Data Structures Mini Project

Submitted
by

M. VENKATARAMANA , 06MA6020
TAPAN MAHAPATRA, 06MA6019
NAVEEN K, 06EE6404
K V S SRILATA, 06EE6401

Under the guidance
Of
Prof: G.P Raja Sekhar
Dept of Mathematics
IIT Kharagpur

Contents

Introduction:

If you want to go from Point A to Point B, you are employing some kind of search. For a direction finder, going from Point A to Point B literally means finding a path between where you are now and your intended destination. For a chess game, Point A to Point B might be two points between its current position and its position 5 moves from now. For a genome sequencer, Points A and B could be a link between two DNA sequences.

As you can tell, going from Point A to Point B is different for every situation. If there is a vast amount of interconnected data, and you are trying to find a relation between few such pieces of data, you would use search.

Searching falls under Artificial Intelligence (AI). A major goal of AI is to give computers the ability to think, or in other words, mimic human behavior. The problem is, unfortunately, computers don't function in the same way our minds do. They require a series of ***well-reasoned out*** steps before finding a solution. Our goal, then, is to take a complicated task and convert it into simpler steps that your computer can handle. That conversion from something complex to something simple is what this tutorial is primarily about. Learning how to use two search algorithms is just a welcome side-effect

Breadth First Search:

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs. Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1.

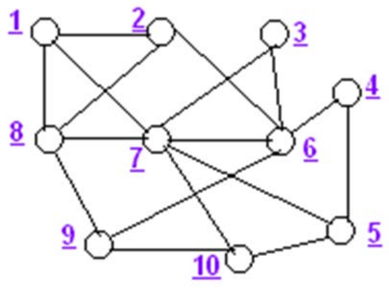
To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If (u, v) E and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the predecessor or parent of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent. Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on a path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

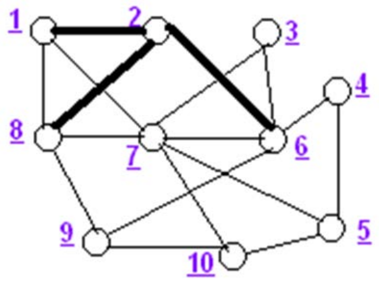
Example:

Figure 1: BFS- Given graph (a) with Source 2

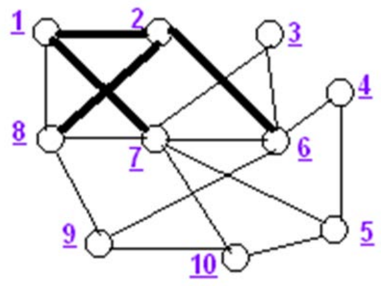
(a):



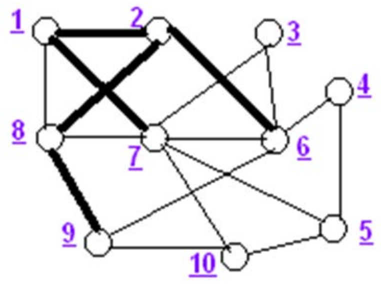
(b):



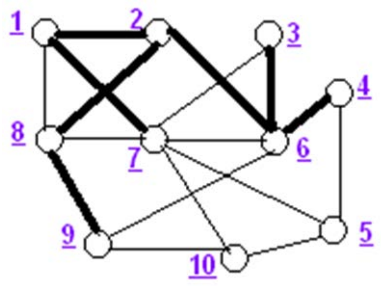
(c):



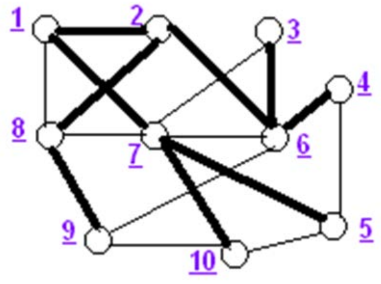
(d):



(e):



(f): This is the ultimate BFS tree with source 2.



