More Graphs!

Data Structures and Algorithms

Announcements

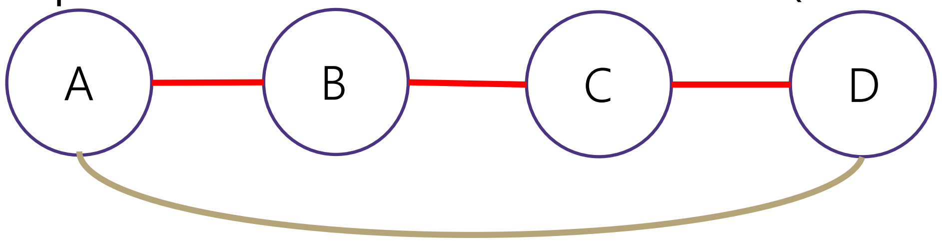
Project 2 Due Tonight!

HW 5 Due Wednesday

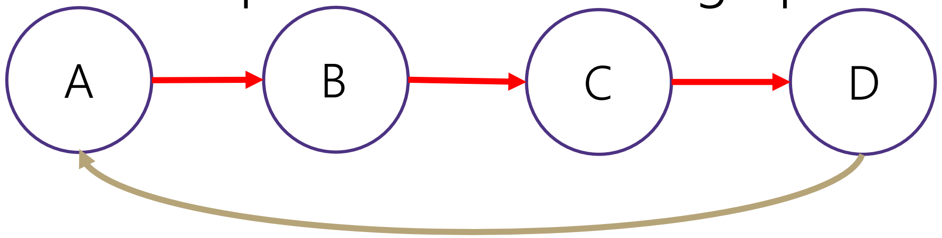
Project 3 released today – please check who your partner is and contact them TODAY. If you do not get a response by Sunday, contact the course staff for reassignment.

Review

Path – A sequence of connected vertices (sometimes called a “walk”)



(Directed) Path – A path in a directed graph must follow the direction of the edges

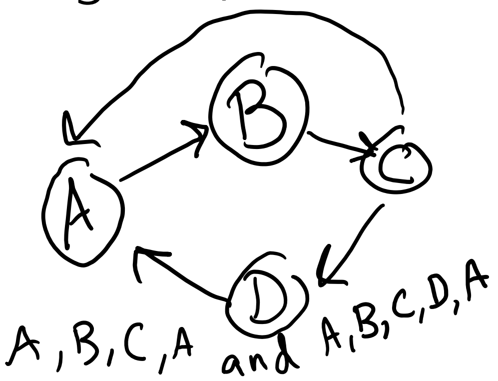


Path Length – The number of edges in a path (unweighted), sum of edges (weighted)

- (A,B,C,D) has length 3.

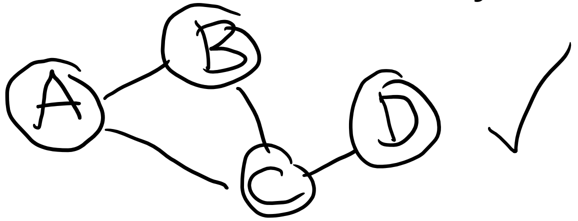
Simple Path – A path that doesn’t repeat vertices (except maybe first=last)

Cycle – A path that starts and ends at the same vertex (of length at least 1)



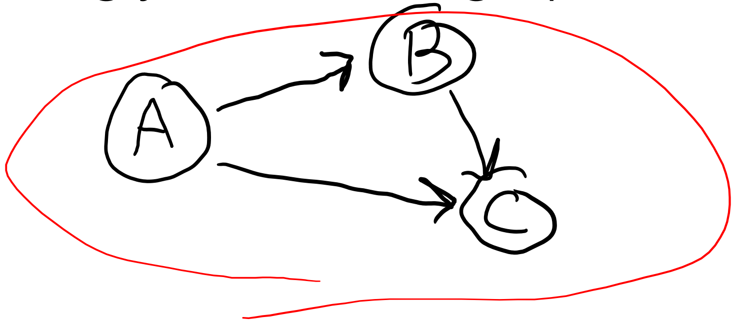
Review: Connected Components

Connected Graph – A graph that has a path from every vertex to every other vertex (i.e. every vertex is **reachable** from every other vertex.



