CS3933 Winter 2007 – Solo Assignment 3
Due Wednesday October 31, 2007, at 5pm.

Sample Solution

Note that, even though desired running times are not specified, faster algorithms are better and thus worth more marks.

1. (10 marks: 8 for the algorithm, 2 for the time analysis)

   Design and write an algorithm that, given a directed graph $D = (V, E)$ and a vertex s, will find a t (if there is one) such that there is more than one simple path in D from s to t.

   Analyse the running time of your algorithm.

   Basic idea of solution: in a DFS, keep track not only of whether a vertex has been previously seen, but whether it is still on the stack (and thus part of a path to the current vertex) or has been removed from the stack.

   Algorithm $TwoPath$(directed graph D, source vertex s): returns vertex t
   - Mark all vertices as unvisited
   - return $findt(D, s)$

   Algorithm $findt$(directed graph D, vertex u): returns vertex t
   - mark u as stacked
   - $t \leftarrow 0$
   - for all v such that $(u, v) \in E$
     - if v is unvisited
       - if $t = 0$
         - $t \leftarrow findt(D, v)$
     - else
       - if v is processed
         - $t \leftarrow v$
   - mark u as processed
   - return t

   Time Analysis:
   The loop in $TwoPath$ iterates n times (to mark all vertices as univisted). $findt$ is recursive, and will, in the worst case where the entire graph must be searched, have

one instantiation for each vertex. The for loop iterates a number of times equal to the degree of the vertex, so over all instantiations of $findt$, it will iterate $2m$ times in the worst case. The rest of the algorithm contributes a constant factor, so the algorithm as a whole runs in time $\in \Theta(m+n)$.

2. (10 marks: 8 for the algorithm, 2 for the time analysis)

   Design and write an algorithm that, given a weighted undirected graph $G = (V, E)$ (where each edge e has a weight $w(e) > 0$), a vertex s, and a vertex t, will find the length of the second shortest path from s to t (where the length of the path is the sum of its edge weights). The second shortest path need not be simple.

   Analyse the running time of your algorithm.

   Basic idea: each vertex needs to be in the priority queue twice, once for its shortest distance from s and once for its second shortest distance from s. We will need to revisit vertices, so we should not mark them off as visited.

```
Algorithm Second(weighted graph G = (V, E), s, t): returns number
    Let PQ be an empty priority queue (using a min heap)
    for all u ∈ V
        d1[u] ← ∞
        add (u, 1) to PQ with priority d1[u]
        d2[u] ← ∞
        add (u, 2) to PQ with priority d2[u]
    d1[s] = 0
    reprioritise (s, 1) in PQ with priority 0
    done ← false
    while done = false and PQ is not empty
        (u, x) ← dequeue(PQ)
        if u = t and x = 2
            done ← true
        for all v such that (u, v) ∈ E
            d ← dx[u] + w(u, v)
            if d <d1[v]
                d2[v] ← d1[v]
                reprioritise (v, 2) in PQ with priority d2[v]
                d1[v] ← d
                reprioritise (v, 1) in PQ with priority d1[v]
            else
```

if $d < d_2[v]$
$d_2[v] \leftarrow d$
reprioritise $(v, 2)$ in PQ with priority $d_2[v]$
return $d_2[t]$

Time Analysis:
All the heap operations, in the worst case, take $\Theta(\log n)$ time. There are $2n + 1$ heap operations used to initialise the heap; however, the first $2n$ operations will not require restructuring of the heap, so the total time needed to initialise PQ is $n + \log n$. The while loop iterates $2n$ times in the worst case (each vertex having to be dequeued twice). Within the while loop, the for loop iterates $deg(u)$ times. In the worst case where each vertex is dequeued twice, each vertex occurs as u twice, so the total number of for loop iterations (over all while loop iterations) is $2 \cdot \sum_{u \in V} deg(u) = 4m$. Since each iteration of the for loop contains a constant number of heap operations, and the rest of the algorithm contributes a constant factor, the algorithm as a whole runs in time $\in \Theta(n + m \log n)$.

3. (15 marks) (seen recently at a programming contest near you...)

