Reminders and Topics

This lecture:

- **Depth-First Search in Graphs**
- **Breadth-First Search in Graphs**
- **Search Applications**

Graph Search / Traversal

- Graph search / traversal is a fundamental operation:
  - Is there a path from vertex X to vertex Y? If so, what's the shortest path?
  - Is this a connected graph? If not, how many connected sub-graphs are there?
  - As you will see later, many interesting problems can be formulated as graph search problem
- For **binary trees**, we learned three types of traversals: in-order, pre-order, post-order.
- For **graphs**, generally two types: **DFS, BFS**

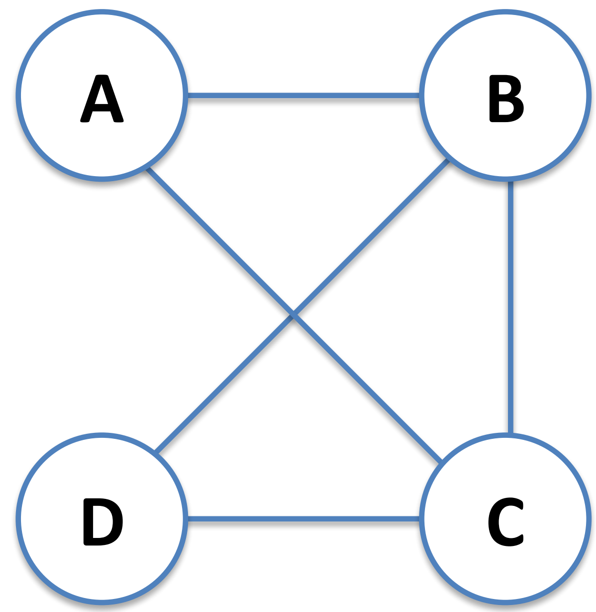
DFS and BFS

- **Depth-First Search**: start at a vertex, follows its edges to visit the deepest point, then moving back.
  - Go as far from the starting vertex as possible (path length or depth), before moving back.
  - Use a **Stack** to track where to go next.
- **Breadth-First Search**: traverse vertices in ‘levels’ — starting from a vertex, visit all its immediate neighbors, then neighbors of neighbors, and so on.
  - Stay as close to the starting vertex as possible (breadth), before moving to the next level.
  - Use a **Queue**.

Depth-First Search (DFS)

Example: start from vertex A. Visit all vertices using DFS. Initial version of pseudo-code:

```
push A to stack
while(stack not empty) {
  v = stack.pop();
  print/visit v
  push v's neighbors
  to stack
}
```

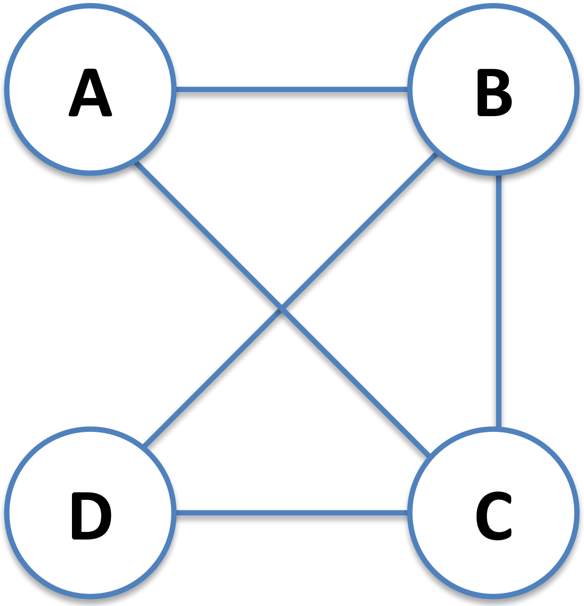


Depth-First Search

Example: start from vertex A. Visit all vertices using DFS. Initial version of pseudo-code:

Stack status:

```
A         // push A
C B       // pop A, and push
             A's neighbors
C D C A   // pop B, and push
             B's neighbors
C D C C B
```

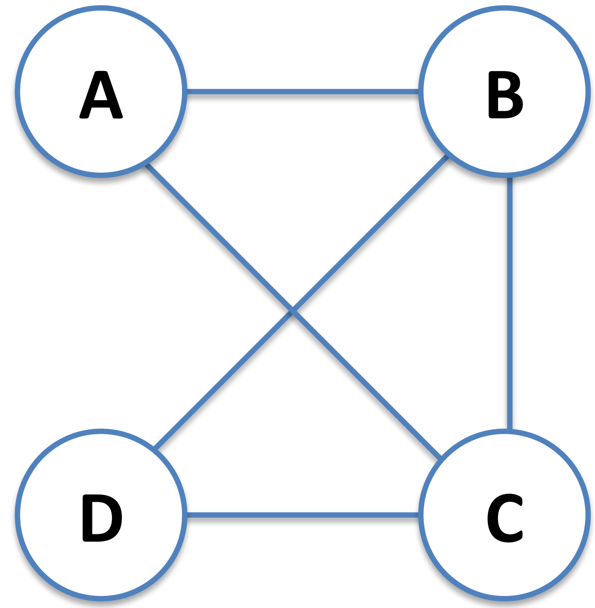


Wait a minute, there is a problem!

