Searching Graphs and Digraphs

Textbook:

Chapter 5, Section 5.4, pp. 216-231

DFS vs. BFS

- Depth-First Search (DFS)
  - Starting with an initial vertex *v*, DFS searches all vertices that are connected to *v*, i.e., in the connected component of *v.*
  - Action is controlled by a stack.
  - Search goes deep and involves backtracking
- Breadth-First Search (BFS)
  - Starting with an initial vertex *v*, BFS searches all vertices that are connected to *v*.
  - Action is controlled by a queue.
  - Search is broad. The entire neighborhood for the current node (one dequeued) is visited.

Depth-First Search (DFS)

procedure *DFS*(*G*,*v*) **recursive**

Input: *G* (a graph with *n* vertices and *m* edges)

v (a vertex where search begins)

Output: the depth-first search of *G* with starting vertex *v*

Mark[*v*] ← 1 // mark *v* as visited

call *Visit*(*v*)

for each vertex *u* adjacent to *v* **do**

if *Mark*[*u*] = 0 **then call** *DFS*(*G*,*u*) **endif**

endfor

end *DFS*

Non-recursive version of DFS

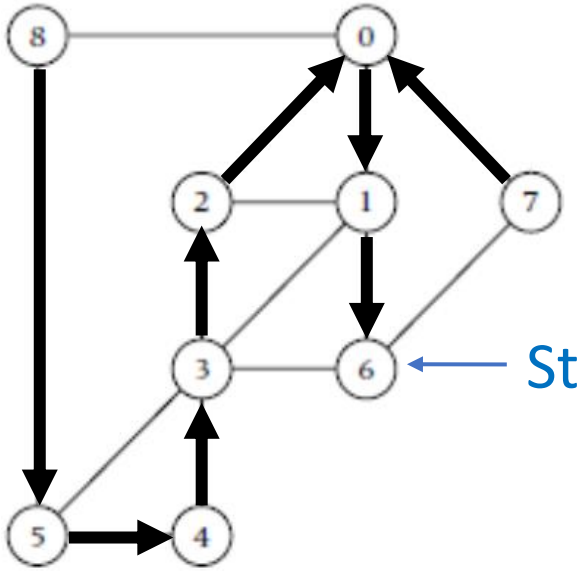
```
procedure DFS(G,v)
Input:    G (a graph with n vertices and m edges)
          v (a vertex)
Output:   the depth-first search of G starting from vertex v
          S a stack initialized as empty
      Mark[v] ← 1    // mark v as visited
       call Visit(v)
       u ← v
       Next(u,w,found)     //find node w in neighborhood of u that is unvisited
       while found .or. (.not. Empty(S))
            if found then        //go deeper
                 Push(S,w)
                 Mark[w] ← 1
                 Visit(w)
                 u ← w
            else
                 Pop(S,u)        //backtrack
            endif
            Next(u,w,found)
      endwhile
end DFS
```

PSN. Modify the recursive version of *DFS* to output the **parent array** of the *DFS* tree.

Action of DFS for Sample Graph and Initial Vertex

We will assume graph *G* is implemented with it adjacency matrix so smallest label in the neighborhood that is not visited is chosen next. If *G* is implement using adjacently list than the next unvisited node in the adjacently list is chosen.

G =



Starting at vertex *v* = 6

DFS tree shown in bold.

DFS order (order in which vertices are visited): 6, 1, 0, 2, 3, 4, 5, 8, 7

PSN. Repeat the action of DFS in computing the DFS Tree and give the DFS order, where the initial vertex is $v = 0$.

Complexity Analysis of DFS

In the worst-case DFS will access edge of G.
Therefore, DFS has linear computing time,
i.e., complexity $O(m)$.

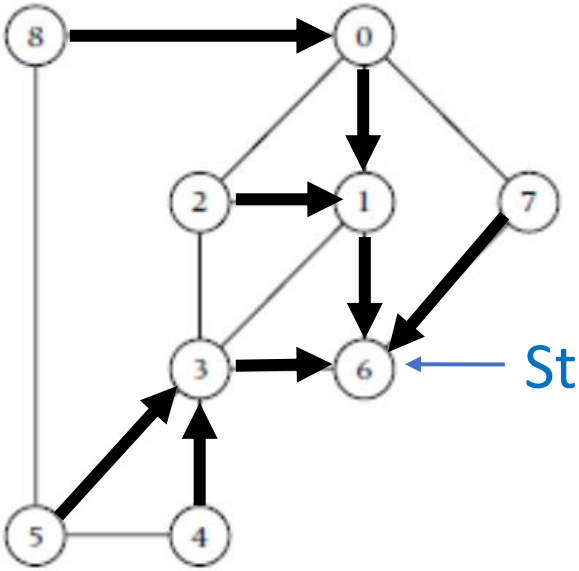
Breadth-First Search – BFS

```
procedure BFS(G,v, Parent[0:n – 1] )
Input:    G (a graph with n vertices and m edges)
          v (vertex)  //the array Mark[0:n– 1] is global and initialized to 0s
          Parent[0:n – 1] (parent array for BFS tree rooted at v)
Output:   the breadth-first search of G starting from vertex v
      Q a queue initialized as empty
      call Enqueue(Q,v)
      Mark[v] ← 1   // mark v as visited
      call Visit(v)
      while .not. Empty(Q) do
            Dequeue(Queue,u)
            for each vertex w adjacent to u do
                  if Mark[w] = 0 then
                        Enqueue(Q,w)
                        Mark[w] ← 1
                        Visit(w)
                        Parent[w] ← u
                  endif
            endfor
       endwhile
end BFS
```

