College of Computer & Information Science
Northeastern University
CSU690: Algorithms and Data

Spring 2005
Handout 14
29 March 2005

Sample Solution to Problem Set 5

1. Weighted activity selection

Consider a weighted version of the activity selection problem, in which each activity has a *weight*, in addition to the start and finish times. (For example, the weight may signify the importance of the activity.) The goal is to select a maximum-weight set of mutually compatible activities, where the weight of a set of activities is the sum of the weights of the activities in the set.

(a) (5 points) Consider a greedy algorithm that repeatedly performs the following step until no more activities can be selected: select an activity that has the maximum ratio of weight over length among all activities that do not overlap with the activities selected thus far.

Give a counterexample to show that the above greedy algorithm will not yield an optimal solution for the weighed activity selection problem.

Answer: Consider the example with just two overlapping activities: one with weight 2 and length 3, and another with weight 1 and length 1. The above greedy algorithm will pick the activity with weight 1, while optimal is to pick the activity with weight 2.

(b) (15 points) Use dynamic programming to solve the weighted activity selection problem. Analyze the running time of your algorithm.

Answer: We first sort all the activities according to their finish times. If two activities have the same finish times, we break ties arbitrarily. Let the sorted set of activites be S and the activities in S be numbered 1 through n in the sorted order. Thus, if $i > j$, we have $f_i \geq f_j$.

For any i in 1 through n, we now give a recurrence relation for $S[i]$, a maximum-weight set of mutually compatible activities with i as the last activity to finish.

$$S[i] = \max\left\{\{i\}, \max\{S[j] \cup \{i\} : j < i \text{ and } i \text{ is compatible with } j\}\right\},$$

where the maximum is taken over the total weight of the sets.

We prove the correctness of the above recurrence relation by induction on i. For i equal to 1, the proof is trivial. Consider $i > 1$. Let S^* be any set of mutually compatible activities with i as the last activity to finish. Let j be the activity with the largest finish time in $S^* \setminus \{i\}$. (If no such j exists, then $S^* = \{i\}$ and has weight at most the weight of $S[i]$ since $S[i]$ includes i.) Since i is the last activity to finish in S^*, we have $j < i$. Therefore, by the induction hypothesis, the weight of $S^* \setminus \{i\}$ is at most the weight of $S[j]$. Since j is compatible with i, the recurrence relation guarantees that the weight of $S[i]$ is at least the weight of $S[j]$ plus the weight of i, which is at least the weight of S^*. Therefore, $S[i]$ is a maximum-weight set of mutually compatible activities with i as the last activity to finish.

So the algorithm computes $S[i]$ for all i going from 1 to n. We then return the maximum-weight set among these n sets. The running time of the algorithm is $O(n^2)$ since the calculation of $S[i]$ takes $O(i)$ time (i compatibility checks and then finding the maximum over a set of i elements).

2. (15 points) Edit distance

Part (a) of Problem 15–3, pages 364–366.

Answer: Given strings $x[1..m]$ and $y[1..n]$, we define two matrices $C[0..m, 0..n]$ and $O[0..m, 0..n]$. We first consider all the operations but the *kill* operation. The entry $C[i, j]$ is the optimal cost of transforming $x[1..i]$ to $y[1..j]$, while $O[i, j]$ is the last operation performed in an optimal transformation of $x[1..i]$ to $y[1..j]$. (If i equals 0, then $x[1..i]$ is the empty string.) We can now compute $C[i, j]$ and $O[i, j]$ using the following procedure, which we refer to as COMPUTE(i, j).

1. If $i = 0$ and $i = 0$, then set $C[i, j]$ to 0 and return $C[i, j]$.
2. Initialize $C[i, j]$ to ∞.
3. *copy*: If $i > 0$ and $j > 0$ then if $x[i] = y[j]$ and $C[i-1, j-1] + \text{cost}(\textit{copy}) < C[i, j]$, then set $C[i, j]$ to $C[i-1, j-1] + \text{cost}(\textit{copy})$ and $O[i, j]$ to *copy*.
4. *insert*: If $j > 0$ then if $C[i, j-1] + \text{cost}(\textit{insert}) < C[i, j]$, then set $C[i, j]$ to $C[i, j-1] + \text{cost}(\textit{insert})$ and $O[i, j]$ to *insert*.
5. *delete*: If $i > 0$, then if $C[i-1, j] + \text{cost}(\textit{delete}) < C[i, j]$, then set $C[i, j]$ to $C[i-1, j] + \text{cost}(\textit{delete})$ and $O[i, j]$ to *delete*.
6. *replace*: If $i > 0$ and $j > 0$, then if $C[i-1, j-1] + \text{cost}(\textit{replace}) < C[i, j]$, then set $C[i, j]$ to $C[i-1, j-1] + \text{cost}(\textit{replace})$ and $O[i, j]$ to *replace*.
7. *twiddle*: If $i > 1$ and $j > 1$, then if $x[i] = y[j-1]$, $x[i-1] = y[j]$, and $C[i-1, j-1] + \text{cost}(\textit{copy}) < C[i, j]$, then set $C[i, j]$ to $C[i-2, j-2] + \text{cost}(\textit{twiddle})$ and $O[i, j]$ to *twiddle*.
8. Return $C[i, j]$.

