CS3933 Winter 2007 – Group Assignment 3
Due Wednesday February 28, 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) (6 marks) Design and write an algorithm that will, given an undirected graph G, find the densest component of the graph. The density of a component is defined as its number of edges divided by its number of vertices.

   Analyse the running time of your algorithm.

```
Algorithm Densest(graph G = (V, E))
    Let Q be an empty queue
    Let C[] be an array of length |V| (one position per vertex)
    mark all vertices as unvisited
    mark all edges as unused
    c ← 0
    max ← 0
    for all s ∈ V
        if s is unvisited
          c ← c + 1
          enqueue s on Q
          C[s] ← c
          nodes ← 1
          edges ← 0
          while Q is not empty
                u ← dequeue(Q)
                for all v such that (u, v) ∈ E
                    if v is unvisited
                      mark v as visited
                      nodes ← nodes + 1
                      C[v] ← c
                      enqueue v
                    if (u, v) is unused
                      mark v as used
                      edges ← edges + 1
          if max < (edges/nodes)
```

$max \leftarrow (edges/nodes)$
$c_{max} \leftarrow c$
print "the densest component has density max, and consists of:"
for all $u \in V$
  if $C[u] = c_{max}$
    print u

Time analysis:
Despite the nested loops, each vertex is enqueued, dequeued, and visited exactly once. All edges are seen twice (once from each endpoint). Additionally, all edges and vertices are initialised once each, and the final for loop iterates through all vertices once more. Since the rest of the algorithm takes constant time, and this analysis is exact, the running time of this algorithm is $\in \Theta(m + n)$.

(b) (5 marks) Implement your algorithm from part (a) using adjacency lists to represent the graph. Hand in your source code and 3 test cases that demonstrate your algorithm well (give both the input and the result). To be kind to the marker, please also draw your input graphs.

Note to marker: please judge the implementation yourself; using adjacency lists is important (since other data structures will produce a larger time complexity). The test cases need to exercise both disconnected and connected graphs, and both sparse and dense components.

2. (6 marks) When we are searching for paths from a source vertex s to a target vertex t, a *bottleneck* vertex is a vertex (that is not either the source or the target) through which all paths from s to t go. Design and write an algorithm that will, given an undirected graph G, source vertex s and target vertex t, find a bottleneck vertex between s and t (if there is one).

Analyse the running time of your algorithm.

There are many potential approaches for this problem. Instead of providing a fully detailed solution to one approach (which might not help much in assessing others), here are the sketches of several different approaches. Of course, to be worth full marks, your solution would need to fill in the details and be structured properly.

(a) (removal approach – worth a little less)
Find a path from s to t using BFS (so that it will be shorter)
For each vertex in the path
remove the vertex
find if t is reachable from s (using BFS or DFS)

This approach would take $\Theta(mn + n^2)$ time in the worst case.

(b) (vertex merge approach)
Find a path from s to t using BFS (so that it will be shorter)
Search the graph using DFS, and find cycles on this path
For each cycle on this path, merge its path vertices
If there are any path vertices left (aside from a merged s and t), then they are bottlenecks

(c) (cycle-graph aproach)
Each backedge in DFS will correspond to a cycle (composed of tree edges and the one backedge). Find if there is a cycle between s and t by:
Building a cycle-graph with one node per DFS backedge
Using DFS in the original graph to find which cycles share an edge
add an edge between cycle-graph nodes if they share an edge
mark a pair of cycle-graph nodes if they share a vertex but not an edge
Using BFS or DFS in the cycle graph to see if there is a path from s to t

If there is no such path, then there is a bottleneck; it will be either a vertex shared by a pair of cycles on the s-t path, or a vertex not in any cycles on the s-t path.

This approach could take either $\Theta(mn + n^2)$ time or $\Theta(m + n)$ time, depending on the efficiency of the cycle testing.

3. You have been trying to shop for gifts, and have decided on a set of n specific items. You have also decided to buy one gift per day for the next n days (starting tomorrow).

Each gift i has a current price p_i, which is the price of that item today. However, all gifts have prices that change in a linear fashion with slope s_i, so that on day j, gift i will cost $p_i + s_i \cdot j$ (where today will be day 0, so gifts are bought on days 1 through n). Some slopes may be negative, but we must still buy one gift per day for the next n days. Do not worry about prices becoming negative. We can assume that all p_i and s_i values are distinct.

We want to find the order of gifts that minimises the total cost of the gifts; so we want to find a permutation of $1, 2, \ldots, n$, given by $o_1, o_2, \ldots, o_n$ (where gift i is bought on day o_i) that minimises

$$\sum_{i=1}^{n} (p_i + s_i \cdot o_i)$$

(a) (5 marks) One potential greedy approach would be to buy the currently most expensive gift first, and continue in descending order of p_i. Give a counterexample

and demonstrate why this approach will not always work.

Counterexample:
2 gifts, where $p_1 = 10$, $s_1 = -4$, $p_2 = 1$, and $s_2 = 50$.
The greedy algorithm would buy gift 1 on day 1 at a cost of $10 - 4 = 6$, and gift 2 on day 2 at a cost of $1 + 50 \cdot 2 = 101$, for a total cost of 107.
However, if the gifts are bought in the other order, gift 1 will be bought on day 2 at a cost of $10 - 4 \cdot 2 = 2$, and gift 2 will be bought on day 1 at a cost of $1 + 50 = 51$, for a total cost of 53, which is less than 107.
Therefore this greedy algorithm does not always produce the optimal order.

(b) (5 marks) Design and write a greedy algorithm that will always find the optimum order. Analyse the running time of your algorithm.

```
Algorithm Cheap(n, {p_i}, {s_i})
    sort the gifts by decreasing (nonincreasing) slope s_i
    cost ← 0
    for i from 1 to n (in sorted order)
        buy gift i on day i
        cost ← cost + (p_i + s_i · i)
    report minimum total cost cost
```

Time Analysis:
The sort takes $\Theta(n \log n)$ time (using mergesort). The for loop has exactly n iterations. Since the rest of the algorithm takes constant time, this algorithm as a whole takes time $\in \Theta(n \log n)$.

(c) (5 marks) Prove that your algorithm from part (b) is correct.

Proof: assume not, *i.e.* that there is some input for which the greedy algorithm produces an order $g_1, g_2, \ldots, g_n$ that is not optimal.

Let the order $o_1, o_2, \ldots, o_n$ represent the order that produces the minimum cost while maximising k such that $\forall 1 \leq i < k$, $o_i = g_i$, and $o_k \neq g_k$ (so that this order, of all the optimal orders, agrees with the greedy order for the longest).

Since $o_k \neq g_k$, and all previous gift choices are identical, the greedy property that selects g_k for nonincreasing slope thus ensures that $s_{o_k} \leq s_{g_k}$. The gift g_k will occur later in the optimal order (so $g_k = o_j$ for some $j > k$). Let O represent the optimal cost. If we modify the optimal order by swapping gift o_k with gift $o_j = g_k$, we will have an order with total cost

$$O + k \cdot (s_{o_j} - s_{o_k}) + j \cdot (s_{o_k} - s_{o_j})$$

$$= O + (k - j) \cdot (s_{o_j} - s_{o_k}) \;=\; O + (k - j) \cdot (s_{g_k} - s_{o_k})$$
$$\leq O \text{ since } (k - j) < 0 \text{ and } (s_{g_k} - s_{o_k}) > 0$$

So, we have either improved or maintained the minimum cost, while constructing an optimal order that agrees with the greedy order longer, contradicting that k was chosen to be maximum.

Therefore the assumption is wrong, and the greedy algorithm does always find the minimum total cost.