How do we fix it?

Depth-First Search

Correct version:

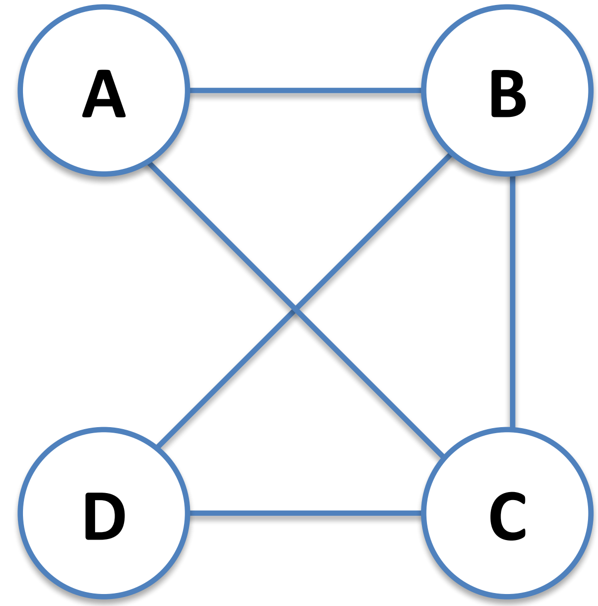



```
push A to stack
while(stack not empty) {
  v = stack.pop();
  print/visit v
  mark v as visited
  iterate through v's neighbors  {
    if a neighbor is not marked yet
      push it to stack
  }
}
```

Depth-First Search

Stack status:

```
A          // push A
C B        // pop, mark A, push
              unvisited neighbors
C D C      // pop, mark B, push
              unvisited neighbors
C D D      // pop, mark C, push
              unvisited neighbors
C D        // pop, mark D, no
              unvisited neighbors any more.
```

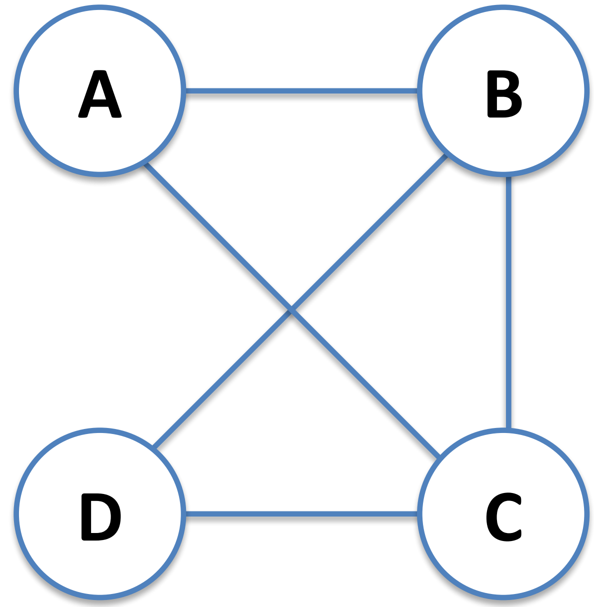


Something still doesn't seem ideal here. Why?

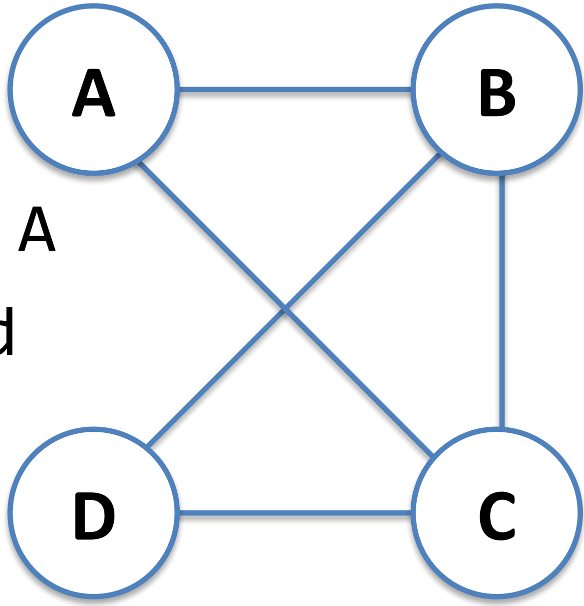
Depth-First Search

Efficient version:

```
mark A and print A
push A to stack
while(stack not empty) {
  v = stack.peek();
  n = getNextUnvisitedNeighbor(v);
  if none found, pop stack;
  else {
    mark n, print n, push n to stack
  }
}
```