We now prove the correctness of our computation above. The proof is by induction on the ordered pair (i, j). For the induction basis, we note that $C[0, 0]$ is indeed zero. For the induction step, we first make the hypothesis that we have correctly computed $C[i', j']$ for all (i', j') such that either $i' \leq i - 1$ or $j' \leq j - 1$. Consider the computation of $C[i, j]$. Let C^* be the optimum cost for transforming $x[1..i]$ to $y[1..j]$ without using the *kill* operation, and let *op* be the last operation in this optimal transformation. We prove that $C[i, j]$ equals C^* by considering different cases depending on the operation *op*. Here we only prove for the cases *op*= *copy* and *op*= *twiddle*. Similar proofs can be given for all other operations save *kill*.

If *op* is *copy*, then the optimal solution would have first transformed $x[1..i-1]$ to $y[1..j-1]$ and then performed the *copy* operation. Therefore, we have $x[i] = y[j]$ and $C^* \geq C[i-1, j-1] + \text{cost}(\textit{copy})$ since the induction hypothesis says that $C[i-1, j-1]$ is the optimal cost of transforming $x[1..i-1]$ to $y[1..j-1]$. In the computation of $C[i, j]$, step 3 guarantees that $C[i, j] \leq C[i-1, j-1] + \text{cost}(\textit{copy}) \leq C^*$. Thus, $C[i, j]$ is indeed the cost of an optimal transformation from $x[1..i]$ to $y[1..j]$.

If *op* is *twiddle*, then the optimal solution would have first transformed $x[1..i-2]$ to $y[1..j-2]$ and then performed the *twiddle* operation. Therefore, we have $x[i] = y[j-1]$, $x[i-1] = y[j]$ and $C^* \geq C[i-2, j-2] + \text{cost}(\textit{twiddle})$ since the induction hypothesis says that $C[i-2, j-2]$ is the optimal cost of transforming $x[1..i-2]$ to $y[1..j-2]$. In the computation of $C[i, j]$, step 7 guarantees

that $C[i,j] \leq C[i-2,j-2] + \text{cost}(\textit{twiddle}) \leq C^*$. Thus, $C[i,j]$ is indeed the cost of an optimal transformation from $x[1..i]$ to $y[1..j]$.

So our algorithm for computing the optimal costs for transformations not using the *kill* operation is the following:

1. **for** $i \leftarrow 0$ to m
2. **for** $j \leftarrow 0$ to n
3. COMPUTE(i,j)

After running the above algorithm, we consider the *kill* operation by running the following algorithm:

- for i from 1 to n, if $C[i,n] + \text{cost}(\textit{kill}) < C[m,n]$, then set $C[m,n]$ to $C[i,n] + \text{cost}(\textit{kill})$ and $O[m,n]$ to "*kill* from i onwards".

To prove the optimality of $C[m,n]$, we note that an optimal transformation of $x[1..m]$ to $y[1..n]$ that ends with a "*kill* from i onwards" consists of an optimal transformation of $x[1..i]$ to $y[1..n]$ followed by the "*kill* from i onwards" operation. Since we consider all possible values of i in the above algorithm and pick the best one, the optimality of $C[m,n]$ holds.

In order to print the optimal transformation, we retrace the computation using the operation matrix O that we have computed. We run PRINT(m,n) using the following recursive algorithm:

PRINT(i, j)
1. **if** $i = 0$ and $j = 0$ then **return**
2. **if** $O[i,j]$ is *copy* or *replace* then PRINT$(i-1, j-1)$
3. **if** $O[i,j]$ is *insert* then PRINT$(i, j-1)$
4. **if** $O[i,j]$ is *delete* then PRINT$(i-1, j)$
5. **if** $O[i,j]$ is *twiddle* then PRINT$(i-2, j-2)$
5. **if** $O[i,j]$ is "*kill* at k" then PRINT(k, j)
6. **print** $O[i,j]$

3. (10 points) Depth-first search

Exercise 22.3-2, page 547.

Answer: See figure at the end of the document.

4. (10 points) Connected components

Exercise 22.3-11, page 549.

