CS3933 Winter 2007 – Solo Assignment 3
Due Wednesday March 14, 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. (a) (7 marks) Design and write an algorithm that, given an undirected graph $G = (V, E)$ and a vertex s, will find if there is a cycle that contains s.

   Analyse the running time of your algorithm.

   Basic idea: compute a DFS from s, and look for a backedge that goes back to s.

   Algorithm FindCycle(graph $G = (V, E)$, source s): returns boolean
   mark all vertices as unvisited
   mark all edges as unused
   mark s as visited
   $iscycle \leftarrow DFSCycle(G, s, s)$
   return $iscycle$

   Algorithm DFSCycle(graph $G = (V, E)$, source s, vertex u): returns boolean
   $retval \leftarrow false$
   for all v such that $(u, v) \in E$
   if v is unvisited
   mark v as visited
   mark (u, v) as used
   if $retval = false$
   $retval \leftarrow DFSCycle(G, s, v)$
   else
   if (u, v) is unused and $v = s$
   $retval \leftarrow true$
   return $retval$

   Time analysis:
   The DFS will need to initialise each vertex and each edge. The traversal will look at each edge twice (once per endpoint) in the worst case (where it needs to visit the entire graph before finding a backedge to s), and the rest of the computation steps take constant time, so the algorithm as a whole takes time $\in \Theta(m + n)$.

(b) (9 marks) Design and write an algorithm that, given a weighted graph $G = (V, E)$ (where each edge e has a weight $w(e)$) and a vertex s, will find the weight of the minimum weight cycle that contains s.

Analyse the running time of your algorithm.

Basic idea: adapt Dijkstra's Shortest Path algorithm for graphs with weighted edges, so that it keeps track of the shortest and second-shortest distance to a vertex (where the "second" shortest has a different initial edge). The first vertex to have both of its distances removed from the priority queue will form (from these paths) the minimum cost cycle including s.

```
Algorithm MinCycle(graph G = (V, E), weight function w vertex s)
    mark all edges as unused
    Let P be a priority queue (using a min heap)
    min ← ∞
    Let d1 be an array of values, one per vertex, all set to ∞
    Let d2 be an array of values, one per vertex, all set to ∞
    d1[s] ← d2[s] ← 0
    for all u such that (s, u) ∈ E
        d1[u] ← w(s, u)
        add u to P with label 1, (s, u) and priority w(s, u)
        mark (s, u) as used
    while P is not empty and min = ∞
        u ← dequeue(P)
        if u has label 2, e
          min ← d1[u] + d2[u]
        else (u has label 1, e)
          for all v such that (u, v) ∈ E and (u, v) is unused
              mark (u, v) as used
              if d1[v] = ∞
                d1[v] ← d1[u] + w(u, v)
                add v to P with label 1, e and priority d1[v]
              else
                if d1[v] > d1[u] + w(u, v)
                  Let x be the edge such that 1, x is a label of v in P
                  if x ≠ e
                    d2[v] ← d1[v]
                    add (or reprioritise) v in P with label 2, x and priority d2[v]
```

$d_1[v] \leftarrow d_1[u] + w(u, v)$
reprioritise v in P with label $1, e$ and priority $d_1[v]$
else
if $d_2[v] > d_1[u] + w(u, v)$ and $d_1[v]$ has label $1, x$ with $x \neq e$
$d_2[v] \leftarrow d_1[u] + w(u, v)$
add (or reprioritise) v in P with label $2, e$ and priority $d_2[v]$
return min

Running time:
This algorithm, in the worst case, visits all vertices once and uses all edges. The total number of iterations (of the while loop and for loop nested inside it) is $2 \cdot |E|$ in the worst case; for each of these iterations, two priority queue operations are needed. Each of the n iterations of the while loop requires an additional priority queue operation. All priority queue operations, using a min heap, will take $\Theta(\log n)$ time (since the priority queue will have at most $2n$ nodes). Thus the work done by the algorithm (except for initialization) takes time $\in \Theta(m \log n)$. The initialization takes $\Theta(n)$ time, so the algorithm as a whole takes time $\in \Theta(n + m \log n)$ time.

Alternative methodology (slower, worth 6 marks incl. time analysis):
(note that to get all of these marks, you would need to fill in the Shortest Path algorithm for weighted graph – you can use Dijkstra's algorithm).

$min = \infty$
for all u such that $(s, u) \in E$
$d = ShortestPath((V, E - \{(s, u)\}, u, s)$
if $min > d + w(s, u)$
$min = d + w(s, u)$
return min

This methodology calls the shortest path algorithm (which runs in time $\in \Theta(n + m \log n))$ $deg(s)$ times. In the worst case, $deg(s) = n - 1$, so it runs in time $\in \Theta(n^2 + nm \log n)$.

2. You are trying to earn some money by working independently. You have total time T to spend on this work, and there is a set of n projects, where a project p takes time t_p (an integer number of hours) and will pay you d_p dollars (for project p as a whole).

   This work is independent, so you can schedule your work as you want (within your available time T). Also, these projects pay by the hour, so you can earn money even from incompleted projects; if you worked some fraction of t_p on the project, you would

get that same fraction of d_p as your pay. You cannot work more than t_p hours on project p.

(a) (4 marks) Design and write a greedy algorithm that, given the time T, set of project times $\{t_p\}$, and set of project dollar amounts $\{d_p\}$. will find a schedule for you that will maxmise the total dollars that you can earn within your time limit T.