Strongly Connected Graph – A directed graph that is connected (note the direction of the edges!)

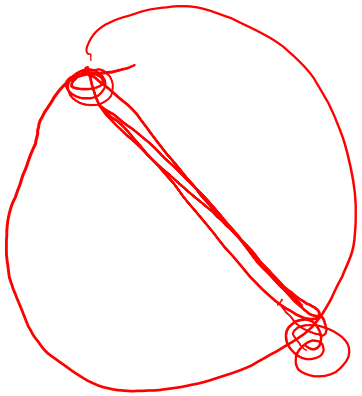
Weakly Connected Graph – A directed graph that is connected when interpreted as undirected. (Note all strongly connected graphs are also weakly connected)



Paths and Reachability

Very common questions:

- Is there a path between two vertices? (Can I drive from Seattle to LA?)
- What is the length of the shortest path between two vertices? (How long will it take?)
- Can every vertex reach every other on a short path?
  - 7 degrees of Kevin Bacon
  - Length of the longest shortest path is the "diameter" of a graph



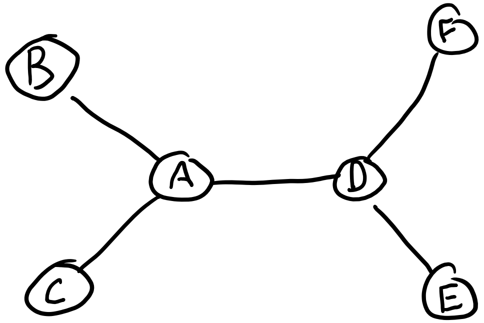
Less common, but still important:

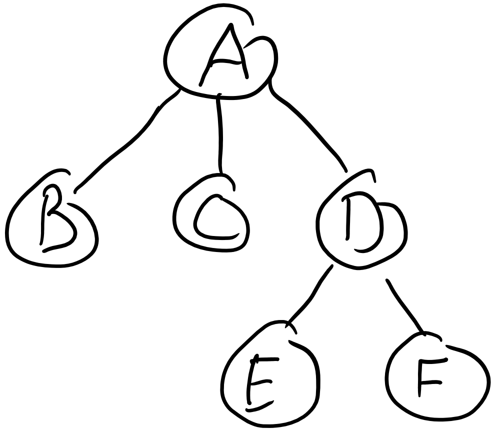
What is the longest path in a graph?

- In general a "hard"™ problem

Trees

A **tree** is a connected, acyclic graph.





In a ~~tree there exist~~ **exactly one** path between every pair of vertices.

A graph consisting of several disconnected trees is called a **forest**.

The trees we have seen so far have been **rooted trees** – interpret one vertex as the **root**, and its neighbors are now **children**, and the root of their own **subtrees.**

How many edges does a tree with **n** vertices have? n - 1

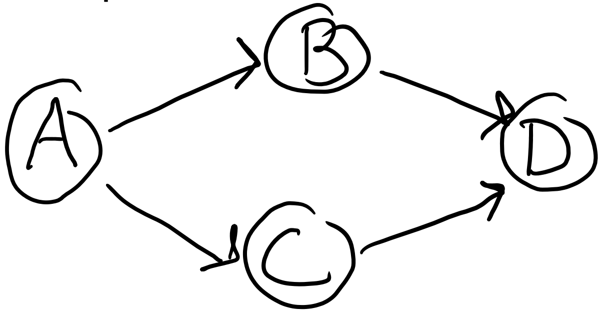
DAGs

DAG stands for **Directed, Acyclic, Graph**

This is the directed graph analog of a forest.

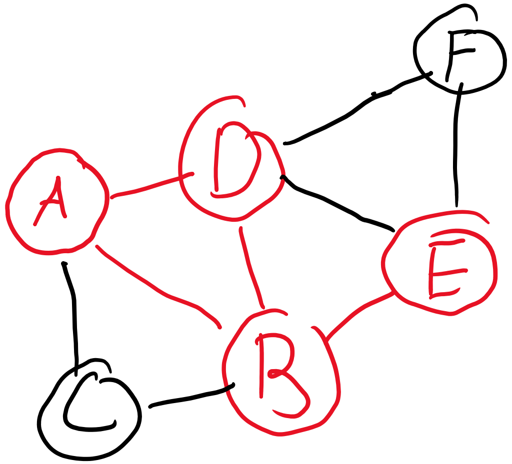
The trees we have made so far in this class have been implemented as weakly connected DAGs.