Depth First Search:

In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in breadth-first search, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, depth-first search records this event by setting v's predecessor field p[v] to u. Unlike breadth-first search, whose predecessor sub-graph forms a tree, the predecessor sub-graph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources. The predecessor sub-graph of a depth-first search is therefore defined slightly differently from that of a breadth-first search:

Let G = (V, E), where E = $\{(p[v], v) : v \in V \text{ and } p[v] \neq \text{NIL}\}$.

The predecessor sub-graph of a depth-first search forms a depth-first forest composed of several depth-first trees. The edges in E are called tree edges.

As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also timestamps each vertex. Each vertex v has two timestamps: the first timestamp d[v] records when v is first discovered (and grayed), and the second timestamp f [v] records when the search finishes examining v's adjacency list (and blackens v). These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search.

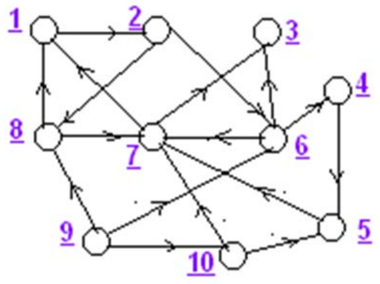
The procedure DFS below records when it discovers vertex u in the variable d[u] and when it finishes vertex u in the variable f [u]. These timestamps are integers between 1 and 2 |V|, since there is one discovery event and one finishing event for each of the |V| vertices. For every vertex u, d[u] < f[u].

Vertex u is WHITE before time d[u], GRAY between time d[u] and time f [u], and BLACK thereafter.

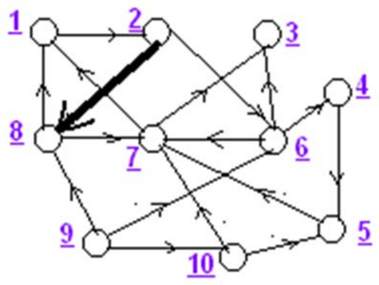
Example:

Figure 2: DFS- Given digraph (a) with source 2

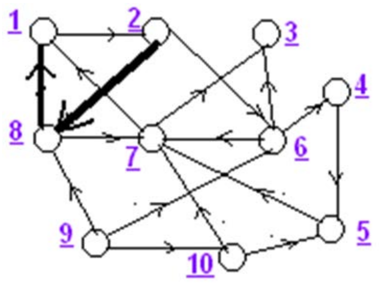
(a):



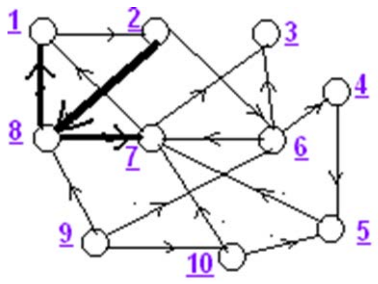
(b):



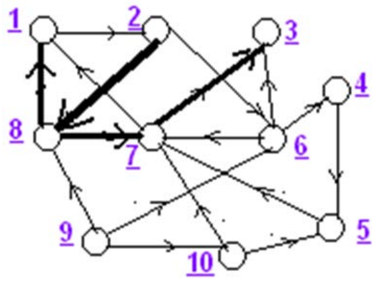
(c):



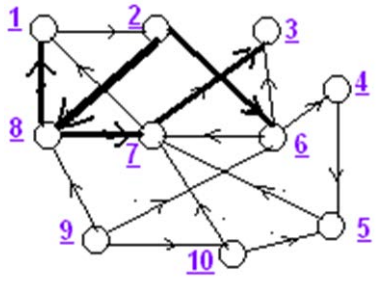
(d):



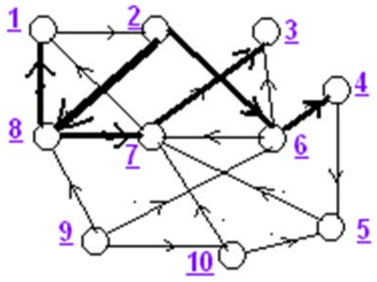
(e):



