Sample Solution for CS3323 Fall 2006 Assignment 6 **(65 marks)**

Due Monday, Dec. 5, by midnight.

- Assignments should be handed in by placing them in the CS3323 bin on E level of Gillin Hall.

---

1. For each of the following, either draw a graph satisfying the given criteria or explain why it can't be done. Your graphs should not have any multi-edges (more than one edge between the same pair of vertices) or self-loops (edges with both ends at the same vertex).

   (a) **(5 Marks)**Draw a graph with 3 connected components, 12 vertices, and 18 edges.
   Solution: There are various solutions. Check whether it is the graph required.

   (b) **(5 Marks)** Draw a graph with 3 connected components, 12 vertices, and 8 edges.
   Solution: This is impossible. Let n_i be the number of vertices in the ith connected component. Then $\sum_{i=1}^{3} n_i = n = 12$. Connected component i must have at least $n_i - 1$ edges, or else it is not a single component. Since $\sum_{i=1}^{3} n_i - 1 = n - 3 = 9 > 8$, it is not possible to draw such a graph.

   (c) **(5 Marks)** Draw a graph with 3 connected components, 12 vertices, and 50 edges.
   Solution: This is also not possible. Let n_i be the number of vertices in the ith connected component. Then $\sum_{i=1}^{3} n_i = n = 12$. The maximum number of edges contained in component n_i is $n_i(n_i - 1)/2$. Thus the maximum number of edges in the whole graph is $\sum_{i=1}^{3} n_i(n_i - 1)/2$. This is maximized when two of the components have one vertex each, and the other has the remaining 10 vertices. In this case the maximum number of edges is $(10 \cdot 9)/2 = 45 < 50$ and so it is not possible to draw such a graph.

   (d) **(5 Marks)** Draw a graph with three vertices of degree 3, and four vertices of degree 2.
   Solution: The number of edges in a graph is the sum of the degrees of the vertices divided by 2, so for this graph the number of edges is $(3 \cdot 3 + 4 \cdot 2)/2 = 17/2$ — which isn't an integer! Thus it is not possible to draw such a graph. (In general, there must be an even number of vertices of odd degree for this very reason.)

2. Would you use the adjacency list or the adjacency matrix representation in each of the following cases? Justify your choice.

(a) **(5 Marks)** The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

(b) **(5 Marks)** The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

(c) **(5 Marks)** You need to answer as fast as possible the query **areAdjacent**, no matter how much space you use.

Solution:

(a) Adjacency list. In general, the adjacency list representation is preferrable when $|E| = O(|V|)$ because $O(|V|^2)$ cells are left empty in the adjacency matrix.

(b) Adjacency list (because there are no wasted cells), though the argument for using an adjacency matrix becomes stronger and one should do a careful analysis of the amount of space needed to store an item in the adjacency list versus the adjacency matrix. (It's possible that there is more overhead in the adjacency list structure and so when very few cells of the matrix are empty, the adjacency matrix uses less space.) In general, when $|E| = O(|V|^2)$, the amount of space wasted by the matrix is a constant and so it is a more desirable choice (especially if the application calls for many uses of the operations that are faster with an adjacency matrix).

(c) Adjacency matrix — **areAdjacent** is constant-time for an adjacency matrix, whereas it depends on the degree of the vertices in question in an adjacency list.

3. Would you prefer DFS or BFS (or both equally) for the following tasks? (Assume the graph is undirected and connected.) Justify your answer.

(a) **(5 Marks)** Determining if the graph is acyclic

Solution: Probably DFS. The solution can be found easily with either DFS or BFS since as soon as you find a back edge or cross edge you know the graph has a cycle. The tie-breaker here is the extra storage space required — DFS is preferable because it will generally require less additional storage space than BFS. BFS requires space proportional to the maximum number of vertices in a given level, whereas DFS requires space proportional to the number of recursive calls (which is at most the length of the longest path from the start vertex to any other vertex), and, in general, the number of recursive calls will be less than the maximum number of vertices on a given level, but not always. For examples at both extremes, consider a tree where every internal node has b children. In the minimum height tree, the height (and thus space required by DFS) is $O(\log n)$, whereas the maximum number of nodes in a level (and thus space required by

BFS) is $O(n)$. In the maximum height tree, the height/DFS space is $O(n)$ and the maximum number of nodes in a level/BFS space is $O(b)$.

(On a side note, for directed graphs, DFS is the traversal of choice since a back edge still means a directed cycle, but a cross edge in the BFS doesn't not necessarily imply this.)

(b) **(5 Marks)** Finding a path to a vertex known to be near the starting vertex

Solution: BFS. Since BFS visits the vertices in order of the shortest path between them and the starting vertex, it will tend to find nearby vertices faster. DFS may luck out and find the vertex quickly, but it also may end up searching nearly all of the graph first. (To see this, consider searching in a tree where every internal node has b children, and the depth of every external node is d. The search starts at the root and you're looking for a node at depth k. BFS must visit at least all of the nodes at depths less than k ($\sum_{i=0}^{k-1} b^i = \frac{b^k-1}{b-1}$ nodes), plus potentially all of the nodes at depth k because the target node could be the last thing found, for a maximum of $\sum_{i=0}^{k} b^i = \frac{b^{k+1}-1}{b-1} = O(b^k)$ nodes visited. DFS might visit as few as k nodes if it picks the right child to visit first at each node, or as many as $\frac{b^{d+1}-b^{d-k+1}}{b-1} + 1 = O(b^d)$ nodes if it picks the right child last at each node. (In this second case, DFS ends up visiting all of the nodes of the tree except those in the subtree rooted at the target node.) Since a tree is just a special kind of graph, this shows that there are at least some graphs for which DFS can be much worse than BFS if $k << d$.)

(c) **(5 Marks)** Finding the connected components of the graph

Solution: Probably DFS. There is no real difference in terms of solution simplicity, since you just pick a starting vertex, do DFS/BFS until everything that can be reached from the start vertex has been reached, and then repeat with a new starting vertex not yet visited. The deciding factor is again the amount of extra storage space required.

4. **(10 Marks)** Describe the pseudo-code of a non-recursive DFS algorithm.

Algorithm Iterative-DFS(graph G):

```
Input graph G
Output labeling of the edges of G as discovery edges and back edges
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
```

```
    for all v ∈ G.vertices()
        if getLabel(v) = UNEXPLORED
          Iterative-DFS(G, v)
```

Algorithm Iterative-DFS(G, v):

```
    Stack L.Push(v)
    setLabel(v, VISITED)
    while (L is not empty)
          u ← L.Pop()
          for all e ∈ G.incidentEdges(u)
              if getLabel(e) = UNEXPLORED
                w ← opposite(v,e)
                if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  L.Push(w)
                else
                  setLabel(e, BACK)
```