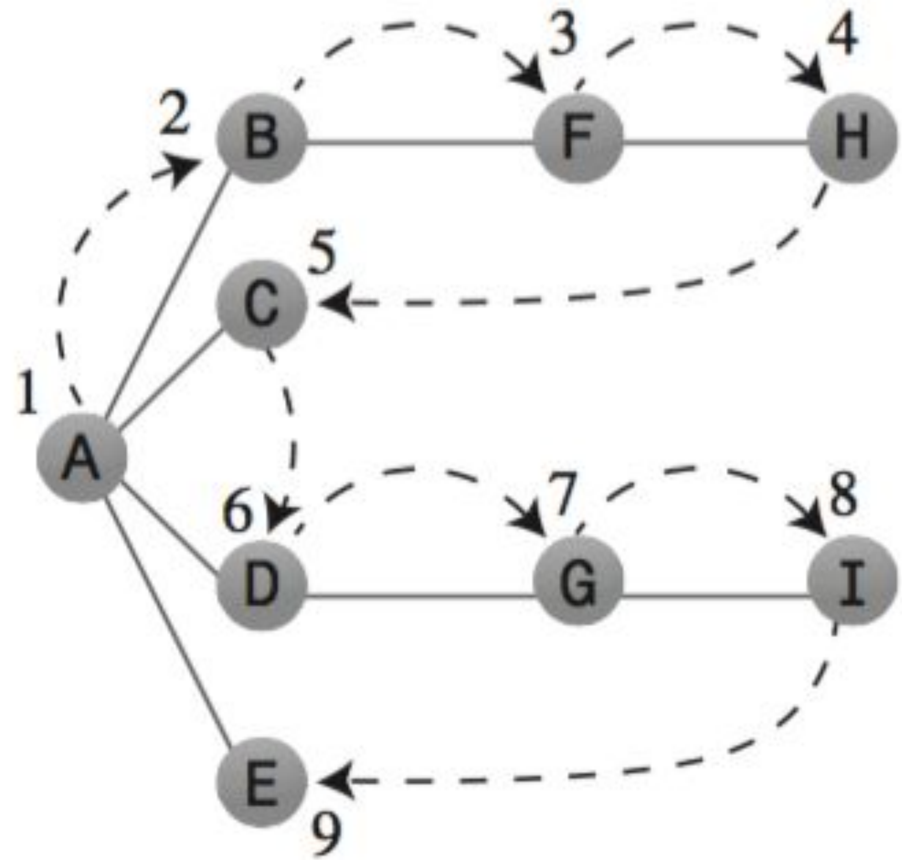
Stack status:




```
A         // print A, mark A, push A
A B       // get A's next unvisited
             neighbor B, mark it
             push it to stack
A B C     // get B's next unvisited
             neighbor C, mark it, push it
A B C D   // get C's next unvisited neighbor
             D, mark it, push it.
A B C     // D has no unvisited neighbor, pop
A B       // C has no unvisited neighbor, pop
A         // pop
(empty)   // the end
```

Depth-First Search

Another Example (this is a tree): start from vertex A. Visit all vertices using DFS.



DFS Implementation Details

Assume the graph data structures stores:

- Vertices in an array called **vertexList**[]
- Each vertex has a field called **wasVisited**
- Adjacency matrix in a 2D array called **adjMat**[][]
- The number of vertices in **nVerts**

```
class Graph {
  private Vertex vertexList[];
  private int adjMat[][];
  private int nVerts;
  … …
```

DFS Implementation Details

```
// returns the next unvisited neighbor of v
public int getNextUnvisitedNeighbor (int v) {
  for (int j=0; j<nVerts; j++) {
    if( adjMat[v][j] == 1 &&
        vertexList[j].wasVisited == false ) {
      return j;
    }
    return -1;     // return index -1 if none found
}
```

DFS Implementation Details

```
// DFS from a given start vertex
public void DFS (int start) {
  vertexList[start].wasVisited = true;   // mark it
  print(start);
  stack.push(start);
  while(!stack.isEmpty()) {









    }
  }
  // clear wasVisited marks
}
```

DFS Implementation Details

```java
// DFS from a given start vertex
public void DFS (int start) {
  vertexList[start].wasVisited = true;   // mark it
  print(start);
  stack.push(start);
  while(!stack.isEmpty()) {
    int b = getNextUnvisitedNeighbor(stack.peek());
    if (b==-1)  stack.pop(); // no unvisited neighbor
    else {
      vertexList[b].wasVisited = true;
      print(b);
      stack.push(b);
    }
  }
  // clear wasVisited marks
}
```

Finding a Path using DFS

```
// DFS from start vertex to end vertex
public boolean hasPath (int start, int end) {
  vertexList[start].wasVisited = true;   // mark it
  stack.push(start);
  int b = -1;
  while(!stack.isEmpty()) {
    b = getNextUnvisitedNeighbor(stack.peek());
    if (b==end) break;
    if (b==-1)  stack.pop(); // no unvisited neighbor
    else {
      vertexList[b].wasVisited=true;   stack.push(b);
    }
  }
  if(b==end) return true;
  else return false;
}
```

So how do I print out the path??

Vertices on the path are all stored in the stack!

Clicker Question #1

Given the adjacency matrix, if we perform DFS starting from vertex 2, in what order will the other vertices be visited?

