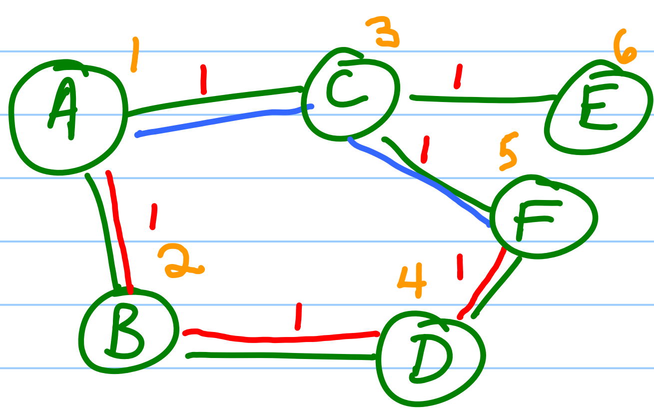
Breadth-First Search

Note Title 12/2/2007

$G = (V, E)$

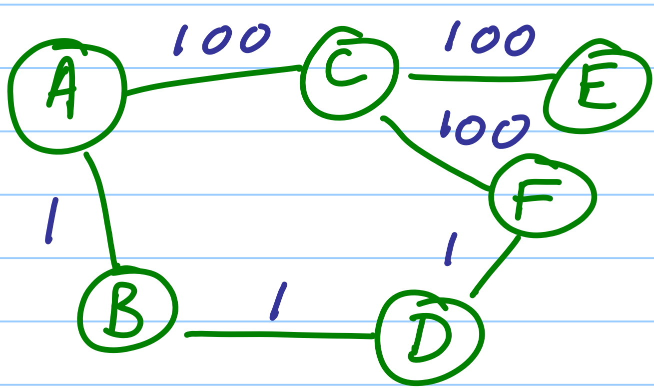
- Given a graph, one way to visit every vertex in that graph is via a breadth-first search.
  - Select a starting point.
  - Visit all vertices that are "one jump" away from it.
  - Visit all vertices that are "two jumps" away from it.



Which node can you get to in 3 jumps?

BFS finds shortest path, eg., from A to F

But what happens if we add weights?



Now what is the shortest path from A to F?

BFS For Shortest Path

- A simple problem that can be solved using this general technique is that of finding the shortest path between two vertices in an undirected and unweighted graph.

If unweighted, just count the "hops".
(Doesn't work for weighted graph.) ← Expedia

- If the graph is not connected, what happens?

We have an easy-to-code method for determining if connected. (We'll see shortly.)

Unweighted
Undirected
Connected

Shortest Path via BFS

Starting at vertex s∈V generate an array of distances from s called dist[] such that for all v∈V, dist[v]=length of shortest path from s to v. |V|

dist[s]=0

We will also create a predecessor array of the last vertex we were at before getting to the end of the path from s to v |V|

for all v∈V, pred[v]="one step back"

pred[s]=none

With just these two arrays, we will be able to reconstruct any shortest part request from s to some vertex.

This is because any **sub-path** of the optimal path must also be an optimal **path between its own endpoints**.

If it weren't, then we could have replaced it and gotten a shorter **overall** path.

Basic Pseudo Code

Start at s.

For each neighbor **v** of s

    dist[**v**]=1
    pred[**v**]=s.

Move outwards from each neighbor you've seen and set the next "ripple" out as "+1" of the current distance, and set pred[] appropriately.

Need a way to make sure we don't end up in cycles!

Avoiding Cycles

We will assign a color to each vertex based on the following rules:

- white = not seen yet at all
- gray = seen but not processed yet (added to queue)
- black = processed (removed from queue)

We will create a queue of gray vertices, and will never add any vertex to the queue more than once.

When we are done processing a vertex (ie: we have touched all its neighbors) we go back to the queue to get the next vertex to process.

More Detailed Pseudo-Code

```
BFS (Graph G, vertex s) {
int size = G.getVertexCount;
int dist[] = new int[size];
vertex pred[] = new int[size];
Queue Q= new Queue<vertex>;
Colors state[] = new Colors[size];
    for each v in G.V {
        state[v]=white; dist[v]=infinity; pred[v]=none;
    }

    state[s]=gray;  dist[s]=0;  pred[s]=none;
    Q.add(s);

    while (!Q.empty()){                      R1
        u=Q.remove();                        R1
        for each unvisited v in G.Adj(u) {   R1
            state[v]=gray;                   R1
            dist[v]=dist[u]+1;               R2 ←
            pred[v]=u;                       R2 ←
            Q.add(v);                        R2 ←
        }
        state[u]=black;
    }
}
```

What's the Runtime? ~~T(n)~~ T(G)

