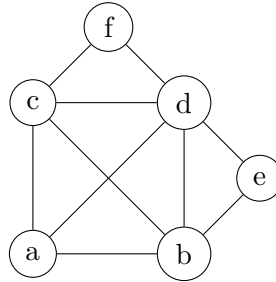## 6.4 Graph Algorithms

Similar to trees, you can do the same traversals.

6.4.1 Breadth First Traversal/Search (BFS)

Consider this graph to be a map of where you are and let's say you start at vertex A. When performing a BFS, you look at everything directly adjacent to you first, like you're scanning the areas side to side (hence breadth). This means vertices B and C, then D E F. In other words, we visit verticies that are 1 edge away from the starting point, then we visit verticies that are 2 edges away from the starting point. How do we do this?

Remember the BFS algorithm for trees? We can modify it slightly to work for graphs.

```
Q <- new Queue
enqueue start
while Q is not empty:
    x <- dequeue from Q
    for each adjacent unvisited vertex y to x
        enqueue y
        mark y as visited
    print y
```

All we did is replace root with the initial vertex and enqueue all adjacent unvisited vertices rather than left or right. We have to mark vertices as visited because graphs can have cycles and we do not want to visit the same location twice. Let's see the output of this algorithm on the example graph.

$$Q = [a] \qquad\qquad P = [] \qquad\qquad (6.1)$$
$$Q = [b, c] \qquad\qquad P = [a] \qquad\qquad (6.2)$$
$$Q = [c, e, d] \qquad\qquad P = [a, b] \qquad\qquad (6.3)$$
$$Q = [e, d, f] \qquad\qquad P = [a, b, c] \qquad\qquad (6.4)$$
$$Q = [d, f] \qquad\qquad P = [a, b, c, e] \qquad\qquad (6.5)$$
$$Q = [f] \qquad\qquad P = [a, b, c, e, d] \qquad\qquad (6.6)$$
$$Q = [] \qquad\qquad P = [a, b, c, e, d, f] \qquad\qquad (6.7)$$

And as we can see it works out. It's not a big deal if we visit E before or after D, however for the exams there will probably be some guidelines to ensure you get a consistent output like visiting adjacent nodes in a certain order.

6.4.2 Depth First Traversal/Search (DFS)

Again, similar to trees, we go as deep as we can into the graph. The important thing to note is that in this case deepness is not necessarily number of edges from the start node. Let's modify the graph to emphasize this point.
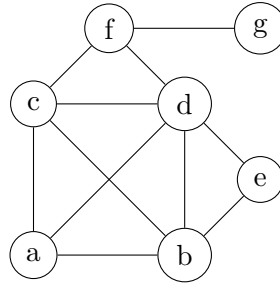
Just like the BFS and DFS algorithms for trees, the DFS algorithm for graphs can be obtained by replacing the queue with a stack and its respective operations. Or recursion.

$$S = [a] \qquad\qquad P = [] \tag{6.8}$$
$$S = [c, b] \qquad\qquad P = [a] \tag{6.9}$$
$$S = [f, d, b] \qquad\qquad P = [a, c] \tag{6.10}$$
$$S = [g, d, b] \qquad\qquad P = [a, c, f] \tag{6.11}$$
$$S = [d, b] \qquad\qquad P = [a, c, f, g] \tag{6.12}$$
$$S = [e, b] \qquad\qquad P = [a, c, f, g, d] \tag{6.13}$$
$$S = [b] \qquad\qquad P = [a, c, f, g, d, e] \tag{6.14}$$
$$S = [] \qquad\qquad P = [a, c, f, g, d, e, b] \tag{6.15}$$

Notice how the last thing that is visited is actually 1 edge away from where we started. Furthermore, depending on how the contents are printed out, we can find some pretty interesting arrangements.

6.4.3 Toplogical Sort

To continue off of the previous idea we touched upon, we can get a topological ordering of a graph by utilizing a depth first traversal.

What is a topological ordering? It's an ordering of vertices such that for every directed edge $\{u, v\}$ u comes before v. One example would be our course requirements. You guys couldn't take data structures before intro, and compilers before data structures. By performing a topological sort, we can figure out what the prerequisites for courses are, or a method of taking courses in an order so that prerequisites are fulilled before taking the actual course.

For this, we do a post order DFS (do something e.g. print after visiting adjacent nodes) then REVERSE the output.

Consider the following graph:
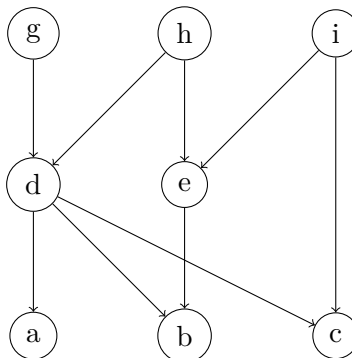
First we're doing a postordering which means print after everything, so instead of popping/dequeueing, we just peek/front. Or we can go the simple recursive way.

```
function dfs_helper(x)
    mark x as visited
    for each adjacent unvisited vertex y to x
        dfs_helper(y)
    print x

function topological_sort()
    for each vertex with no outgoing edges x
        if x is unvisited
            dfs(x)
```

After everything, we will print out the following: $[a, b, c, d, g, e, h, i]$. But remember, we wanted it reversed, so it would come out as $[i, h, e, g, d, c, b, a]$. One way we can do this is to push to a stack instead of a print and at the end of dfs, we pop and print from the stack until it is empty. As such, a quick observation is all that's needed to confirm that this is a proper topological ordering.

- d completed before a? check

- d completed before b? check

- d completed before c? check

- d completed before c? check

- g and h completed before d? check

- h and i completed before e? check

6.4.4 Djikstra's Algorithm

Naturally when thinking about graphs and maps, you might want to find the shortest path between points. Djikstra's (check spelling) algorithm is one way to do this. Specifically, Djikstra's algorithm is able to determine the shortest path from one vertex to all other vertices as long as there are no negative weights.