(a) 4, 0, 3, 1

(b) 3, 0, 1, 4

(c) 3, 4, 0, 1

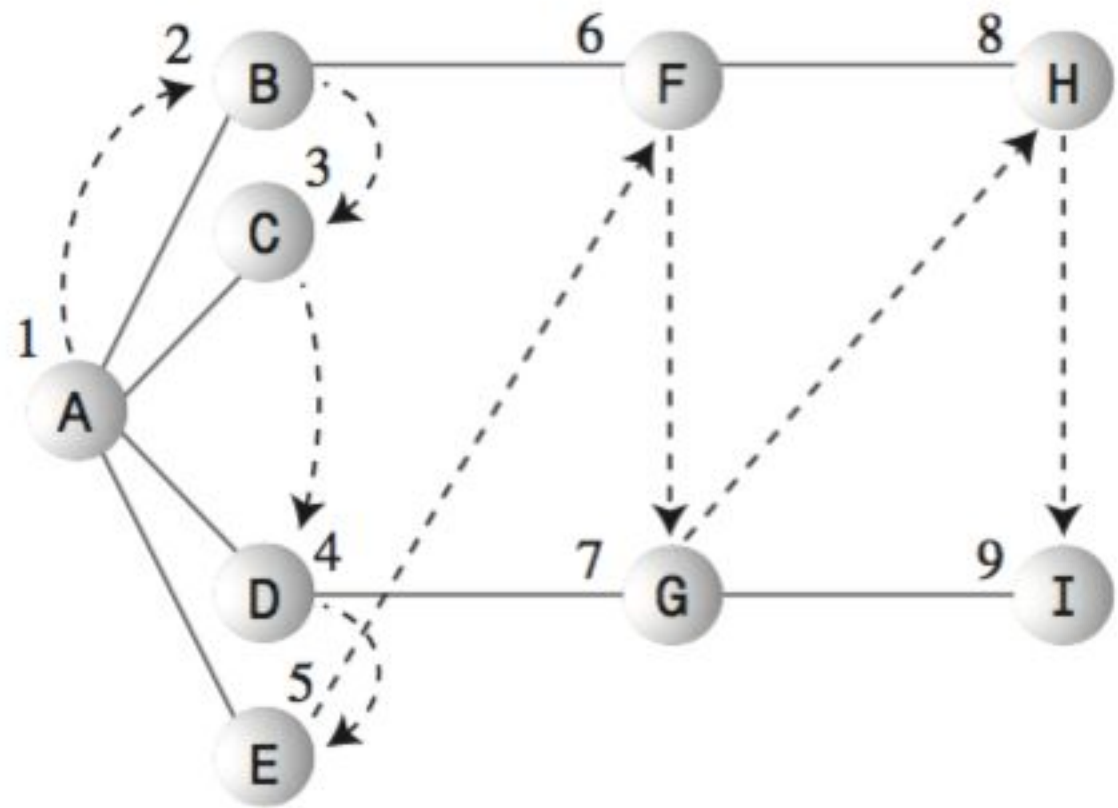
(d) 3, 0, 4, 1

(e) 3, 4, 1, 0

	0	*1*	*2*	*3*	*4*
0	0	0	0	1	1
1	0	0	0	1	0
2	0	0	0	1	1
3	1	1	1	0	1
4	1	0	1	1	0

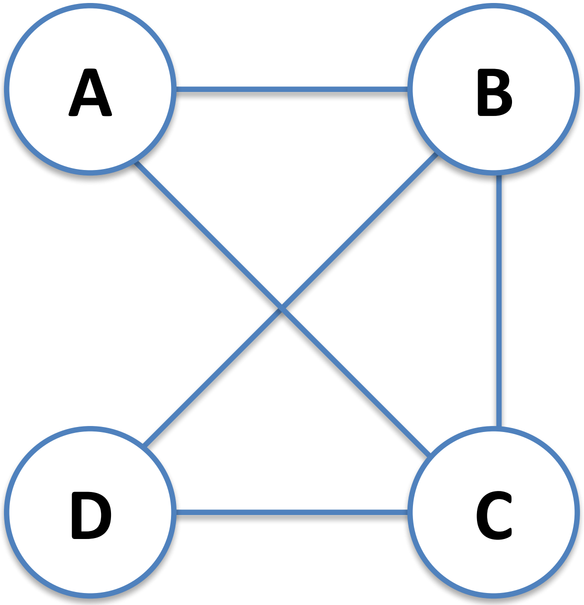
Breadth-First Search (BFS)

Example: start from vertex A. Visit all vertices connected to A (i.e. A's neighbors) using BFS.