Answer: In depth-first search, we start a new component whenever we call DFS-VISIT from within DFS. Refer to the pseudocode on page 541 of the textbook. So we can maintain a component counter c in the DFS subroutine, initialized to 0. Just before calling DFS-VISIT, we increment c and pass it to DFS-VISIT (by calling DFS-VISIT(u,c) at line 7). In DFS-VISIT(u,c), whenever we color a node u GRAY, we will set $cc[u]$ to be c.

The value of c at termination of DFS(G) yields the number of connected components in G.

5. (10 points) Topological sort

Exercise 22.4-5, page 552.

Answer: For each vertex v, we maintain a field in(v) that represents the current in-degree of the vertex. We also maintain a set Z of vertices that have in-degree 0 and have not been removed from the graph.

The initialization phase is the following. We set in(v) to be the in-degree of v in the original graph G. The set Z is a linked list of vertices in V that have zero in-degree. All of these steps can be done in $\Theta(V + E)$ time.

In the iterative routine, we repeatedly remove a vertex u from Z. For every edge (u, v) coming out of u, we decrement in(v) by 1. If in this process, any in(v) becomes zero, we add the vertex to Z. We repeat this process until Z is empty, which is when the procedure terminates.

If G has cycles, then Z may become empty, yet all of the vertices will not be output. This is because we may reach a state in which there are vertices in the graph and none of them have a zero in-degree. In this situation, the algorithm will terminate without outputting all of the vertices. An example of this case is when G is a directed ring.

6. (10 points) Minimum spanning tree

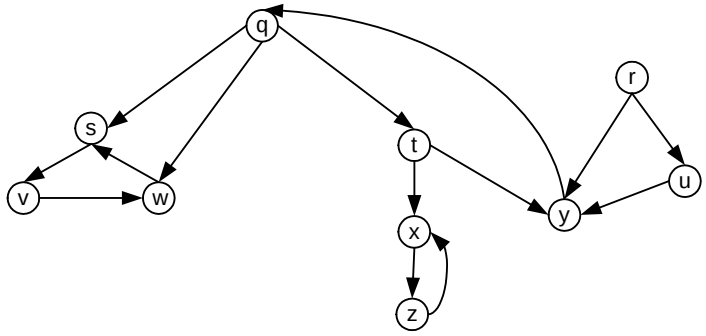
Let G be an undirected weighted graph. Show the weight of the maximum-weight edge of any MST of G is minimum among all spanning trees of G.

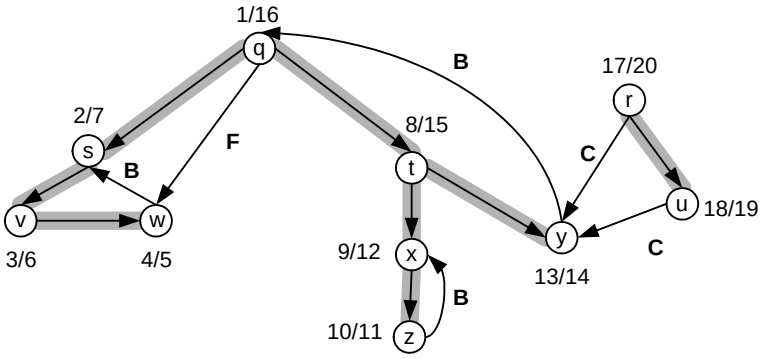
(*Hint:* Consider two trees T_1 and T_2, where T_1 is an MST. If the maximum-weight edge of T_1 has higher weight than the maximum-weight edge of T_2, then argue that you can replace this edge of T_1 with another edge of T_2, thus yielding a contradiction.)

Answer: Consider two trees T_1 and T_2, where T_1 is an MST. Let e be a maximum-weight edge in T_1. We now show that there is at least one edge in T_2 that has weight at least $w(e)$. It then follows that the largest edge weight of T_1 is at most the largest edge weight among all spanning trees.

The edge e partitions T_1 into two subsets of vertices, say S and $V - S$. Since T_2 is a spanning tree of G, there is an edge e' in T_2 that crosses the cut $(S, V - S)$. We now show that $w(e) \leq w(e')$. The proof is by contradiction. If $w(e) > w(e')$, then $T_3 = T_1 - e + e'$ is a spanning of lesser weight than T_1, contradicting the fact that T_1 is an MST. Thus $w(e) \leq w(e')$, hence completing the proof.

22.3-2


q
r
s
t
u
v
w
y
x
z


1/16
q
B
17/20
r
2/7
s
F
8/15
t
C
B
u
18/19
v
w
y
C
3/6
4/5
9/12
x
13/14
B
10/11
z