Action of BFS for Sample Graph and Initial Vertex

We will assume graph *G* is implemented with it adjacency matrix so smallest label in the neighborhood that is not visited is chosen next. If *G* is implement using adjacently list than the next unvisited node in the adjacently list is chosen.

G =



Starting at vertex *v* = 6

BFS tree shown in bold.

BFS order (order in which vertices are visited): 6, 1, 3, 7, 0, 2, 4, 5, 8

PSN. Repeat the action of BFS in computing the BFS Tree and give the DFS order, where the initial vertex is $v = 0$.

Complexity Analysis of BFS

In the worst-case BFS will access every edge of G. Therefore, BFS has linear computing time, i.e., complexity $O(m)$.

Traversing Disconnected Graphs

- If the graph is disconnected DFS and BFS do not visit all the vertices.
- In this case all the vertices can be visited using a Depth-First Traversal (DFT) or a Breadth-First Traversal (BFT).
- The idea is to scan all the vertices, i.e., $v = 0, 1, \ldots, n - 1$, and, if v is unvisited, perform a DFS or BFS at v.
- We obtain a DFT forest or BFT forest.

Depth-First Traversal – DFT

procedure *DFT*(*G*,*Parent*[0:*n* – 1])
Input: *G* (a graph with *n* vertices and *m* edges)
Output: Depth-first traversal
Parent[0:*n*– 1] (an array implementing the *DFT* forest of *G*)

```
    dcl Mark[0:n– 1] a 0/1 array initialized to 0s
    for v ← 0 to n – 1 do
        Parent[v] ← -1
    endfor
    for v ← 0 to n – 1 do
        if Mark[v] = 0 then
            DFS(G,v,Parent[0:n – 1])
        endif
    endfor
end DFT
```

Breadth-First Traversal – BFT

procedure *BFT*(*G*,*Parent*[0:*n* – 1])
Input: *G* (a graph with *n* vertices and *m* edges)
Output: Depth-first traversal
Parent[0:*n*– 1] (an array implementing the *BFT* forest of *G*)

```
    dcl Mark[0:n– 1] a 0/1 array initialized to 0s
    for v ← 0 to n – 1 do
        Parent[v] ← -1
    endfor
    for v ← 0 to n – 1 do
        if Mark[v] = 0 then
            BFS(G,v,Parent[0:n – 1])
        endif
    endfor
end DFT
```

Complexity Analysis of DFT and BFT

DFT or BFT will access every vertex and edge of G. Therefore, they have complexity $O(m + n)$.

Search and Traversal of Digraphs

DFS and BFS are the same for digraphs except that we must choose the edges in a consistent directions, i.e., when the current vertex is u we only consider the vertices w in the out-neighborhood of v, i.e., having tail v and head w. This generates DFS Tree or BFS Tree that is **out-directed** from the root (initial vertex) *v*.

We could take the in-neighborhood instead, in which case we get a DFS or BFS tree that is **in-directed** to the root *v*.

Important Applications

1. Testing whether graph is connected
2. Finding Shortest Paths
3. Computing the connected components
4. Computing the diameter

Testing whether graph is connected

Apply either DFS or BFS starting at any vertex. If all other vertices are visited the graph is connected, otherwise it is disconnected.

Finding Shortest Paths

- Compute the BFS Tree *T* rooted at *v*. Then, for every vertex *w* in *T*, the path in the *T* from *v* to *w* is a shortest path, i.e., has shortest length over all paths from *v* to *w* in the graph or digraph.
- Complexity to compute shortest paths from a given vertex *v* to all other vertices that are connected to *v* is $O(m)$.
- Let *dist*(*v*,*w*) = distance from *v* to *w.* BFS can compute *dist*(*v*,*w*) by simply adding 1 to *dist*(*v*,*u*) when *w* is enqueued by *v*, where *v* is the parent of *w* in the BFS tree
- This applies for both graphs and digraphs.

Computing the Connected Components

Perform a DFT or BFT.

Each tree in the DFT or BFT forest spans the a connected component.

Complexity is $O(m + n)$

Computing the Diameter of a Connected Graph

Initialize *Diameter* to 0.

Perform a BFS at each vertex $v = 0, 1, \ldots, n - 1$, to compute $dist(v,w)$, for all $w \in V$, and if it is greater than Diameter, update Diameter to $dist(v,w)$.

Complexity is $O(mn)$

What do you call a dinosaur that asks a lot of deep and searching questions?

A philosiraptor.