Breadth-First Search (BFS)

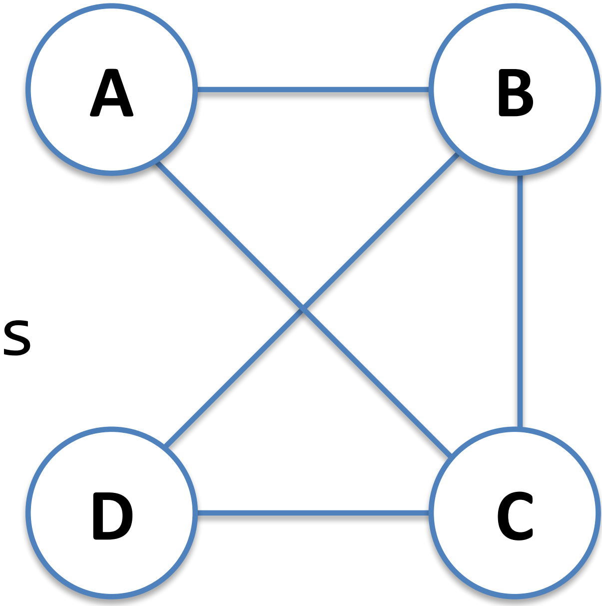
Pseudo-Code:




```
mark A and print A
enqueue A to queue
while(queue not empty) {
   v = queue.dequeue();
   while((n=getNextUnvisitedNeighbor(v)) != -1) {
      mark n, print n, enqueue n
   }
}
```

Queue status:

```
A          // enqueue A
B C        // dequeue A, add all A’s
              unvisited neighbors,
              visit them and,
              mark them as visited
C D        // dequeue B, add all B’s
              unvisited neighbors,
              visit and mark them
D          // dequeue C,
              C has no unvisited neighbors
(empty)    // the end
```



BFS Implementation Details

```
// BSF from start vertex
public void BFS (int start) {
  vertexList[start].wasVisited = true;   // mark it
  print(start);  queue.enqueue(start);
  int b;
  while(!queue.isEmpty()) {
    int v = queue.dequeue();
    while((b=getNextUnvisitedNeighbor(v)) != -1) {
      vertexList[b].wasVisited = true;
      print(b);  queue.enqueue(b);
    }
  }
  // clear wasVisited marks
}
```

Clicker Question #2

Given the adjacency matrix, if we perform BFS starting from vertex 2, in what order will the other vertices be visited?

(a) 3, 4, 0, 1

(b) 3, 0, 1, 4

(c) 4, 0, 3, 1

(d) 3, 4, 1, 0

(e) 3, 0, 4, 1

	0	*1*	*2*	*3*	*4*
0	0	0	0	1	1
1	0	0	0	1	0
2	0	0	0	1	1
3	1	1	1	0	1
4	1	0	1	1	0

BFS Implementation Details

- Similar to DFS, we can also modify the basic BFS code to find a path from a starting vertex to an ending vertex.

- Once a path is found, it’s guaranteed to be the shortest (in terms of path length) from start to end. Why?

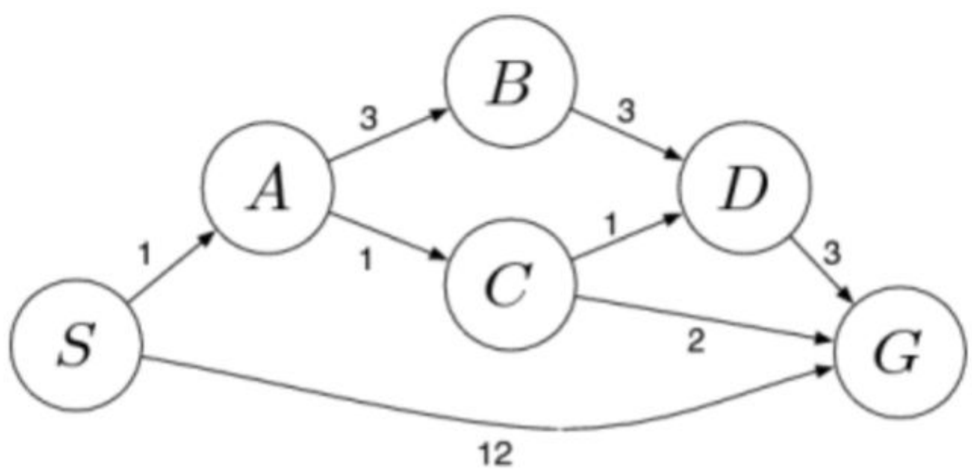
- Question though: how do we reconstruct the path from start to end? What information would you need to reconstruct this path?

BFS Implementation Details

- How do we reconstruct the path from start to end when a BFS path is found?
  - You can modify the Vertex data structure to store a ‘parent’ index. During BFS, when you get a vertex v’s next unvisited neighbors, you will mark those neighbors as having v as their parents (i.e. where they came from).
  - Once a path is found, simply trace back the parent link to reconstruct the path.

Search in Weighted Graphs

- In weighted graphs, we generally care about the shortest path from vertex X to Y in terms of the path weight (sum of weights on the path), not merely path length.
- Although we can still perform DFS and BFS in weighted graphs, they are often not so useful as they don't account for edge weight. For example, perform **BFS** on the following graph to find a path from S to G, what would you get? Is it the shortest path in terms of weight?