Can be used to represent **dependencies**: i.e. A must be completed before either B or C, and both B and C must be completed before D. Scheduling these tasks is called **topological sort.**



Subgraphs

Take a graph, and delete vertices and edges so you still have a graph.



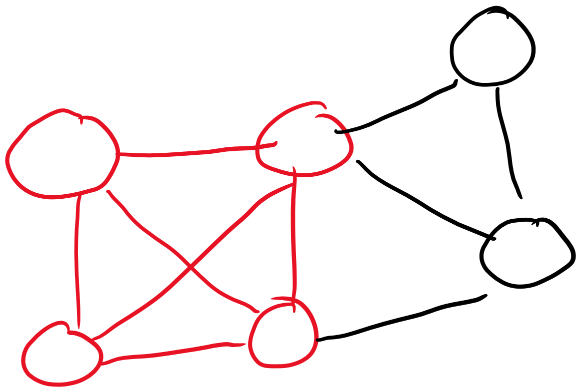
The remaining red vertices and edges form a subgraph.

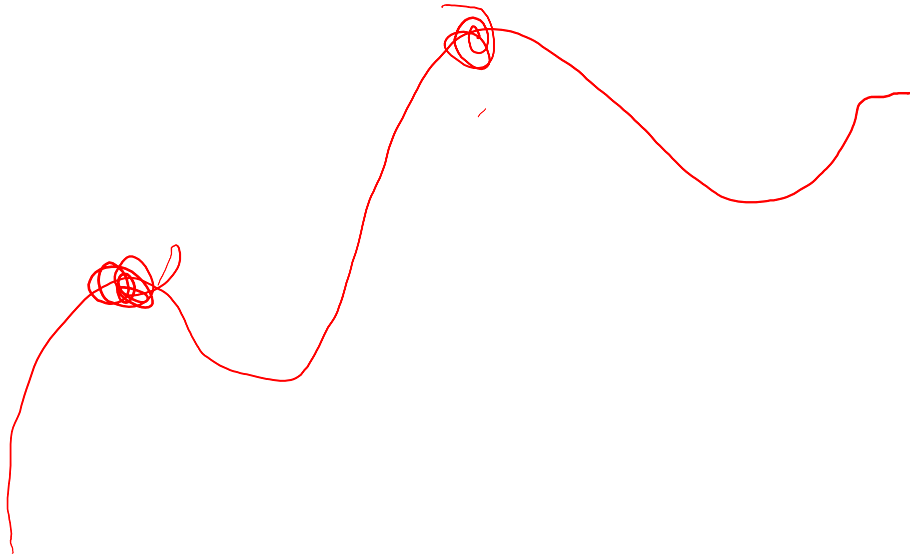
Formally: $G' = (V', E') \subseteq G = (V, E) \Leftrightarrow V' \subseteq V$ and $E' \subseteq E$ *and* G' *is a graph*

