Graph traversal

One problem we often need to solve when working with graphs is to decide if there is a sequence of edges (a path) from one vertex to another, or to find such a sequence. There are various versions of this problem, and the most familiar one to you is probably Google Maps which finds the shortest path from one location to another. You will learn more about shortest path problems in COMP 251.

Today we will consider the problem of finding the set of all vertices that can be reached from a given vertex v, or equivalently, the set of all vertices w for which there is a path from v to w. In the shortest path problem you will see in COMP 251, "shortest" refers to the sum of the edge weights and one tries to find the path from v to w that minimizes this sum.

Depth First Traversal

Recall the depth first traversal algorithm for trees.

```
depthfirst_Tree(root){
   if (root is not empty){
      visit root                        // preorder
      for each child of root
         depthfirst_Tree(child)
   }
}
```

This algorithm generalizes to graphs as follows.

```
depthfirst_Graph(v){
   v.visited = true
   for each w such that (v,w) is in E
      if !w.visited                              //  avoids cycles
         depthfirst_Graph(w)
}
```

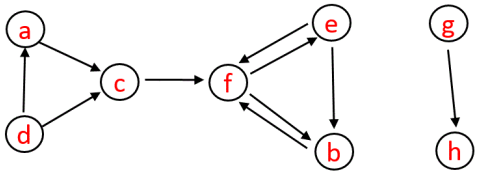
A few notes: first, I call this algorithm a "traversal" but in fact it only reaches the nodes in the graph for which there is a path from the input vertex. This algorithm is sometimes called (depth first) "search" since we are searching for all vertices that can be reached from the input vertex.

Second, before running this algorithm, we need to set the `visited` field to false for all vertices in the graph. To do so, we need to access *all* vertices in the graph. This is a different kind of traversal, which is independent of the edges in the graph. For example, if you were to use a linked list to represent all the vertices in the graph, then you would traverse this linked list and set the `visited` field to false, before you called the above traversal algorithm. If you were using a hash table to represent the vertices in the graph, you would need to go through all buckets of the hash table by iterating through the hash table array entries and following the linked list stored at each entry. You would set the `visited` field to false on each vertex (value) in each bucket.

Example *(see slides for another example)*—

Let's run the above preorder depth first traversal algorithm on the graph shown below. Also shown is the *call stack* and how it evolves over time, and the *call tree*. (A node in the call tree represents one "call" of the `depthfirst_Graph` method. Two nodes in the call tree have a parent-child relationship if the parent node calls the child node.)

Note that the call stack is actually constructed when you run a program that implements this recursive algorithm, whereas the call tree is not constructed. The call tree is just a way of thinking about the recursion.




```
      CALL STACK                CALL TREE
        b   e                       a
        f f f f f                   |
      c c c c c c c                 c
    a a a a a a a a                 |
                                    f
-----------------> time           / \
                                  b   e
```

Notice that nodes `d`, `g`, `h` are not visited.

Non-recursive depth first traversal

We do not need recursion to do a depth first traversal. We can do depth first traversal using a stack. Our algorithm here generalizes the non-recursive tree traversal algorithm that used a stack.

The tree traversal algorithm using a stack went like this:

```
treeTraversalUsingStack(root){
  s.push(root)
  while !s.isEmpty(){
    cur = s.pop()
    visit cur
    for each child of cur
       s.push(child)
    }
}
```

Here we visited each node after popping it from the stack. Let's rewrite this slightly so that we visit each node *before* pushing it onto the stack.

```
treeTraversalUsingStack(root){
   visit root
   s.push(root)
   while !s.isEmpty(){
      cur = s.pop()
      for each child of cur{}
         visit child
         s.push(child)
      }
   }
}
```

Now let's generalize this to graphs. In the tree case, we pushed the children of a node onto the stack. In the graph case, we will push the adjacent vertices (nodes) onto the stack. *We only do so, however, if the adjacent vertex has not yet been visited.* The reason is that the graph may contain a cycle and we want to ensure that a vertex does not get pushed onto the stack more than once, since this would lead to an infinite loop.

```
depthFirst(v){
   initialize empty stack s
   v.visited = true
   s.push(v)
   while (!s.empty) {
     u =  s.pop()
     for each w in u.adjList{
        if (!w.visited){
           w.visited = true
           s.push(w)
        }
}
```

Note that this still is a preorder traversal. We visit a vertex before we visit any of the children vertices.

Breadth first traversal

Like depth first traversal, breadth first traversal finds all vertices that can be reached from a given vertex v. However, breadth first traversal visits all vertices that are one edge away, before it visits any vertices are two vertices away, etc. We have already seen breadth first traversal in trees, also known as level order traversal. Breadth traversal in graphs is more general: the levels correspond to vertices that are a certain distance away (in terms of number of edges i.e. path length) from a given vertex.

Since we are working with a graph rather than a tree, we visit the nodes before enqueueing them.

```
breathFirst(u){
  initialize empty queue q
  u.visited = true
  q.enqueue(u)
  while ( !q.empty) {
    v = dequeue()
    for each w in adjList(v)
      if !w.visited{
        w.visited = true
        enqueue(w)
      }
}
```

Notice that we enqueue a vertex only if we have not yet visited it, and so we cannot enqueue a vertex more than once. Moreover, all vertices that are enqueued are dequeued, since the algorithm doesn't terminate until the queue is empty.

Take the same example graphs as before. Below we show the queue `q` as it evolves over time, namely we show the queue at the end of each pass through the `while` loop.

Since this is not a recursive function, we don't have a "call tree". But we can still define a tree. Each time we visit a vertex – *i.e.* `w in adjList(v)` and we set `w.visited` to `true` – we get a edge `(v, w)` in the tree. We can think of the **w** vertex as a child of the `v` vertex, which is why we are calling this a tree. Indeed we have a rooted tree since we are starting the tree at some initial vertex that we are searching from.

Note how the queue evolves over time, and the order in which the nodes are visited. Also note that three in this case happens to be the same as the tree defined by the depth first traversal.

```
   QUEUE q    TREE
(snapshots)
   a           a
   c           |
   f           c
   be          |
   e           f
             /   \
            b     e

  order visited = acfbe
```

Another Example

Here is another example, which better illustrates the difference between `dft` and `bft`. For the sake of simplicity (drawing!), I suppose here that the graph is undirected. In the slides, I did a directed version of this.

```
  GRAPH      ADJACENCY LIST    depth first       depth first         breadth first
                               (recursive)         (stack)            (queue)

a - b - c    a - (b,d)         a - b - c         a - b   c           a - b - c
|   |   |    b - (a,c,e)              |         |       |           |   |   |
d - e - f    c - (b,f)         d - e - f         d - e   f           d   e   f
|   |   |    d - (a,e,g)       |                 |       |           |   |   |
g - h - i    e - (b,d,f,h)     g - h - i         g - h - i           g   h   i
             f - (c,e,i)
             g - (d,h)
             h - (e,g,i)
             i - (f,h)

           order visited:      abcfedghi         abdeghifc           abdcegfhi
```

[ADDED Nov. 22]

For the depth first stack case, the order listed in the originally posted notes and slides was incorrect. We visit before we push. (I had notice this mistake during one of the two sections during the lecture, but I must have forgotten to correct it. Sorry for the confusion!)

Note that in both the depth first stack and the breadth first case, the order in which vertices are visited here cannot be directly inferred from the structure of the tree.