Search in Weighted Graphs

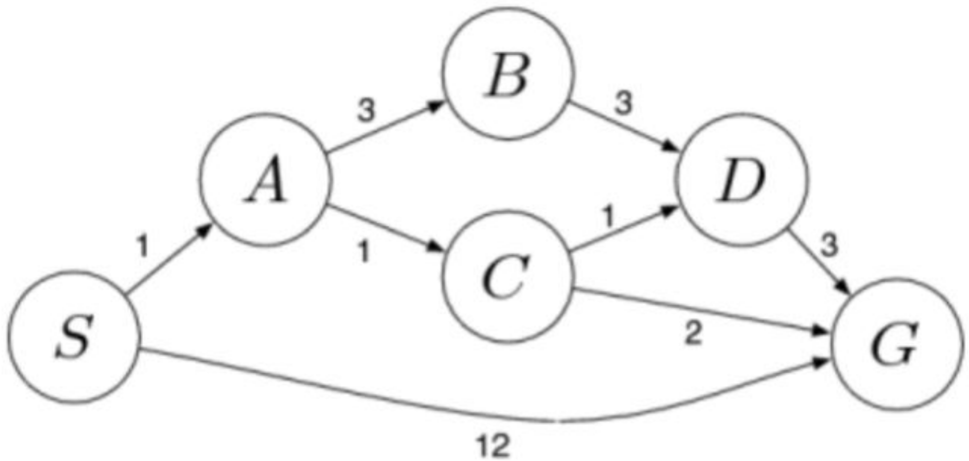
- BFS gives you a path with minimum number of edges / segments / hops, but not necessarily the shortest path (in terms of total weight).
- Analogy: BFS in a air flight graph can give you an itinerary with minimum number of connections, but not necessarily the cheapest total price!
- **Shortest-Path Algorithm**: turns out we can modify BFS slightly to implement shortest-path algorithm.
  - The trick here is to use a Priority Queue instead of the standard FIFO Queue.
  - Shortest-path algorithm is a rich topic. You will learn more in upper-level classes.

Uniform Cost Search (UCS)

```
Insert the starting vertex into the queue
while the queue is not empty
  Dequeue the highest priority vertex v from the queue
  Mark v as visited
  if v is the end vertex
      print the path and exit
  else
      for each of node's unmarked neighbors n
        COST = cost(node) + cost(node -> n)
        queue.add(n, COST)
```

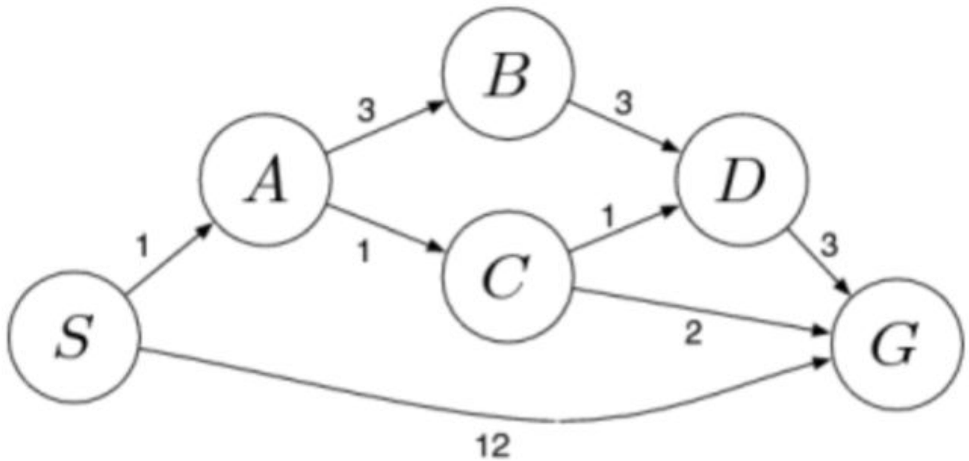
Uniform Cost Search (UCS)

```
Insert the starting vertex into the queue
while the queue is not empty
  Dequeue the highest priority vertex v from the queue
  Mark v as visited
  if v is the end vertex
      print the path and exit
  else
      for each of node's unmarked neighbors n
        COST = cost(node) + cost(node -> n)
        queue.add(n, COST)
        If n has no parent OR COST < shortest cost to n
            set n's parent to node and shortest cost to COST
```



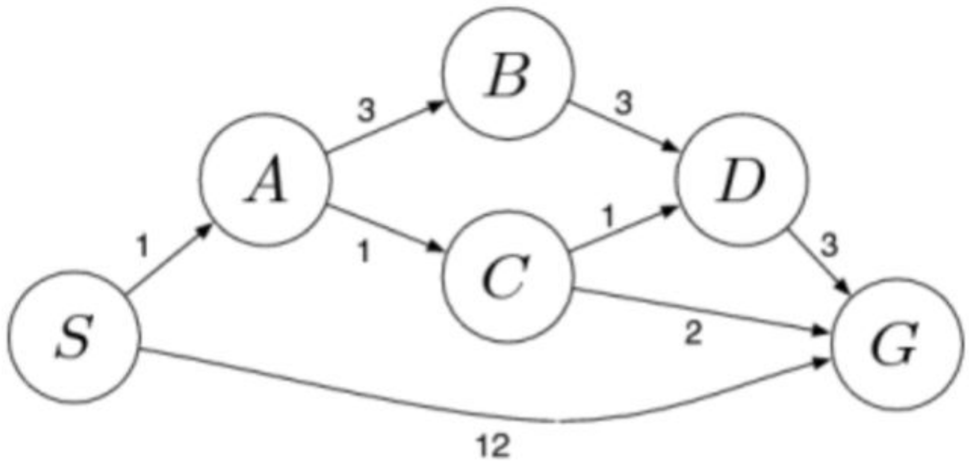
[S,0]