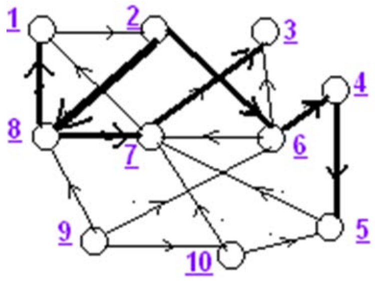
(f)



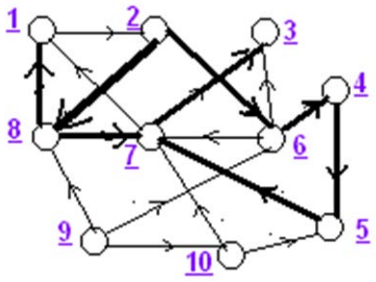
(g):



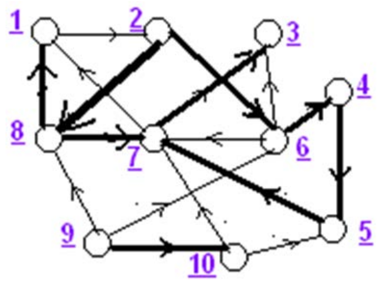
(h):



(i):



(j):



Pseudo code for BFS: BFS(G,s)

```
1.  for each vertex u ε V [G] - {s}
2.      do color[u] = WHITE
3.         d[u] = ∞
4.          p[u] = NIL
5.  Color[s] = GRAY
6.  d[s]= 0
7.  p[s] =NIL
8.  Q = Ø
9.  ENQUEUE (Q, s)
10.  While (Q != Ø)
11.      do u =DEQUEUE (Q)
12.         for each v ε Adj[u]
13.             do if color[v] = WHITE
14.                   then color[v] = GRAY
15.                        d[v] = d[u] + 1
16.                        p[v] = u
17.                         ENQUEUE (Q, v)
18.         color[u] = BLACK
```

Pseudo code for DFS:

```
DFS(G)
1.  for each vertex u ε V [G]
2.      do color[u] = WHITE
3.          p [u] = NIL
4.  time = 0
5.  for each vertex u ε V [G]
6       do if color[u] = WHITE
7           then DFS-VISIT (u)

DFS-VISIT (u)
1.  Color[u] =GRAY    // White vertex u has just been discovered.
2.  time = time +1
3.  d[u]= time
4.  for each v ε Adj[u]
5.      do if color[v] = WHITE
6.          then p[v] = u
7.              DFS-VISIT(v)
8.  color[u] =BLACK      //Blacken u; it is finished.
9.  f [u] = time = time +1
```

Complexity:

BFS:

The operations of enqueuing and dequeuing take O(1) time, so the total time devoted to queue operations is O(V). Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency list is Θ(E), the total time spent is scanning adjacency lists is O(E). The overhead for initialization is O(V), and thus the total running time of BFS is O(V+E). Thus, breath-first search runs in time linear in the size of the adjacency-list representation of graph G.

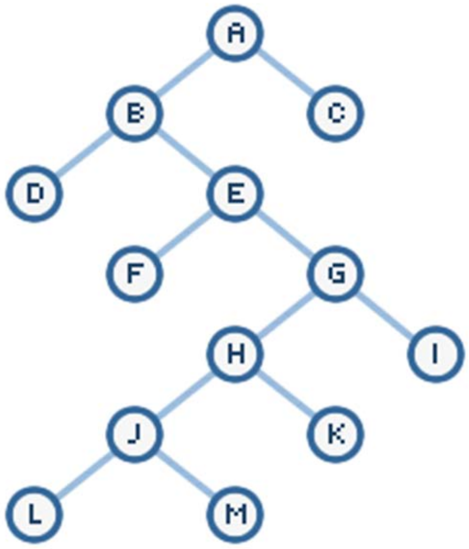
DFS:

The loops on lines 1-3 and lines 5-7 of DFS take time Θ(V), exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex v ε V , since FS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray. During an execution of DFS-VISIT(v), the loop on lines 4-7 is executed |Adj[v]| times. Since

$\sum |Adj[v]| = \Theta(E)$, the total cost of executing line 4-7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V+E)$.