I find the simplest way to memorize this algorithm is to remember that it is greedy. How is it greedy? You always take the next shortest path possible that takes you to an unvisited vertex. Let's look at an example
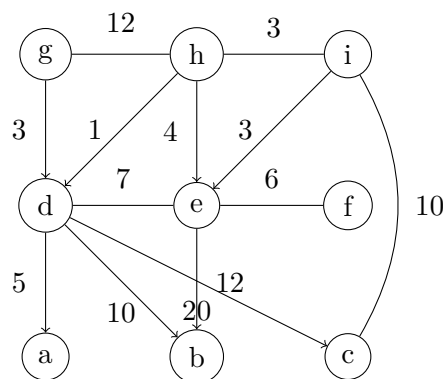
Consider this beast of a graph (sorry I'm not good at positioning the weights). First, notice this is a mixed graph because it has both directed and undirected edges. With all our initial observations done, let's try to find the shortest path from e to all other vertices.

We start from e and right away we know that the shortest path from e to e is 0. Now we look at all adjacent verticies for all vertices that we have visited so far. We visited e so we can reach f, d, b for 6, 7, and 20 respectively. The shortest path is 6, so we visit f and now we know the shortest path to f is 6. Now we do the same thing; we can visit d, b for 7 and 20 so we go to d. Now we know the shortest path to d is 7. And one more time before I show our current progress. Now that we have visited e, f, d and know the shortest paths between e and those vertices, we can now visit b and now a and c by going through d. Furthermore, notice that we can also visit b by going through d which is actually a shorter path! Let's see a table that shows our current progress:

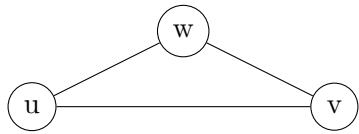| _	_	e	f	d
a				12
b	20	20	20	17
c				19
d	7	7	**7**	
e	**0**			
f	6	**6**		

In our first column, we knew the distances from e to it's adjacent vertices. The blank cells are infinities unless there are previous cells. The shortest distance in that column is 0 for e so we know the shortest path from e to e is 0. We take note at the top of the next column and update the paths to all reachable vertices. Then we pick the next shortest path which is 6 for f. One important thing to note is that we actually found a shorter path to b by going through f. So we just fill out the table until there are no more paths to look at.

| _	_	e	f	d	a	b	c	i	h	g
a				**12**						
b	20	20	20	17	**17**					
c				19	19	**19**				
d	7	7	**7**							
e	**0**									
f	6	**6**								
g									**44**	
h								**32**		
i							**29**			

And finally we have the shortest distances from e to everything else (they're the bold numbers). We can also use this table to figure out the actual path, not just the distance. We know the distance of the shortest path from e to g is 44, but what is the actual path? Look on the table where the bolded number first appeared in its row. Notice at the top of that column is h. This means h was visited before g. We repeat this easily and end up with e, d, c, i, h, g.

6.4.5 Floyd-Warshall Algorithm

The Floyd Warshall Algorithm is an algorithm for finding the shortest paths between all pairs of vertices. It relies on one idea which is that for every path between vertices u and v, there may be a path that goes through w such that

$$dist(\{u, w\}) + (\{w, v\}) < dist(\{u, v\})$$

Consider the following graph:
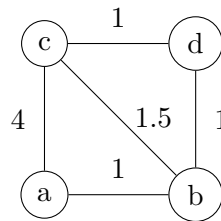
We start out by filling a table with the known edges and distances along with another table that lists the initial vertex in the path.

_	a	b	c	d
a	0	1	4	
b	1	0	1.5	1
c	4	1.5	0	1
d		1	1	0

_	a	b	c	d
a	a	a	a	
b	b	b	b	b
c	c	c	c	c
d		d	d	d

And all we do is loop through all the possible vertices and treat them as potential w's. For our first iteration, we will treat a as w. u and v will be the column and row of the path. Let's take an example: b as u and c as v.
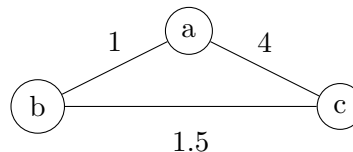
As we can see, the distance is not shorter. However, we will check all pairs (u, v) along with a as w. One example with this graph where it will be replaced is when b is w, and u and v are a and d respectively; let's take a look.
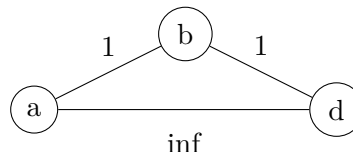
Naturally 2 is less than infinity so we would update our table with the value 2. We would also come across the same case when d is u and a is v so I'll go ahead and insert 2 in that spot too. We would also update our second table with whatever value w was, in this case b

_	a	b	c	d
a	0	1	4	**2**
b	1	0	1.5	1
c	4	1.5	0	1
d	**2**	1	1	0

_	a	b	c	d
a	a	a	a	**b**
b	b	b	b	b
c	c	c	c	c
d	**b**	d	d	d

We'll loop through all possible u, v, w triplets and end up with the distances of shortest paths between all pairs of vertices in addition to a table that will help us figure out what the path actually is.

6.4.6 Minimum Spanning Trees (Prim's/Kruskal's Algorithm)

A minimum spanning tree is a subset of all edges of the graph that form a tree such that the total weight of the edges is minimized.

There are two simple algorithms that do this:
Prim's algorithm: take the shortest edge that adds a new node to the tree. Kruskal's algorithm: take the shortest edge that connects two unconnected trees.

These are very simple and should pose no problem.