	S	A	B	C	D	G
Visited						
Parent						
Shortest						



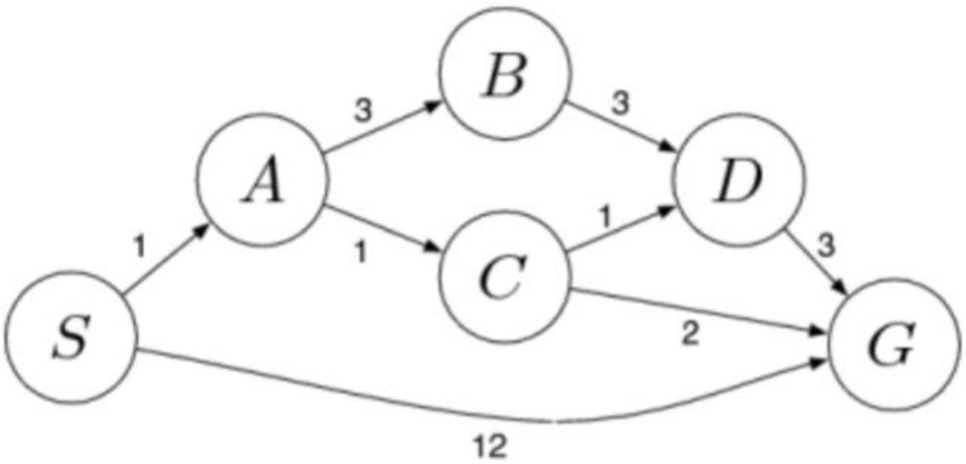
[G,12] [A,1]

	S	A	B	C	D	G
Visited	true					
Parent		S				S
Shortest		1				12



[G,12] [B,4] [C,2]

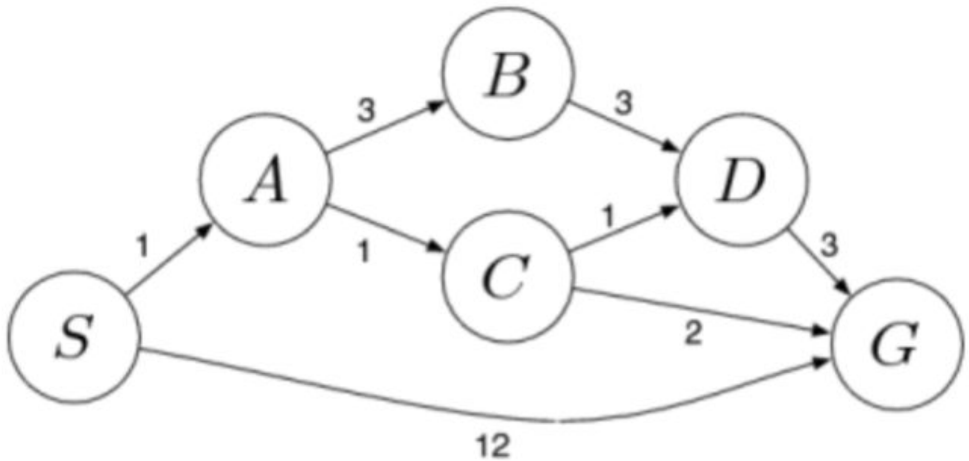
	S	A	B	C	D	G
Visited	true	true				
Parent		S	A	A		S
Shortest		1	4	2		12



[G,12] [G,4] [B,4] [D,3]

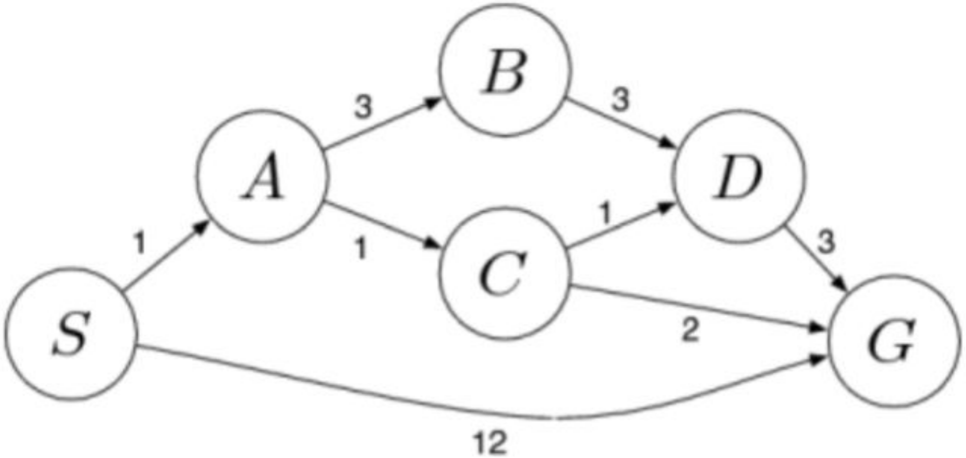
Here we found a shorter path to G, so replace it's parent and cost