- Each vertex gets enqueued at most one time, so each is processed at most one time.
  - Let's write this up using a summation to represent the processing of all of the vertices… $|V|$ $G = (V, E)$

$$T(G) = O\left(|V| + \sum_{v \in V} (1 + \deg(v))\right)$$

- $|V|$: initialization of state, dist, pred
- $\sum_{v \in V}$: while loop (while queue not empty)
- 1: work outside for loop (color black)
- $\deg(v)$: for loop (go through each new vertex)

simplify

while loop's +1 cost

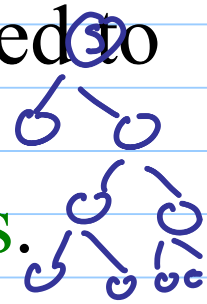
$$|V| + |V| + \underbrace{\sum_{v \in V} \deg(v)}_{2|E|}$$

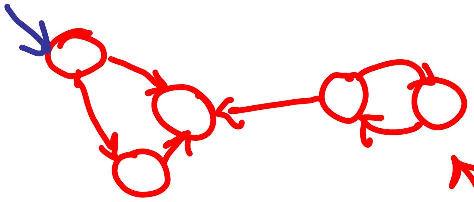
$$\text{So } 2|V| + 2|E| \in O(|V| + |E|)$$

What else does BFS give us?

- It allows us to organize the entire graph as "ripples" away from a central point.
  - This could be useful if we could restate other questions within this framework.

- Our predecessor array could be used to create a tree rooted at source s of vertices that can be reached from s.
  - This is often called a breadth-first tree.
  - If we could phrase a problem as a traversal of this tree…





Could you use BFS to...

- Detect whether a graph has any cycles?

  Undirected: YES - if you try to enqueue a gray/black vertex

  Runtime? Trivially $O(|V| + |E|)$

  but

  An acyclic graph cannot have more than $|V| - 1$ edges

  So $O(|V|)$ really

  Can get trapped if directed, even though there might be a cycle. (Must be connected in order to guarantee that it will find a cycle.)

- Determine whether every vertex is reachable from a particular vertex?

  YES - $O(\underbrace{|V| + |E|}_{BFS} + |V|)$ ← make sure every vertex has a d[] value

  Look at color or distance. If white or distance is ∞, not reachable.

- Find the longest simple path through the graph between two vertices?

  No!

  NP Complete.

Depth-First Search (DFS)

Implementation:

- Change queue to stack

- Rewrite as recursive algorithm

Use of DFS:

- Can be used to determine what vertices are reachable in $O(|E| + |V|)$ time.

DFS on Directed Graph w/ "Timing" Info

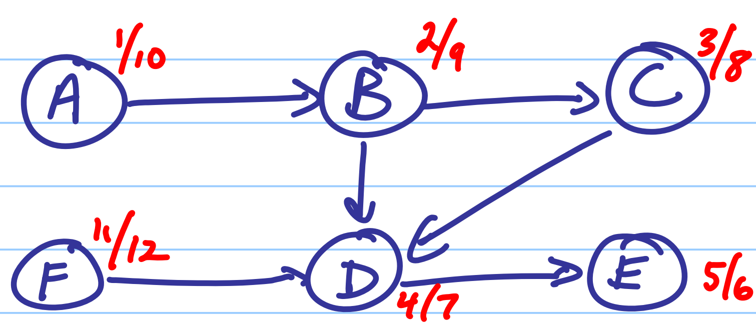
- Add more arrays and store information such as when (in terms of a continuously advancing ticker) each vertex is first visited and finally processed.

Running Counter of who we visited + when we first encountered them + when we last encountered them.

- Even in a connected graph, we might end up having to build a forest of trees to give every vertex a set of times.

  – After doing a DFS from a given starting point, if there are vertices with no times, choose one of them and continue.

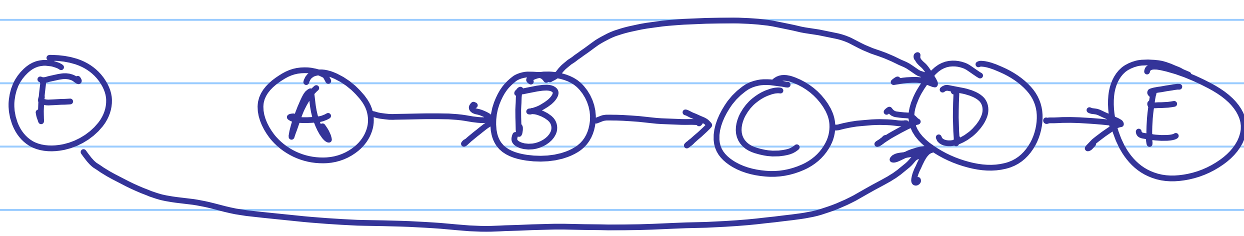
⇓



FABCDE

Arrows always point to the right.

⇓



Topological Sort of a Digraph

(Good for Course prereq structures)

NOTE: This only works if there are no cycles, since if there are cycles there isn't the notion of a sorted order.

Imagine a graph as beads where the edges are strings of equal length connecting ordered pairs of beads.

You want to arrange the beads so that all edges point left-to-right.

How can you use a DFS with "timing" info to accomplish this?