CSc 3102: Search and Traversal of Graphs

Supplementary Notes

- Breadth-First Search
- Depth-First Search

Introduction

Many important problems require an examination (visit) of the vertices of a graph. Bread-first search and depth-first search are the two most common traversal strategies.

Breadth-First Search

The strategy underlying breadth-first search is to visit all unvisited vertices adjacent to a given vertex before moving on. It is easily implemented by visiting these adjacent vertices and then enqueueing them. A vertex is then dequeued and the process is repeated. The algorithm below may be used to obtain the breadth-first tree of a connected graph.

Listing 1: Breadth-First-Search Traversal Algorithm

```
ALGORITHM: BFS(G,v)
   {Input: G - a graph with n vertices
           v is some vertex in G, usually
           lexicographical ordering is assumed
           in selecting vertices.
           Q is a queue
   }

   mark v 1
   Q.enqueue(v)
   while Q is not empty
      u <- Q.dequeue()
      mark u 2
      visit(u)
      for each vertex w adjacent to u and marked 0
         mark w 1
         Q.enqueue(w)
endAlgorithm
```

If the graph is any arbitrary graph (connected or not), this algorithm which calls the one above can be used.

Listing 2: BFS Algorithm Wrapper

```
ALGORITHM: BFT(G)
   Input: G - a graph with n vertices
   if G is not empty
      mark all vertices 0
   for v <- 1 to n do
      if v is marked 0
         BFS(G,v)
endAlgorithm
```

Analysis: BFS is $O(n+m)$, where n is the number of vertices and m is the number of edges, for the adjacency linked list implementation. The adjacency matrix implementation is $O(n^2)$

Depth-First Search

The search philosophy of depth-first search is a desire to see what is ahead. It moves immediately on to an unvisited neighbor, if one exists, after visiting a node. Whenever it is at an explored node it backtracks until an unexplored node is encountered and then it continues. Here is a non-recursive version of the algorithm that determines the depth-first search tree of a connected graph.

Listing 3: Pre-Order DFS Algorithm

```
ALGORITHM: DFS(G,v)
   {Input: G - a graph with n vertices
           v is some vertex in G, usually
           lexicographical ordering is assumed
           in selecting vertices.
           S is a stack
    This algorithm does pre-order DFS Traversal.
   }

   mark v 1
   S.push(v)
   while S is not empty
      u <- S.pop()
      mark u 2
      visit(u)
      for w in adj(u) marked 0
         mark w 1
         S.push(w)
endAlgorithm
```

If the graph is any arbitrary graph (connected or not), this algorithm which calls the one above can be used.

Listing 4: Pre-Order DFS Algorithm Wrapper

```
ALGORITHM: DFT(G)
   Input: G - a graph with n vertices
   if G is not empty
      mark all vertices 0
   for v <- 1 to n do
      if v is marked 0
         DFS(G,v)
      endif
   endfor
endAlgorithm
```

Analysis:
If G is connected, then every vertex is visited exactly once. So there are n nodes visits. The worst-case complexity of the algorithm is $O(n+m)$, for the adjacency linked list implementation. The adjacency matrix implementation is $O(n^2)$.

Problem 1. How would you modify the DFS algorithm so that it performs post-order DFS Traversal?

Problem 2. Draw the tree induced by a BFS traversal of the graph in figure (a) on p.371 of your textbook. Repeat this exercise for a DFS traversal.