Interesting Subgraphs

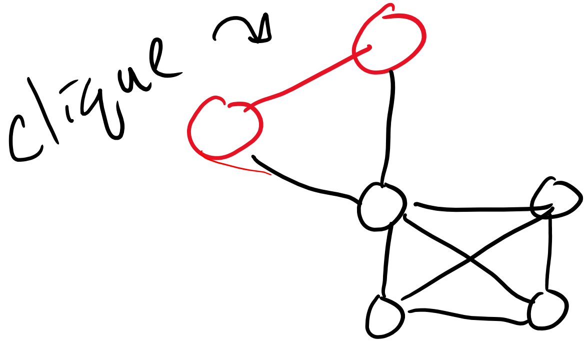
Cliques

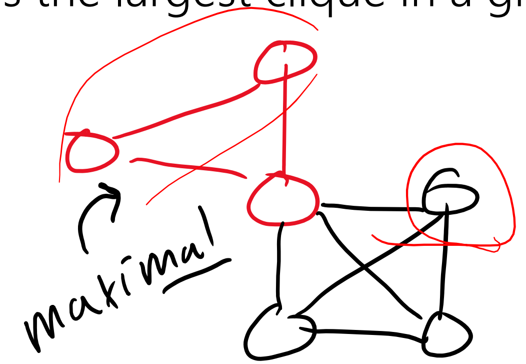
- A **clique** is a complete subgraph

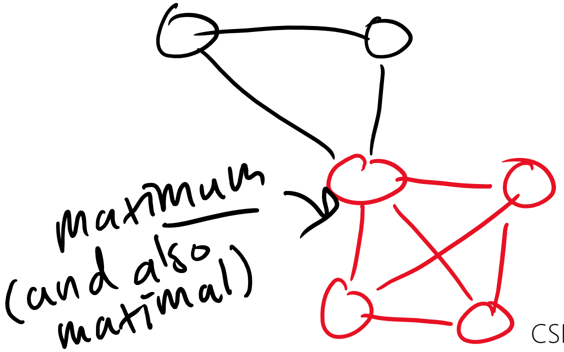




- A **maximal** clique is a clique that you could not add any more vertices to and still have a clique. This is distinct from the **maximum** clique, which is the largest clique in a graph.