Advantages and Disadvantages:

If we have tried various combinations of search results, you can tell that each search method has its advantages and disadvantages. Let's the below given search tree:



When you use depth first search, what shall you think the path between nodes A and C will be? The following is what the path actually will be:

```
    a, b, d, e, f, g, h, j, l, m, k, i, c
```

Are not we surprised at how long the path was even though C is a direct neighbor of A? What happens if we repeat the same example using the breadth first search example?

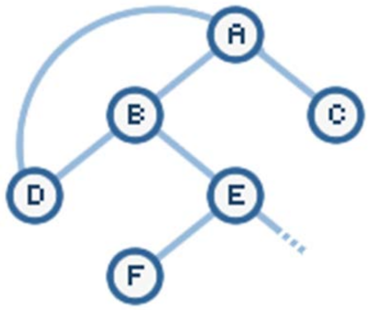
Likewise, compare the path produced by depth first search and breadth first search when going from A to D. In this case, depth first search produces a better result.

Because of the way both of these algorithms work, we will often miss the forest for the trees. Despite C being just on hop away from its destination; depth first search will avoid it for a long time because the nodes it examines first are those of its neighbors as opposed to nodes on the same depth.

We will find numerous such abnormal behaviors in both methods. This is because both search methods are considered blind searches. They don't know anything about their future or where the target is. The paths the expand are mechanically defined. If a better path exists, they will not take it. If they are taking the wrong path, they won't know it.

For many real-world situations, though, we will not have nicely balanced trees such as what you have seen in this tutorial. Ideally, we would let these algorithms loose on graphs containing hundreds of thousands of nodes with a myriad of edges. After all, if we could solve these ourselves, we would have no need to have a computer do it for us.

With large graphs, we may run into cycles or loops. If we really want to see our depth first search algorithm fail, simply add an edge between nodes D and node A:



We will eventually crash the system because depth first search will simply keep adding in the values a, b, and d continuously without ever breaking out of the loop.

Another problem is, what if the left side of your tree has millions of nodes, but your destination is close to the origin on the right side of the tree. Depth first would waste numerous cycles exploring the left side before ever reaching the right side.

Breadth first search does not suffer from the same loop problems because it moves horizontally across each depth. Breadth first will always find a solution regardless of what type of search tree we have unless there are infinite nodes.

Memory is often a limiting factor. Having millions or billions of nodes, as is the case with searching the web, both depth first and breadth first searches are at the mercy of the computers powering them. Fortunately, there are many better, more complicated search methods that address all of the problems of both depth and breadth first searches AND produce far better results.

Applications of BFS:

BFS can be used to solve many problems in graph theory, for example:

1. Finding the shortest path between two node u & v (in an unweighted graph)
2. Testing a graph for bipartiteness
3. (Reverse) Cuthill-McKee mesh numbering

Testing bipartiteness:

BFS can be used to test bipartiteness, by starting the search at any vertex and giving alternating labels to the vertices visited during the search. That is, give label 0 to the starting vertex, 1 to all its neighbors, 0 to those neighbors and so on. If at any step a vertex has (visited) neighbors with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

Usage in 2D grids for computer games:

BFS has been applied to path finding problems in computer games, such as Real-Time Strategy games, where the graph is represented by a tile map, and each tile in the map represents a node. Each of that node is then connected to each of its neighbor (neighbor in north, north-east, south-east, south, south-west, west, and north-west).

It is worth mentioning that when BFS is used in that manner, the neighbor list should be created such that north, east, south and west get priority over north-east, south-east, south-west and north-west. The reason for this is that BFS tends to start searching in a diagonal manner rather than adjacent, and the path found will not be the correct one. BFSD should first search adjacent nodes, then diagonal nodes.