Algorithm MaxEarnings(n, $\{t_p\}$, $\{d_p\}$, T)
  sort the projects by descending (nonincreasing) rate $\frac{d_p}{t_p}$
  $sum \leftarrow 0$
  $time \leftarrow 0$
  for all projects p in sorted order
    if $time < T$
      $frac = \frac{\min\{T-time, t_p\}}{t_p}$
      print "work at job j at time $time$ for $t_p \cdot frac$ hours"
      $time \leftarrow time + t_p \cdot frac$
      $sum \leftarrow d_p \cdot frac$
  return sum

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

The sort, using mergesort, will take time $\in \Theta(n \log n)$. The for loop will have n iterations, and each iteration takes constant time. Since the rest of the algorithm takes constant time, the algorithm as a whole takes time $\in \Theta(n \log n)$.

(c) (5 marks) Prove that your algorithm works correctly.

Proof: assume not. Then there is an input for which the greedy algorithm does not produce the optimal.

For each job j, let g_j be the fraction of the job to be done in the greedy schedule, and let o_j be the fraction of the job to be done in the optimal schedule. If there is more than one optimum job fraction assignment, we will consider the one that, where the jobs are ordered in the nonincreasing $\frac{d_p}{t_p}$ greedy order, r is the maximum index possible where $g_r \neq o_r$ and $g_i = o_i \ \forall 1 \leq i < r$.

Since the greedy algorithm will take all of a job as long as there is time remaining (within the limit) to work, $g_r > o_r$, and the optimum selection spends $t_r \cdot (g_r - o_r)$ time on some other jobs that the greedy algorithm does not work all of (so that there are $t_r \cdot (g_r - o_r)$ hours that the optimum works on jobs other than what the greedy does). Since the greedy algorithm picks the jobs with the highest rate of pay, the optimum works this time to get an amount no higher than $d_r \cdot (g_r - o_r)$.

Modify the optimum to move those $t_r \cdot (g_r - o_r)$ hours to job r; this will change the optimum result to agree longer with the greedy algorithm's result while not decreasing the total amount of money earned, thus contradicting the maximality of r for the optimum.

Therefore the assumption is false, and the greedy algorithm does always produce the optimum result.

(d) (3 marks) Consider a different situation where you are only paid for a job if you complete all of it. Give, and demonstrate, a counterexample that shows that your algorithm from (a) will not work in this situation.

Consider a situation where $T = 10$, and there are three jobs: $t_1 = 6$, $d_1 = 18$, $t_2 = 5$, $d_2 = 10$, $t_3 = 5$, $d_3 = 10$.

The greedy algorithm will pick job 1 first (since it pays 3 dollars per hour, versus 2 dollars per hour for each of the other two jobs), but then has no time left for another complete job, so it would pay a total of 18 dollars. However, the other two jobs together would take 10 hours and pay 20 dollars, so the greedy algorithm does not find the optimum.

3. (4 marks) You are trying to weigh a set of n books, where each book j has a corresponding positive height h_j. The weighing apparatus is a special type that needs to be adjusted to fit the height of the book, so weighing a book j will take time $1+|h_j-h_{j-1}|$, where h_j is the height of book j, and h_{j-1} is the height of the book weighed immediately before book j. The first book weighed (book 1) would take time $1+h_1$. You can weigh the books in any order.

(a) Design and write a greedy algorithm that will find the minimum time required to weigh all the books.

```
Algorithm MinTime(n, {h_j})
    sort the books by ascending (nondecreasing) height h_j
    time ← 1 + h_1
    for all books j in sorted order, from 2 to n
        time ← time + 1 + h_j − h_{j−1}
    return time
```

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

The sort, using mergesort, will take time $\in \Theta(n \log n)$. The for loop will have n iterations, and each iteration takes constant time. Since the rest of the algorithm takes constant time, the algorithm as a whole takes time $\in \Theta(n \log n)$.

(c) (5 marks) Prove that your algorithm works correctly.

Proof: Assume not.

Then there is some input such that the greedy algorithm does not produce an optimum. Let the greedy order be represented by $g_1, g_2, \ldots, g_n$, and consider the optimum order $o_1, o_2, \ldots, o_n$ that agrees with the greedy order the longest (maximising r such that $g_i = o_i \ \forall 1 \leq i < r$ and $o_r \neq g_r$).

Let k be the index such that $g_r = o_k$. Since $o_i = g_i$ for all $i < r$, and $o_r \neq g_r$, we must have $k > r$. Since the greedy algorithm sorts in ascending order, $o_k = g_r \leq o_r$.

Let Opt represent the value of the optimum. If we modify the optimum by moving o_k to just before o_r, we get the modified optimum result

$$ModOpt = Opt - (|o_r - o_{r-1}|) + (|o_k - o_{r-1}| + |o_r - o_k|) - (|o_k - o_{k-1}| + |o_{k+1} - o_k|) + (|o_{k+1} - o_{k-1}|)$$

Since $o_{r-1} \leq o_k \leq o_r$: $(|o_r - o_{r-1}|) = (|o_k - o_{r-1}| + |o_r - o_k|)$.
Since $o_k = g_r \leq o_{k-1}$ and $o_k = g_r \leq o_{k+1}$: $(|o_k - o_{k-1}| + |o_{k+1} - o_k|) \geq (|o_{k+1} - o_{k-1}|)$.

So $ModOpt \leq Opt$, and thus either improves or maintains the optimum while also agreeing with the greedy algorithm for a longer prefix. This contradicts our optimum being the optimum with the longest common prefix with the greedy.

Therefore our assumption is false, and the greedy algorithm is optimal.