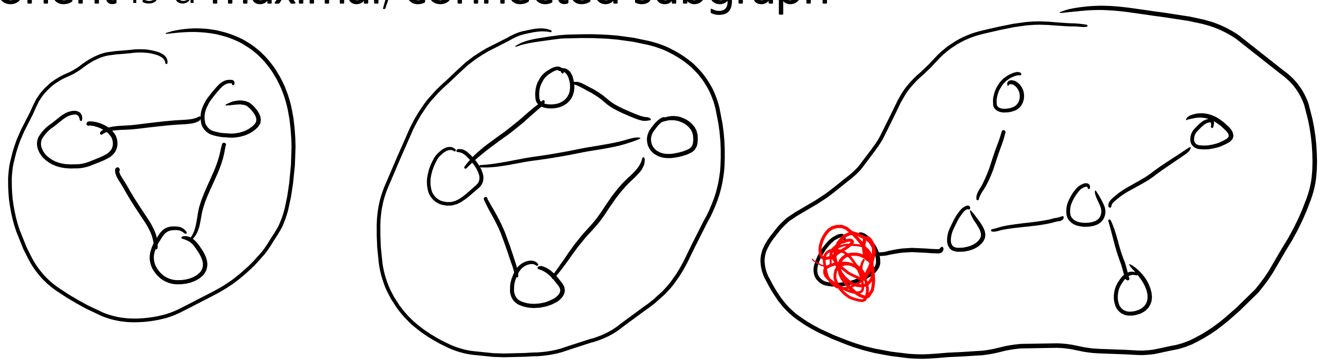


Interesting Subgraphs

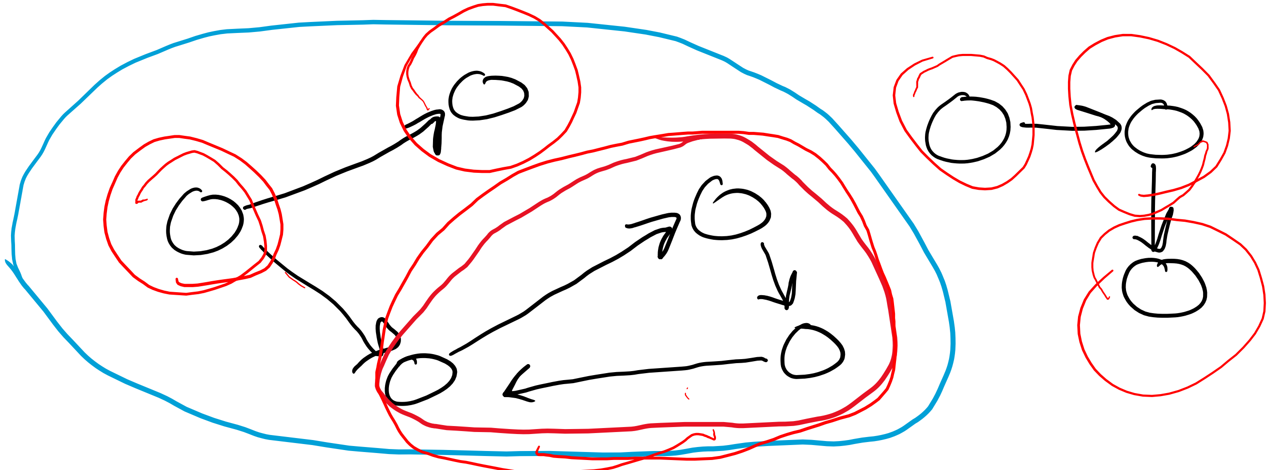
can't insert another vertex

Connected Components

- A **connected component** is a **maximal**, **connected subgraph**

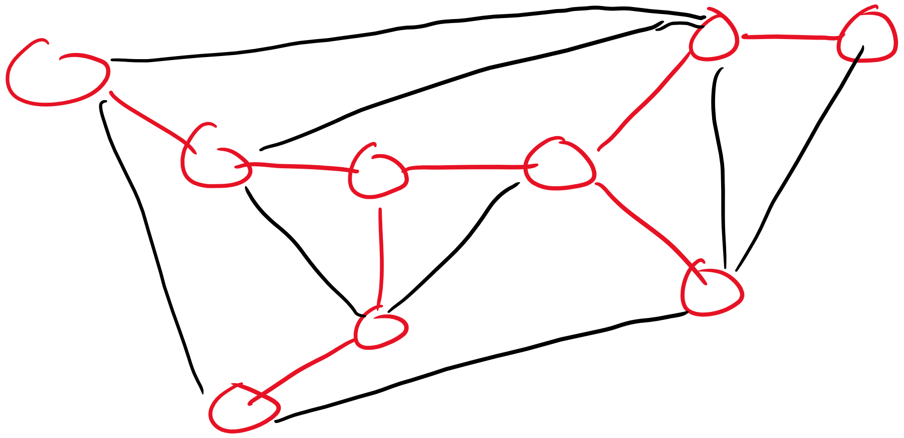


- In **directed** graphs, you have two kinds: strongly connected, and weakly connected:



Interesting Subgraphs

A **Spanning Tree** is a subgraph that is both a **tree** and includes **every vertex** (it **spans** the graph).



Every **connected** graph has at least one spanning tree.

An important problem is finding the **minimum spanning tree**. We will learn 2 algorithms for this. It is useful for optimization tasks (e.g. minimum cost to build a road network).

Graph Algorithms

Traversing a Graph

In all previous data structures:

1. Start at first element
2. Move to next element
3. Repeat until end of elements

For graphs – Where do we start? How do we decide where to go next? When do we end?

1. Pick any vertex to start, mark it "visited"
2. Put all neighbors of first vertex in a "to be visited" collection
3. Move onto next vertex in "to be visited" collection
4. Mark vertex "visited"
5. Put all unvisited neighbors in "to be visited"
6. Move onto next vertex in "to be visited" collection
7. Repeat…

