CSE 332: Data Structures & Parallelism
Lecture 16: Graph Traversals

Ruth Anderson

Autumn 2025

Administrative

EX06 – On Sorting: Due Fri Nov 7

EX07 – On Graphs: programming, coming soon

Resources!

- **Conceptual Office Hours**: 11:30 Tues (Connor) and 11:30 Wed (Samarth) both in CSE1 006. A space to ask about **course content and topics only** *as opposed to direct help with exercises*.
- 1-on-1 Meeting Requests - Request a meeting with a staff member if you cannot make it to regularly scheduled office hours, or feel like you have an issue that requires a more in depth discussion.

What do we do with graphs

So many things!

- That's why we said graphs are more general than a single ADT---they don't have a standard set of operations.

As a starting point---how could we process the entire graph? Examine every vertex and every edge?

Called a "search" of the graph or a "traversal"

Two algorithms (with different purposes)

BFS

Start somewhere…

For every vertex (in some order)

Do whatever you want on that vertex

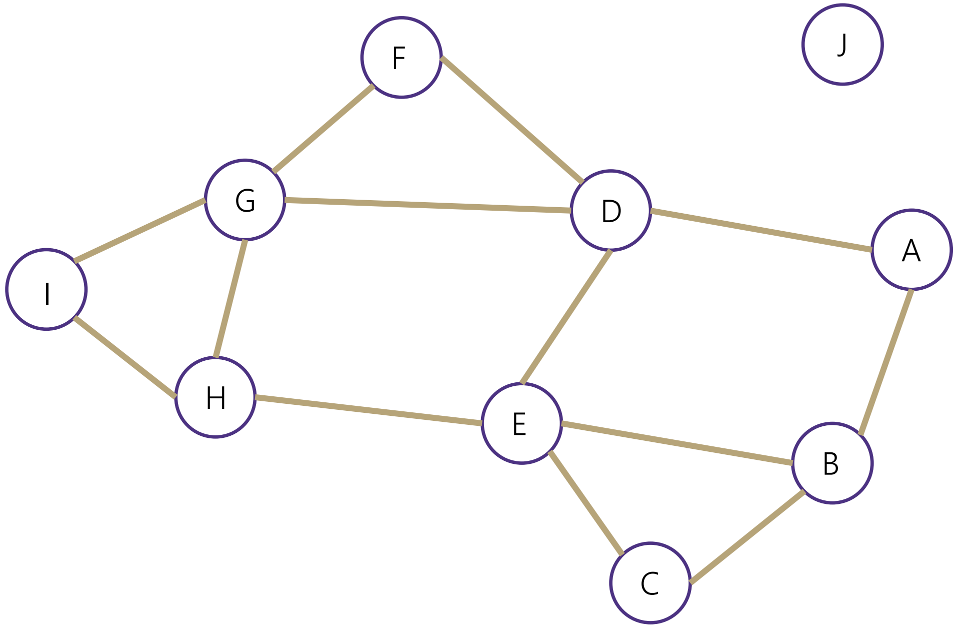
- Sometimes, record some information, store something in there, etc.
- At least, we'll mark it as having been "visited"

You need to process all of its neighbors…store them in some data structure to process them. Then back to the top of the loop.

If you use a Queue for your storage structure, you get BFS.

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:

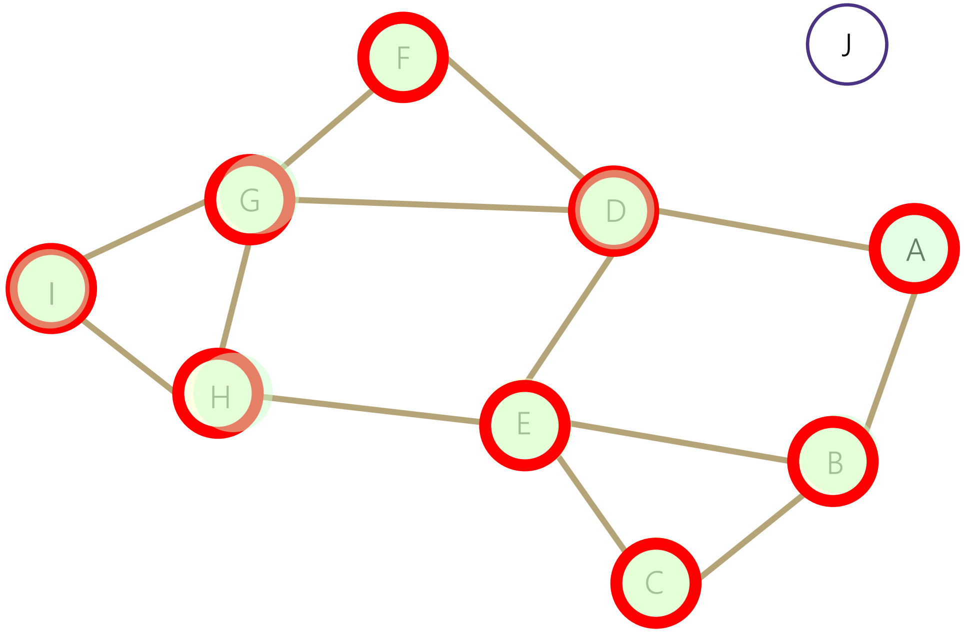
Queue:

Finished:

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)
```

V times



Current node: I

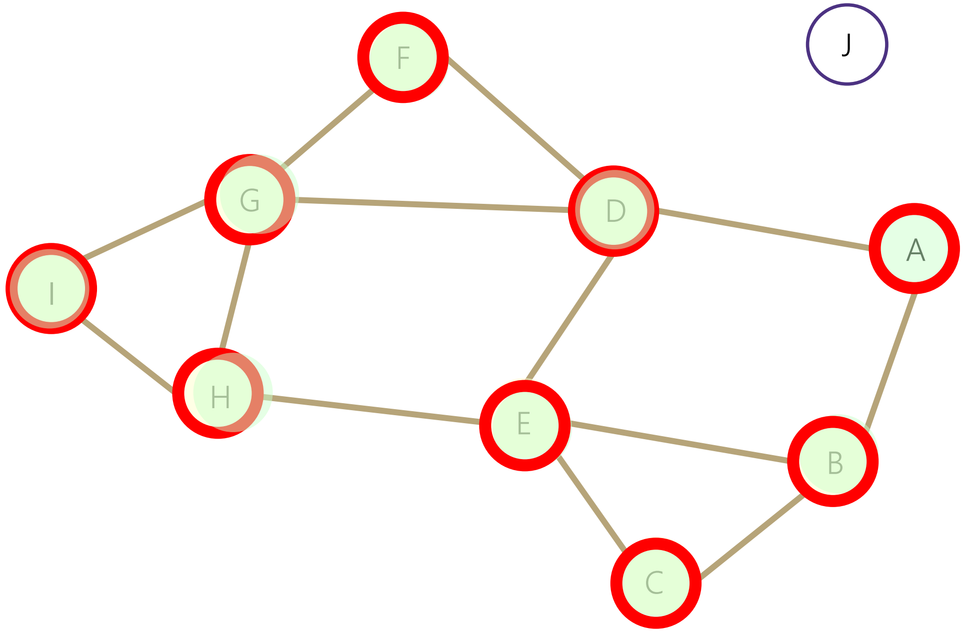
Queue: B D E C F G H I

Finished: A B D E C F G H I

$O(V+E)$

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)
```



What's the running time of this algorithm?

We visit each vertex at most twice, and each edge at most once: $O(|V| + |E|)$

Depth First Search (DFS)

