

Graph traversal

One problem you often need to solve when working with graphs is whether there is a sequence of edges (a path) from one vertex to another or, more generally, what is the set of all vertices that can be reached from a given vertex v , that is, the set of all vertices w for which there is a path from v to w . By a *path* here, I mean a sequence of vertices $v_1 v_2 \dots v_k$ such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \dots, k - 1$, and $v_1 = v$ and $v_k = w$.

Depth First Traversal

Recall the depth first traversal algorithm for trees. This algorithm generalizes to graphs as follows. The algorithm is similar to preorder traversal of a tree.

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

Before running this algorithm, you would need to set the `visited` field to false for all vertices in the graph. Of course, your graph data structure needs to allow you to access all vertices in the graph. For example, if you are using a hash table to represent all vertices, you can walk through all buckets of the hash table by iterating through the hash table array entries (buckets) and following the linked list stored at each entry. You set the `visited` field to false on each vertex (value) in each bucket.

After running the algorithm, the vertices that have `v.visited == true` are the vertices that can be reached by a path from the vertex that you started with, namely the input vertex of the algorithm. You could generate a list of these vertices “on the fly” e.g. by adding an instruction that says to add an item to a list (say, right before or after the `visited` field is assigned `true`).

Another point discussed in class is whether or not a postorder traversal is possible. Yes, it is, but we need to be careful. With graphs there can be cycles, namely paths that contain the same node more than once. Your algorithm needs to ensure that it doesn’t visit a node more than once (which could send you into an infinite loop i.e. u to v to w to u etc). The following will NOT work, for example, if the graph consists of a single cycle:

```
depthFirstTraversal(v){
    for each w such that (v,w) is in E
        if !w.visited
            DepthFirstTraversal(w)
    v.visited = true
}
```

To make a post-order traversal work, we need to distinguish between “reaching” a vertex and “visiting” a vertex. Reaching a vertex could just mean that we access the adjacency list of the vertex, whereas visiting it might mean that we write into a field in the vertex. The following works – in the sense that it avoids the infinite loop from a cycle, and also allows us to visit a vertex after all the adjacent vertices have been visited.

```

depthFirstTraversal(v){
  v.reached = true
  for each w such that (v,w) is in E
    if !w.reached
      DepthFirstTraversal(w)
  v.visited = true
}
    
```

Notice that for the above algorithms to work properly, the fields `visited` and `reached` should both be initialized to false.

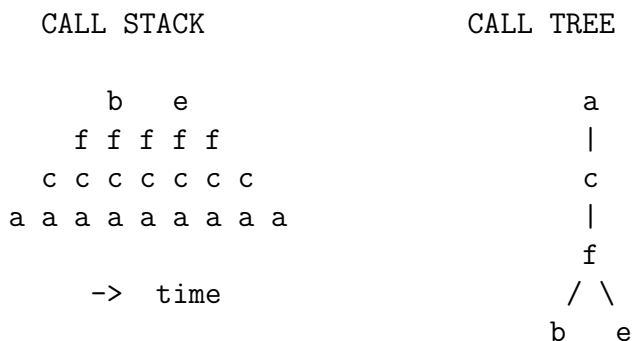
Example (slightly different from slides)

Let's run the depth first traversal algorithm on the *directed graph* from last lecture i.e. the graph on the right which had some directed edges between vertices.



Below is the *call stack* (evolving over time) for the dft recursion. Also shown is a tree, known as the *call tree*. A node in the call tree represents one “call” of the `dft` 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.



Breadth first traversal

Like depth first traversal, “breadth first traversal” searches for all the vertices that can be reached from a given vertex *v*. The difference is that breadth first traversal attempts to create paths from a given vertex that are as short as possible. First, it visits vertices that are a distance 1 away. Then it visits vertices that are a distance 2 away, etc.

We have already seen an example of breadth first traversal, namely level order traversals in trees. Here we are considering a more general level order traversal, where the levels correspond to vertices that are a certain distance away (in terms of number of edges) from a given vertex.

The new wrinkle in the algorithm below comes from the fact that we are working with a graph rather than a tree, and so we need to avoid cycles.

```

bft(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 an 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 u that we are searching from.

Note how the queue evolves over time, and what is the order in which the nodes are visited.

UNDIRECTED GRAPH		DIRECTED GRAPH	
-----		-----	
QUEUE q	TREE	QUEUE q	TREE
(snapshots)		(snapshots)	
a	a	a	a
cd	/ \	c	
df	c d	f	c
f		be	
be	f	e	f
e	/ \		/ \
	b e		b e
order visited:	acdfbe	order visited =	acfbe

Another Example

Here is another example, which better illustrates the difference between `dft` and `bft`. We suppose the graph is undirected.

GRAPH	ADJACENCY LIST	DFT	BFT
a - b - c	a - (b,d)	a - b - c	a - b - c
	b - (a,c,e)		
d - e - f	c - (b,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
	f - (c,e,i)		
	g - (d,h)		
	h - (e,g,i)		
	i - (f,h)		
	order visited:	abcfedghi	abdcegfhi