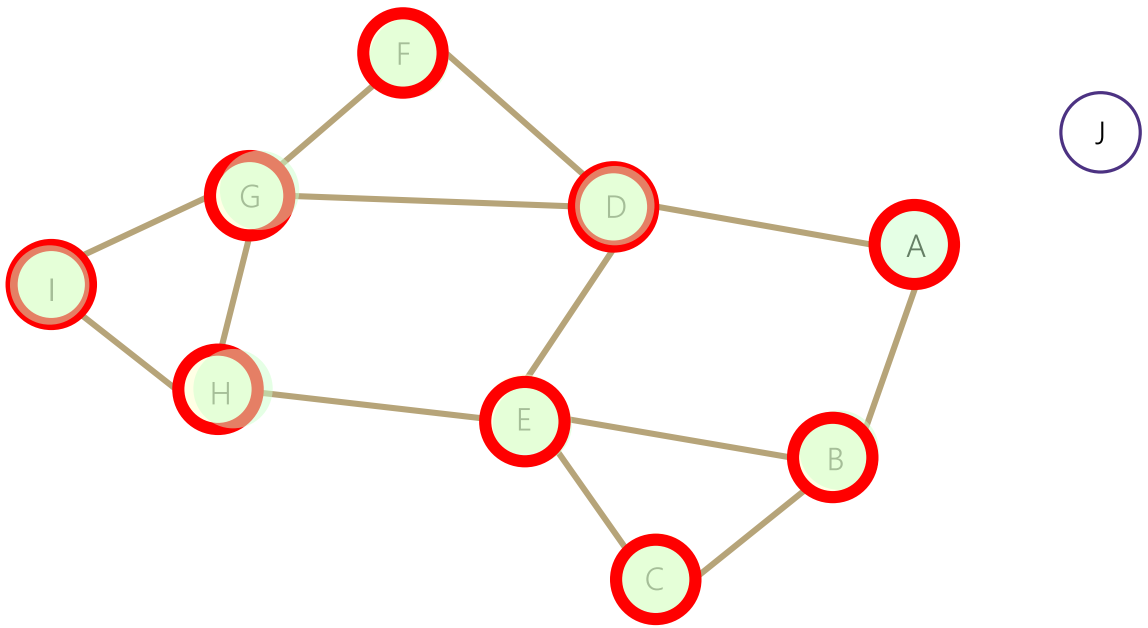
Breadth First Search

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as visited
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (V : current.neighbors())
      if (v is not visited)
        toVisit.enqueue(v)
        mark v as visited
    finished.add(current)
```

Current node: I

Queue: B D E C F G H I

Finished: A B D E C F G H I

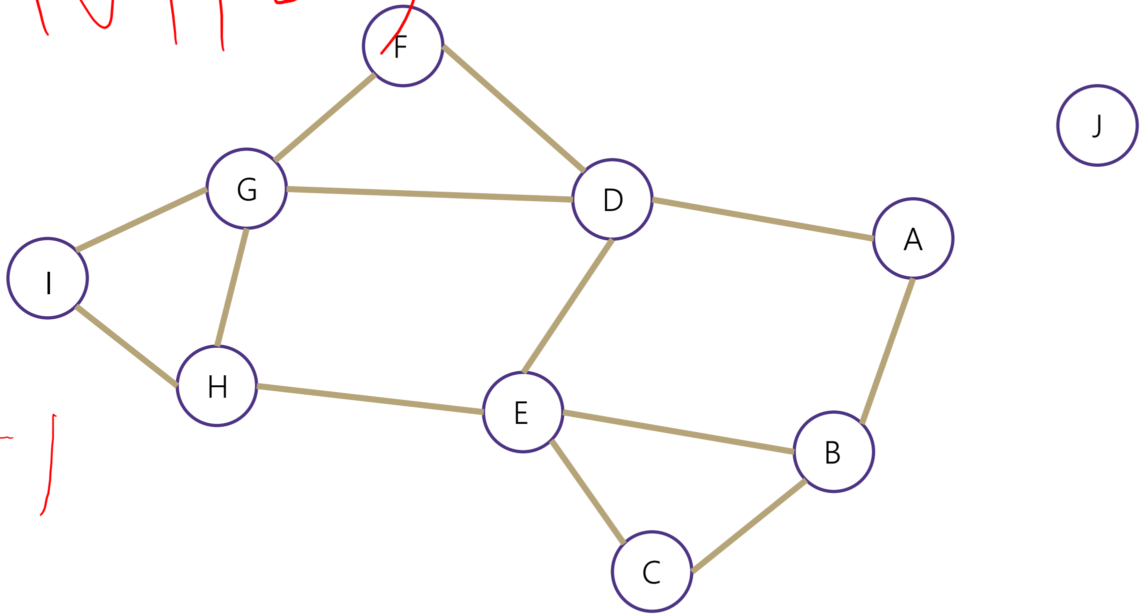


Breadth First Search Analysis

