0</td>
<td>5.d = 0</td>
<td>6.d = 0</td>
</tr>
<tr>
<td>(1,5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2,1)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2,4)</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3,2)</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3,6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

With six more passes to go. For a small graph, a visual inspection is feasible... The computation will lead to the graph and distances:
We now construct the function \( \hat{w} \) on the edges of the original graph:

\[
\begin{align*}
\hat{w}(1, 5) & = w(1, 5) + h(1) - h(5) = -1 + (-2) - (-3) = 0 \\
\hat{w}(2, 1) & = w(2, 1) + h(2) - h(1) = 1 + (-3) - (-2) = 0 \\
\hat{w}(2, 4) & = w(2, 4) + h(2) - h(4) = 2 + (-3) - (-1) = 0 \\
\hat{w}(3, 2) & = w(3, 2) + h(3) - h(2) = 2 + 0 - (-3) = 5 \\
\hat{w}(3, 6) & = w(3, 6) + h(3) - h(6) = -8 + 0 - (-8) = 0 \\
\hat{w}(4, 1) & = w(4, 1) + h(4) - h(1) = -1 + (-1) - (-2) = 0 \\
\hat{w}(4, 5) & = w(4, 5) + h(4) - h(5) = 3 + (-1) - (-3) = 5 \\
\hat{w}(5, 2) & = w(5, 2) + h(5) - h(2) = 7 + (-3) - (-3) = 7 \\
\hat{w}(6, 2) & = w(6, 2) + h(6) - h(2) = 5 + (-8) - (-3) = 0 \\
\hat{w}(6, 3) & = w(6, 3) + h(6) - h(3) = 10 + (-8) - (0) = 2
\end{align*}
\]

The graph now becomes;

And you can apply Dijkstra’s Algorithm. Depending on how you choose the exit edges from each vertex (e.g., clockwise from “straight down” or counterclockwise), you will compute different shortest paths.