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

Gives us a growing "frontier" movement across graph

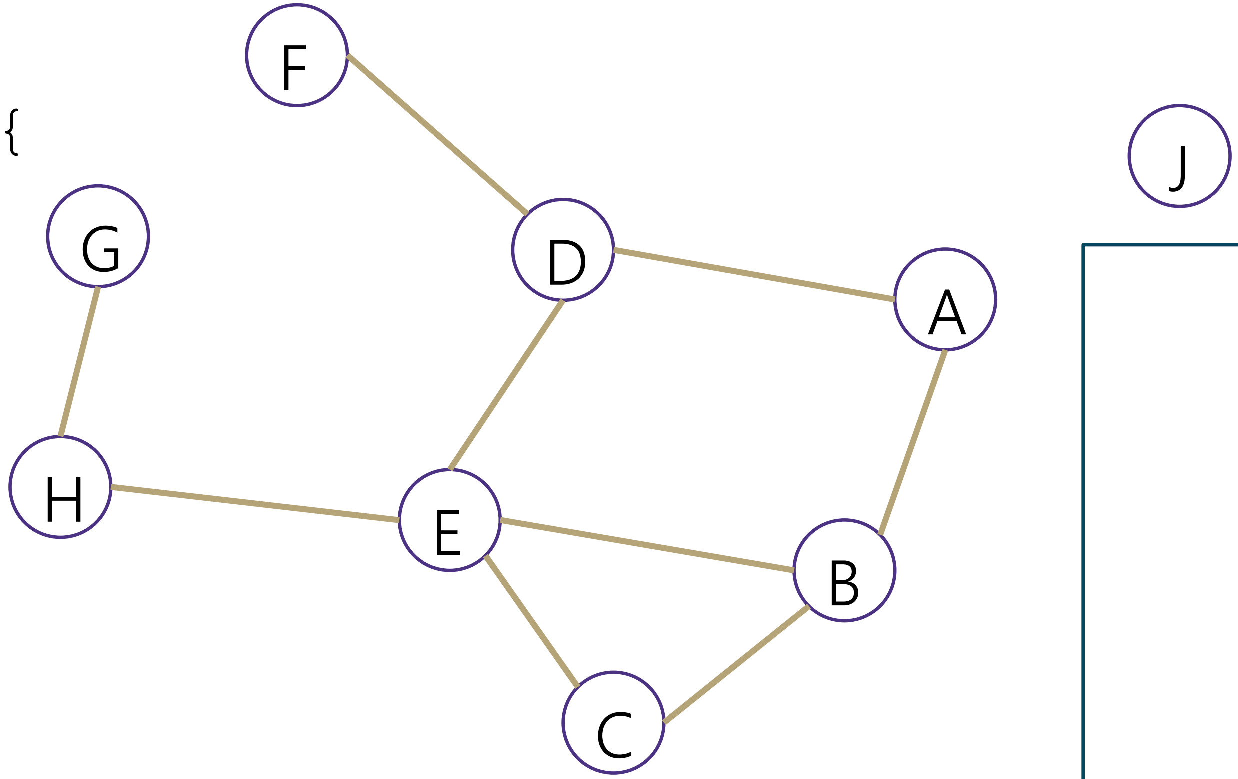
Can you move in a different pattern? What if you used a stack instead?

```
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, curr)
   mark curr as visited
   for(v : curr.neighbors()){
      if(v is not visited){
         dfs(graph, v)
      }
   }
   mark curr as "done"
```

Depth First Search

```
dfs(graph, curr)
    mark curr as visited
    for(v : curr.neighbors()){
        if(v is not visited){
            dfs(graph, v)
        }
    }
    mark curr as "done"
```