You run a small photocopying service with a single large machine. Every morning you get a set of jobs from customers. You want to do the jobs in the order that keeps the customers happiest. Let customer i's job take time t_i to do. Given a schedule, an ordering of all the jobs, let c_i be the time at which this job is completed, which equals the sum of all the times of all the jobs that are done before plus t_i. For example if job i is done first then $c_i = t_i$, and if job i is done immediately after job j, then $c_i = c_j + t_i$.

The happiness of a customer is assumed to be proportional to the completion time of their job. You want to minimize the sum of the completion times, $c_1 + c_2 + \ldots c_n$. You need an algorithm that, when the times of the customers jobs are given, returns the best order in which to do the jobs.

(a) (4 marks) Design and write a greedy algorithm that, given the set of job times $\{t_j\}$, will find a schedule for the jobs that will minimise the sum of the completion times.

sort the set of jobs by nondescending completion time t_j, using mergesort.
$t \leftarrow 0$
for j from 1 to n, in sorted order
$s_j \leftarrow t$

$c_j \leftarrow s_j + t_j$
$t \leftarrow c_j$

(b) (1 marks) Analyse the running time of your algorithm.

The sort will take $\Theta(n \log n)$ time. The for loop will iterate exactly n times. The rest of the algorithm takes constant time, so the algorithm as a whole runs in time $\in \Theta(n \log n + n) \;=\; \Theta(n \log n)$.

(c) (6 marks) Prove that your algorithm works correctly.

Assume not. Then there is some input instance such that the greedy algorithm is not optimal.

Let $g_1, g_2, \ldots, g_n$ be the indices in the greedy order, and let $o_1, o_2, \ldots, o_n$ be the indices in an optimal order. We can assume, without loss of generality, that the optimal order considered is the one that agrees with the greedy for the longest, ie. over all optimal orders, it has the largest value of r such that $g_i = o_i \; \forall 1 \leq i < r$ and $g_r \neq o_r$.

Let k be such that $o_k = g_r$ (the index position of g_r in the optimal order). Since position r is the first point of difference, $k > r$. Since the greedy algorithm picks shorter jobs first, $t_{g_r} \leq t_{o_r}$, so $t_{o_k} \leq t_{o_r}$.

The sum of the optimal is:

$$Opt \;=\; \sum_{i=1}^{n} \sum_{j=1}^{i} t_{o_j}$$

Modify the optimal by swapping o_k with o_r, producing the sequence

$o_1, \ldots, o_{r-1}, o_k, o_{r+1}, \ldots, o_{k-1}, o_r, o_{k+1}, \ldots, o_n$

This sequence agrees with the greedy order longer than the optimal does; if it is also an optimal sequence, then we will contradict the maximality of r.

In the modified optimal, jobs o_1 through o_{r-1} and o_{k+1} through o_n have exactly the same jobs before them (even if the order has changed for the later jobs), so their completion time has not changed.

For a job j such that $r < j < k$, its completion time will have changed by $(t_{o_k} - t_{o_r})$, which is ≤ 0 since $t_{o_k} \leq t_{o_r}$.

The completion time of job r will have increased by $t_{o_k} + \sum_{j=r+1}^{k-1} t_{o_j}$. Similarly, the completion time of job k will have decreased by $t_{o_r} + \sum_{j=r+1}^{k-1} t_{o_j}$. Taken together, the sum of their completion times will have changed by $(t_{o_k} - t_{o_r})$, which is ≤ 0.

Therefore the sum of completion times of the modified optimal solution is not larger than the sum for the original optimal, so this modification preserves optimality. Since it violates the maximality of r among optimal orders, this produces a contradiction.

Therefore our original assumption is false, so the greedy algorithm is always optimal.

(d) (4 marks) Consider a different algorithm where the order in which the jobs are considered is the reverse of the order you used in (a). Give a counterexample to show that this order will not always work.

Let $n = 2$, with $t_1 = 5$ and $t_2 = 1$. The jobs are thus already in the reverse of nondescending order.
The sum of completion times is: $c_1 + c_2 \;=\; 5 + (5 + 2) \;=\; 12$,
while completing job 2 first would produce the sum $2 + (2 + 5) \;=\; 9 \;< 12$. Thus the reverse order does not work for this example.