	S	A	B	C	D	G
Visited	true	true		true		
Parent		S	A	A	C	C
Shortest		1	4	2	3	4



[G,12] [G,6] [G,4] [B,4]

	S	**A**	**B**	**C**	**D**	**G**
Visited	true	true		true	true	
Parent		S	A	A	C	C
Shortest		1	4	2	3	4



We don’t insert D again because we already visited it

[G,12] [G,6] [G,4]

	S	**A**	**B**	**C**	**D**	**G**
Visited	true	true	true	true	true	
Parent		S	A	A	C	C
Shortest		1	4	2	3	4

Clicker Question #3

Suppose we perform a UCS on this directed graph starting at vertex a, with b as the goal. When we first put b on the priority queue, what is its distance?

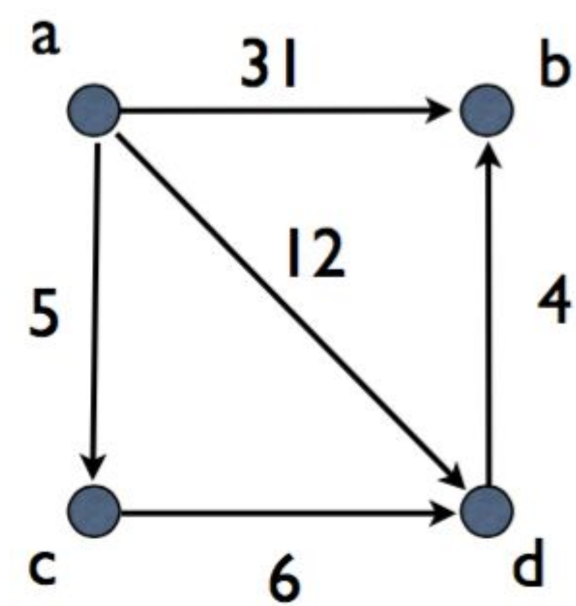
(a) 0

(b) 4

(c) 15

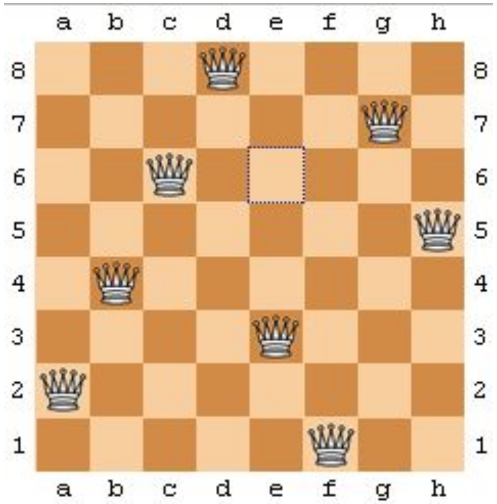
(d) 16

(e) 31

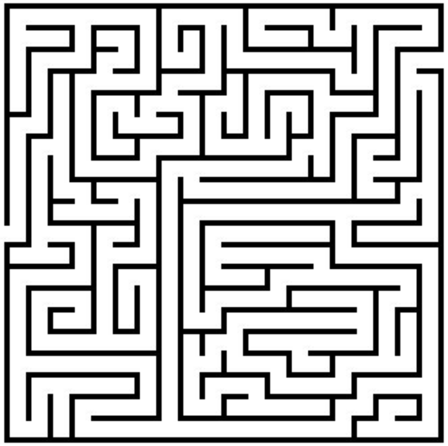


Graph Search Applications

- A lot of interesting computational problems can be viewed as graph search problems, where each vertex represents a state and each edge represents a valid move from one state to another state. Examples:



  - [Eight Queens Puzzle]()
  - [Knight's Tour]()
  - Sudoku
  - Maze
  - Flood Fill



Graph Search Applications

- A generally useful algorithm is called **backtracking**. It incrementally builds the candidates to the solutions, and abandons a partial solution **s** (i.e. backtracks) as soon as **s** is found to be impossible to form a valid complete solution.

- We can use a **stack** to store the current partial solution (e.g. a subset of eight queens, or a partial path in a maze). The stack allows us to go back (backtrack) to a previous state, and proceed to build the next partial solution.

Graph Search Applications

- For example, for the **Four Queens** Problem:
  - **We know that each row holds 1 and only 1 queen.**
  - Place the first queen at (1, 1), push it to stack.
  - Place the second queen at (2, 1). Is there a conflict with the first queen? If so, move the second queen to (2, 2), (2, 3) and so on until there is no conflict. Push it to stack.
  - Place the third queen at (3, i) where i is from 1 to 4, until there is no conflict with the previous queens. Push it to stack.
  - If you can’t find a valid spot for a queen (say, the third queen), pop the stack (thus you are back to working on the second queen), search for the **next** valid spot for it and push to stack.