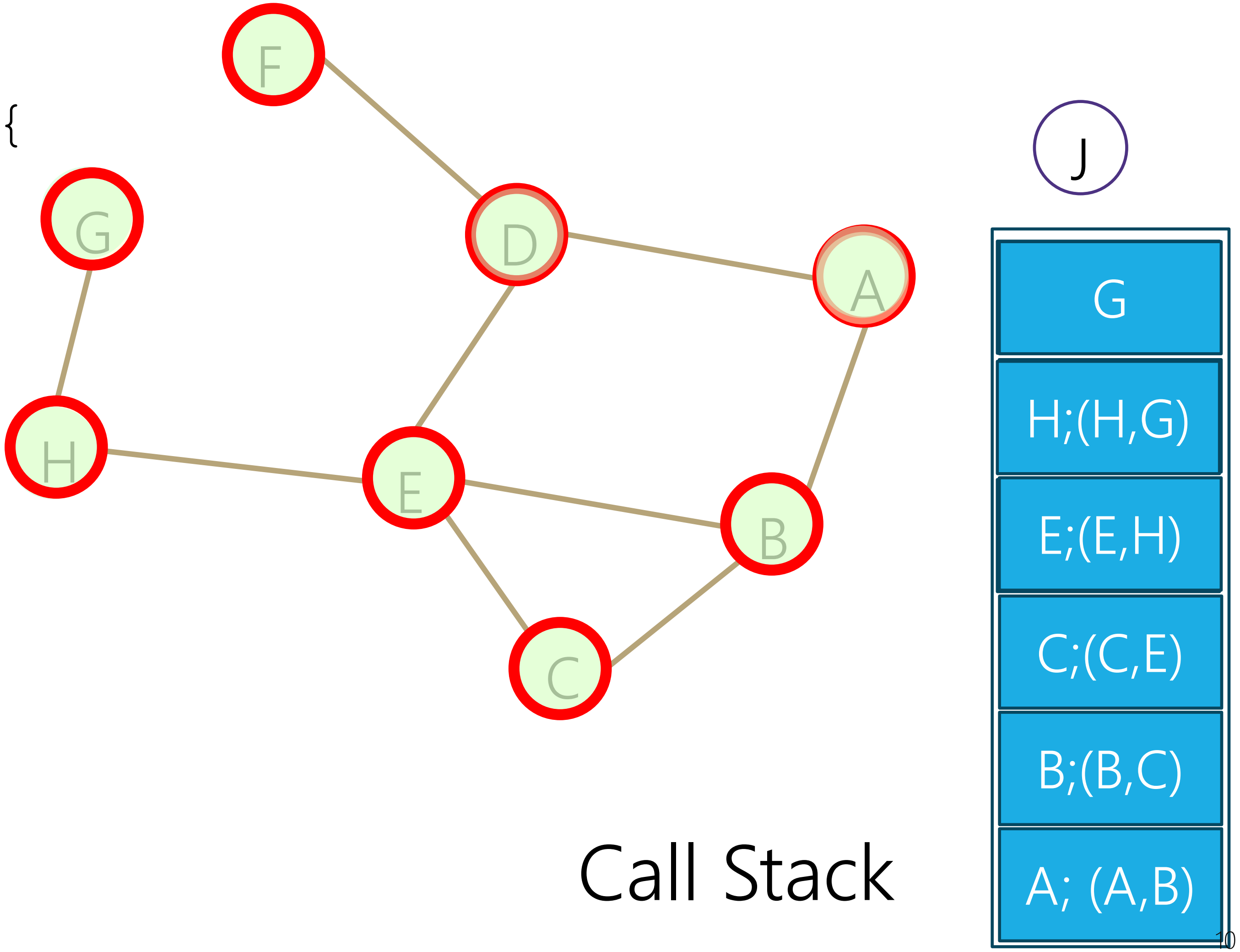


Finished :

Call Stack

Depth First Search

```
dfs(graph, curr)
    mark curr as visited
    for(v : curr.neighbors()){
        if(v is not visited){
            dfs(graph, v)
        }
    }
    mark curr as "done"
```



Finished: F D G H E C B A

DFS

Running time?

- Same as BFS: $\Theta(|V| + |E|)$

You can rewrite DFS to be an iterative method (that explicitly uses a stack data structure). Use that in place of the call stack.

Getting the details right is actually pretty annoying/subtle.

Next: Using BFS, DFS and other algorithms to solve problems!

DFS for applications

Applications for DFS (and BFS) are often:

Run [D/B]FS, and do some extra bookkeeping.

- Many applications work (easily) with only one ordering.