```
search(graph)
   toVisit.enqueue(first vertex)
   mark first vertex as visited
   while(toVisit is not empty)
      current = toVisit.dequeue()
      for (V : current.neighbors())
         if (v is not visited)
            toVisit.enqueue(v)
            mark v as visited
      finished.add(current)
```

O(|V| + |E|)

|E|



Visited: A B D E C F G H I

How many times do you visit each node? 1 time each

How many times do you traverse each edge? Max 2 times each

- Putting them into toVisit
- Checking if they're visited

Runtime? O(V + 2E) = O(V + E) "graph linear"

Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing "frontier" movement across graph

Can you move in a different pattern? Can you use a different data structure?

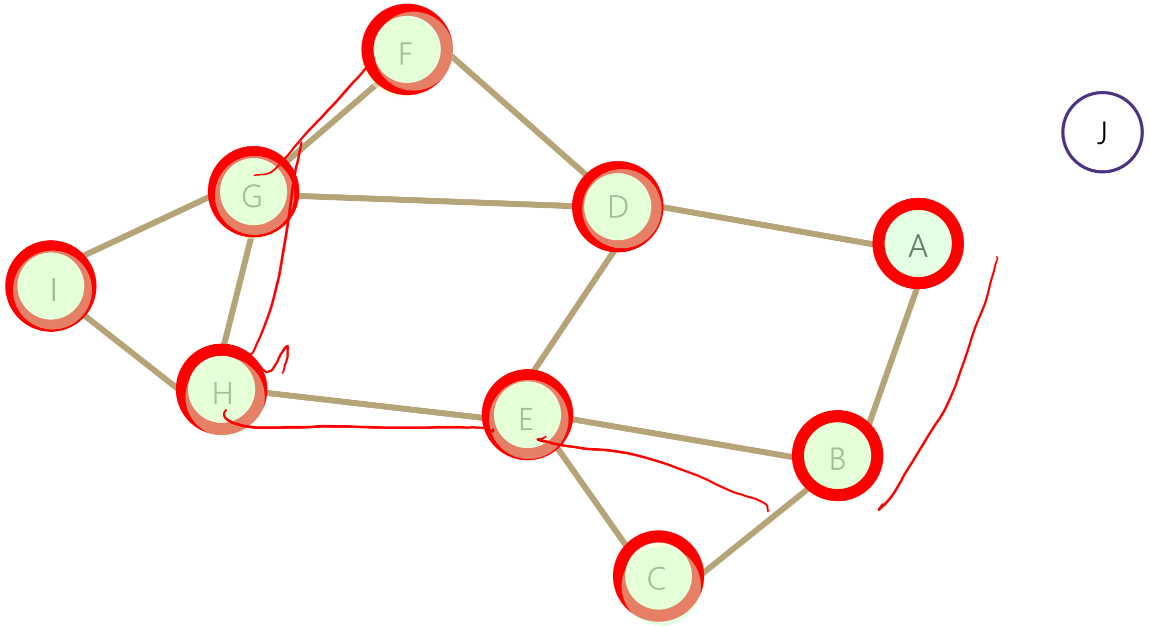
What if you used a stack instead?

```
bfs(graph)
    toVisit.enqueue(first vertex)
    mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (V : current.neighbors())
            if (v is not visited)
                toVisit.enqueue(v)
                mark v as visited
        finished.add(current)
```

```
dfs(graph)
    toVisit.push(first vertex)
    mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.pop()
        for (V : current.neighbors())
            if (V is not visited)
                toVisit.push(v)
                mark v as visited
        finished.add(current)
```

Depth First Search

```
dfs(graph)
   toVisit.push(first vertex)
   mark first vertex as visited
   while(toVisit is not empty)
      current = toVisit.pop()
      for (V : current.neighbors())
         if (V is not visited)
            toVisit.push(v)
            mark v as visited
      finished.add(current)
```



Current node: D

Stack: D CB EI HG

Finished: A B E H G F I C D

How many times do you visit each node? 1 time each

How many times do you traverse each edge? Max 2 times each

- Putting them into toVisit
- Checking if they're visited

Runtime? O(V + 2E) = O(V + E) "graph linear"

Recursive DFS

DFS can be implemented using recursion as well:

```
dfs(vertex, visited)
    visited[vertex] = true
    for each neighbor of vertex:
        if not visited[neighbor]:
            visited = dfs(neighbor, visited)
return visited
```

What about BFS? Not really. Function calls act like a stack, so DFS works well, but not DFS.