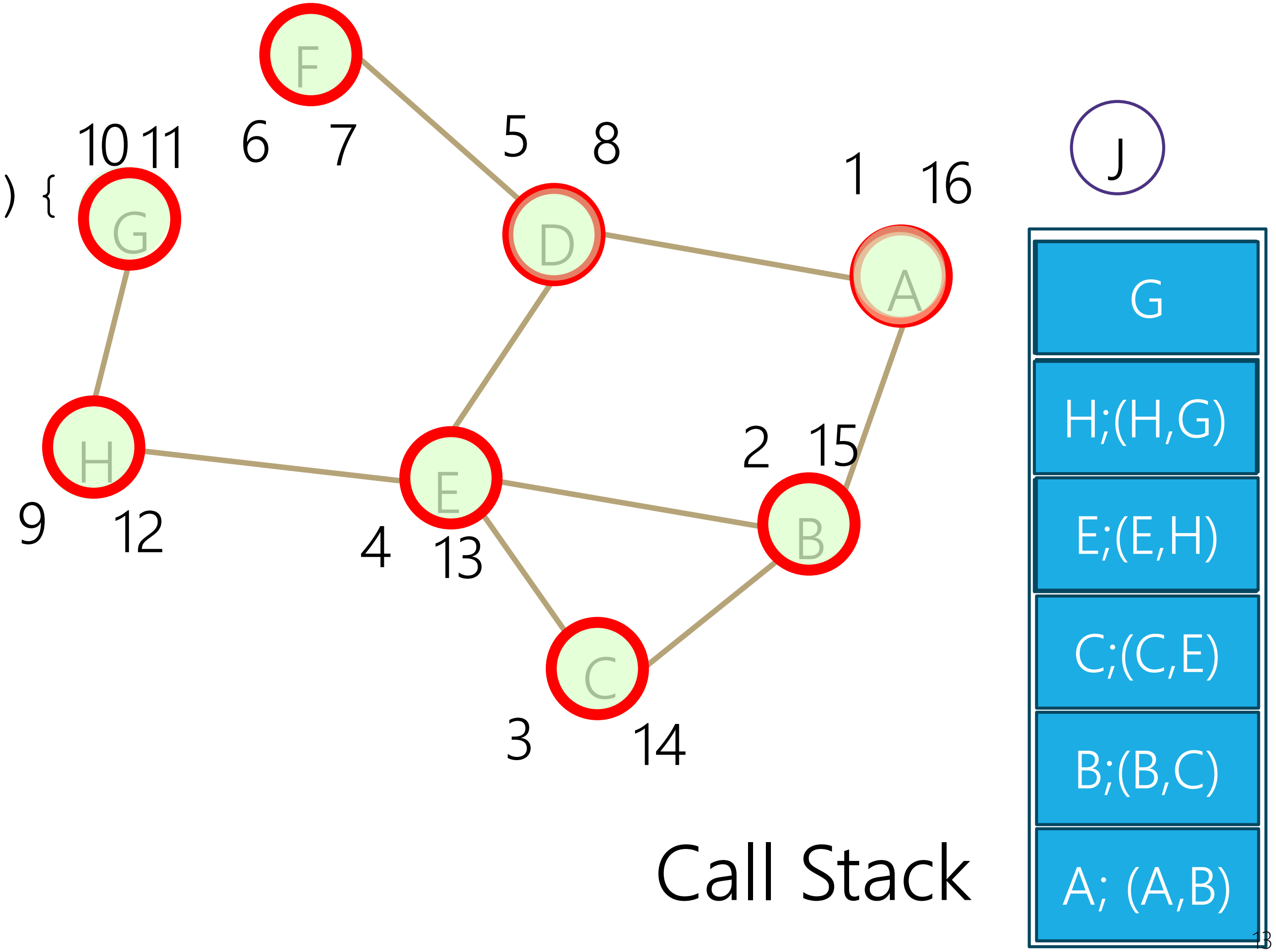
For DFS, it’s common to classify based on “start” and “finish” times

When vertices go on the (call) stack, and when they come off.

Depth First Search

(A,B), (B,C), (C,E) cause a new vertex to go on the stack.
(E,B) goes **"back"** to an edge that's already on the stack, but not finished.

```
dfs(graph, curr)
   mark curr as visited
   record curr.start
   for(v : curr.neighbors()){
      if(v is not visited){
         dfs(graph, v)
      }
   }
   mark curr as "done"
   record curr.end
```



Finished : F D G H E C B A

Saving the path

Our graph traversals can answer the “reachability question”:

- “*Is there* a path from node x to node y?”

Q: But what if we want to *output the actual path*?

- Like getting driving directions rather than just knowing it’s possible to get there!

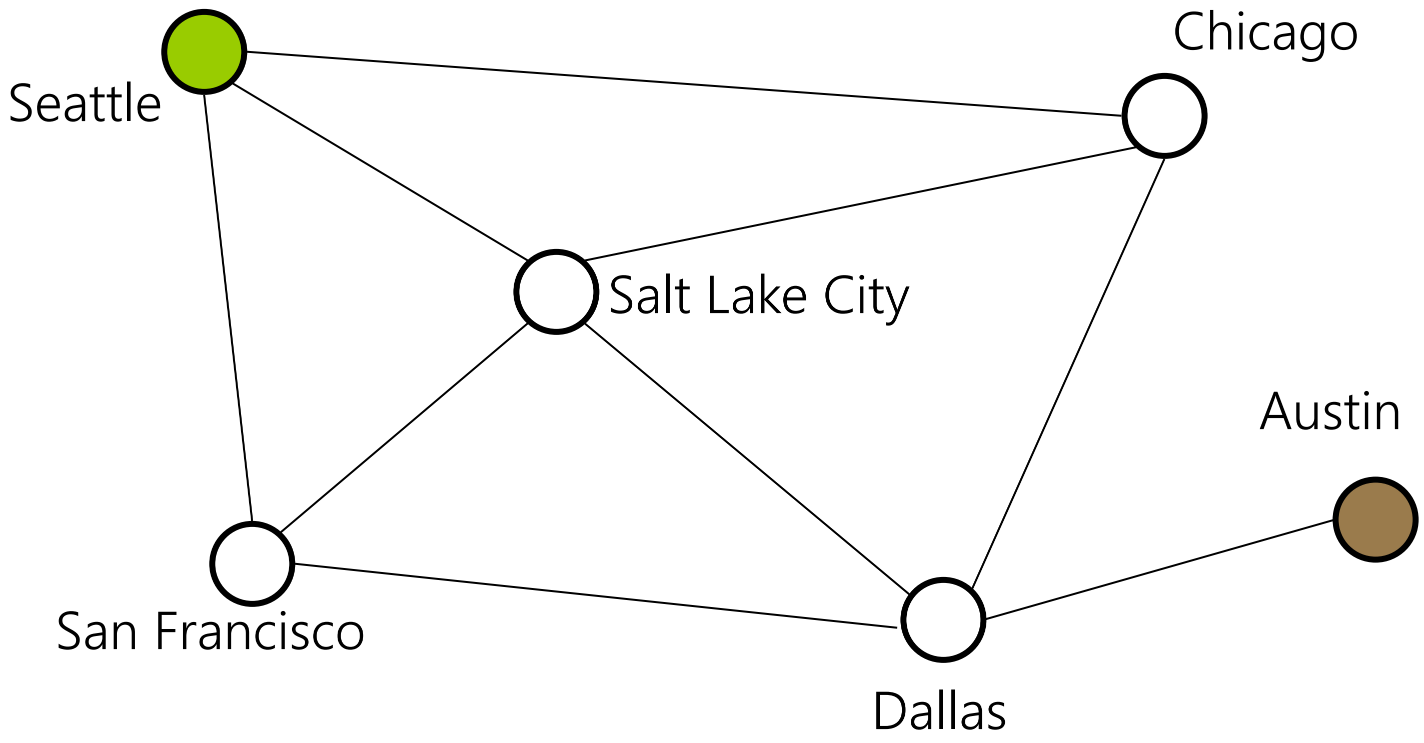
A: Like this:

- Instead of just “marking” a node, store the previous node along the path (when processing **`u`** causes us to add **`v`** to the search, set **`v.pred`** field to be **`u`**)
- When you reach the goal, follow **`pred`** fields backwards to where you started (and then reverse the answer)
- If just wanted path *length*, could put the integer distance at each node instead

Example using BFS

What is a path from Seattle to Austin

- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique



Example using BFS

What is a path from Seattle to Austin
- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique