Graph Traversal

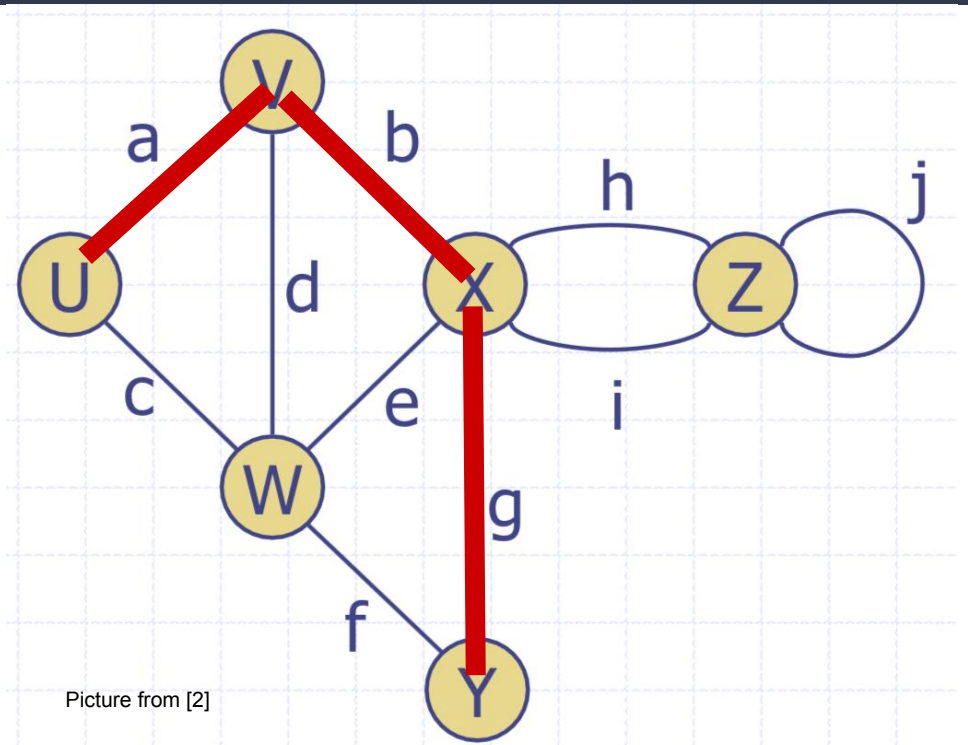
Part 1: DFS
Part 2: BFS

Part 1: Depth-First Search

- More Terminology
- DFS
- Path Finding
- Cycle Finding

More Terminology...

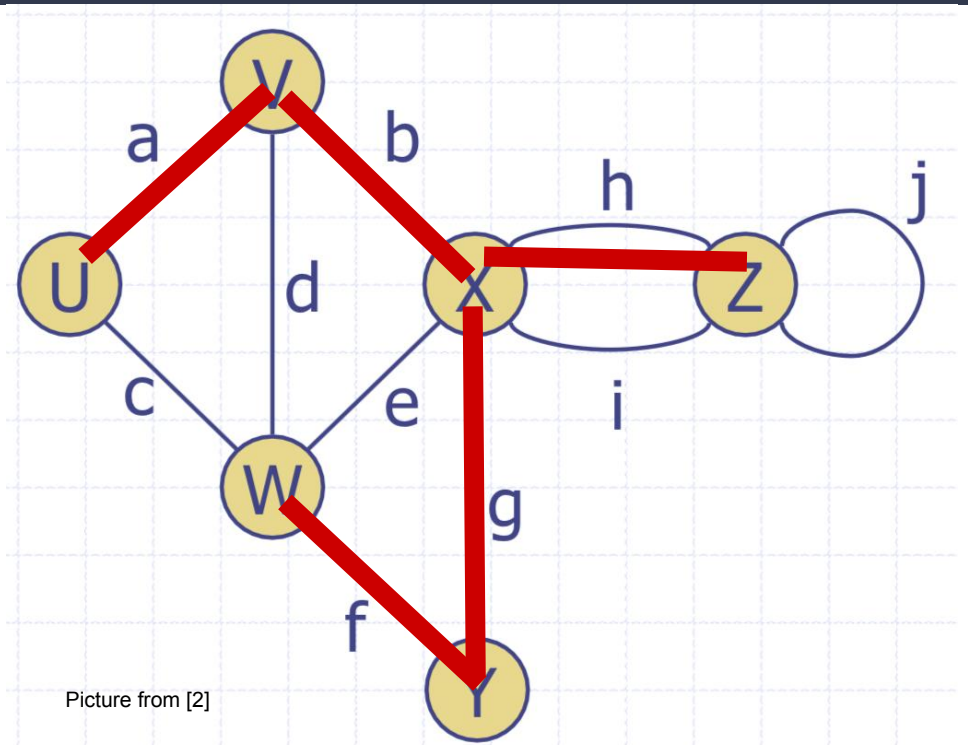
- **subgraph** (of a graph **G**): a graph made up of a subset of the vertices of **G** and a subset of the edges of **G**
- EX: **S** = ({U, V, X, Y}, {a, b, g})



Picture from [2]

More Terminology...

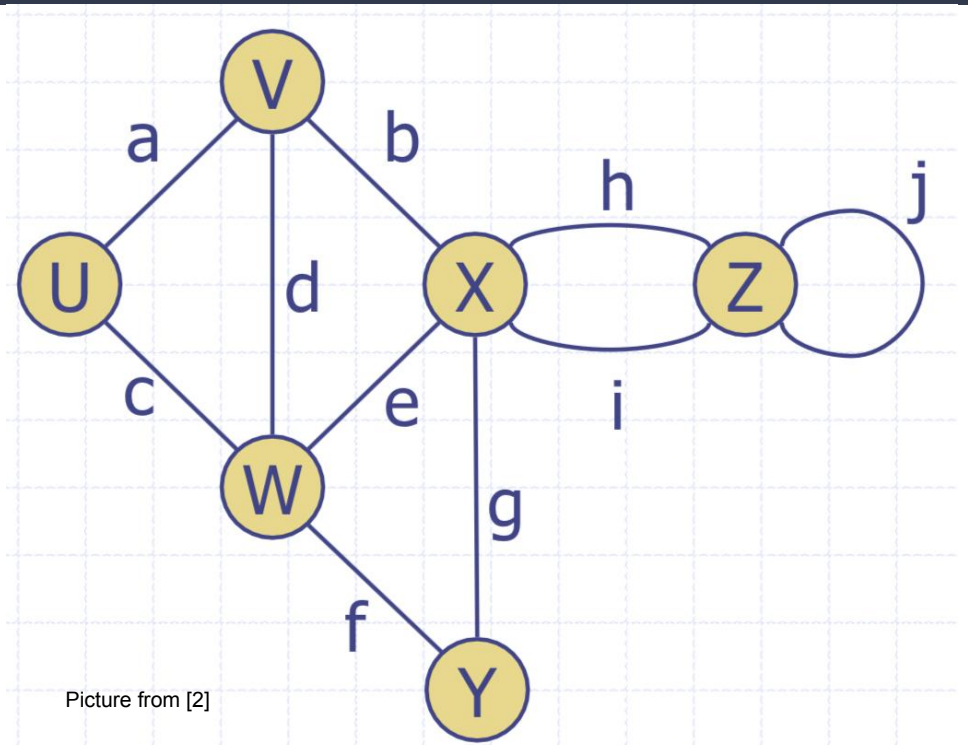
- **subgraph** (of a graph **G**): a graph made up of a subset of the vertices of **G** and a subset of the edges of **G**
- EX: **S** = ({U, V, X, Y}, {a, b, g})
- **spanning subgraph** (of a graph **G**): a subgraph that includes all the vertices of **G**
- EX: **S** = ({U, V, W, X, Y, Z}, {a, b, g, f, h})



Picture from [2]

More Terminology...

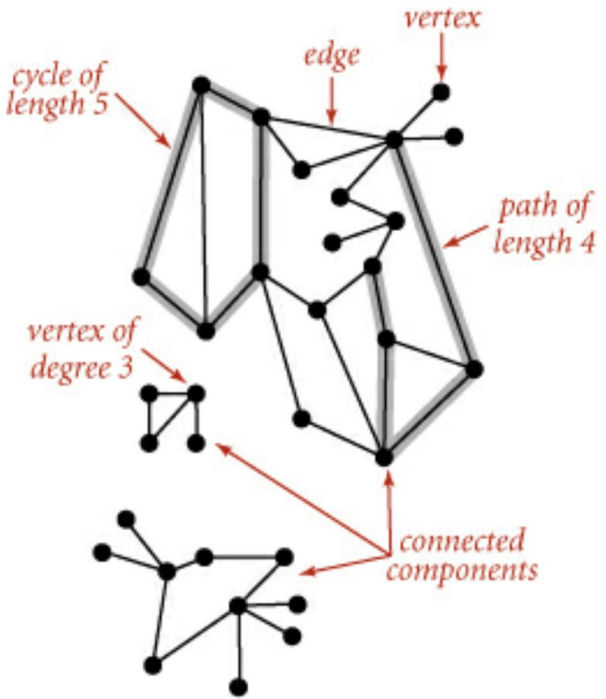
- A **connected** graph is a graph where there is a path from every vertex to every other vertex



Picture from [2]

More Terminology...

- A **connected** graph is a graph where there is a path from every vertex to every other vertex
- A **connected component** of a graph **G** is a maximal connected subgraph of **G**

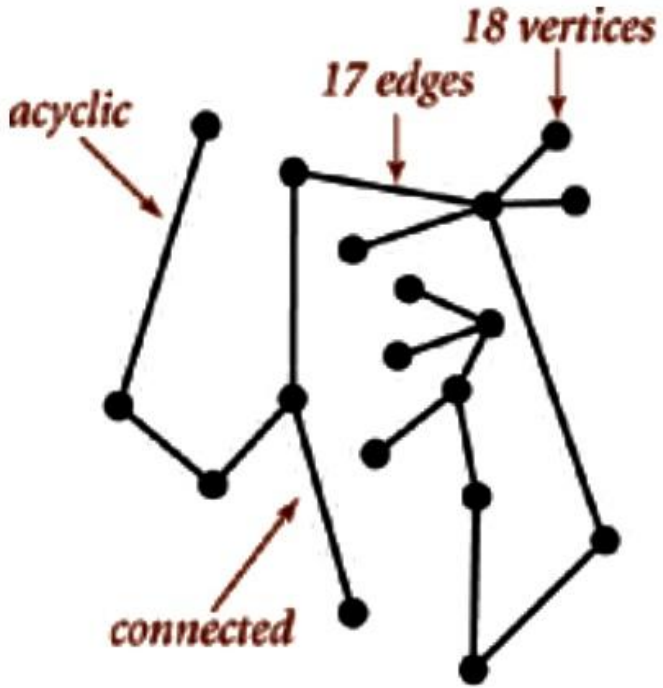


Anatomy of a graph

Picture from [1]

More Terminology...

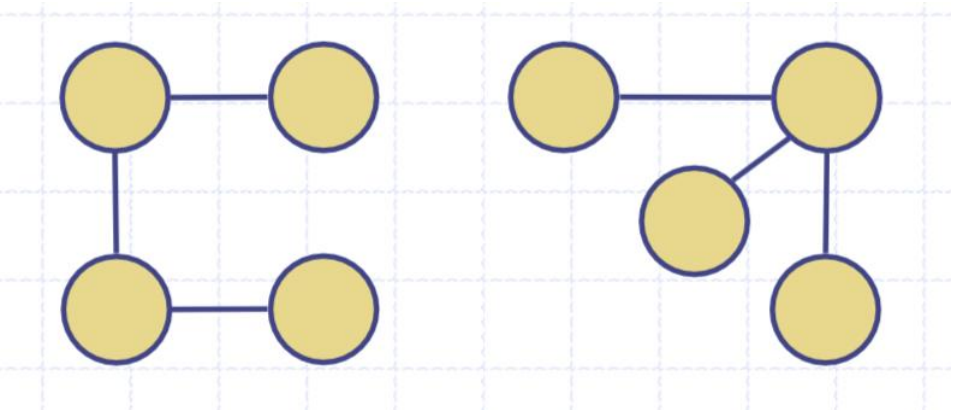
- A **tree** is an undirected graph **T** such that
  - **T** is connected
  - **T** has no cycles (acyclic)



Picture from [1]

More Terminology...

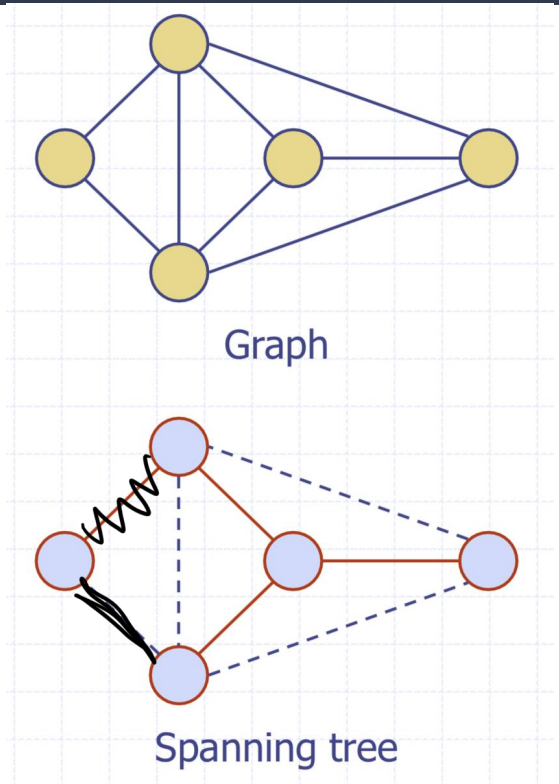
- A **tree** is an undirected graph **T** such that
  - **T** is connected
  - **T** has no cycles (acyclic)
- A **forest** is an undirected graph without cycles (i.e. one or more trees).
- The connected components of a forest are trees.



Picture from [2]

More Terminology...

- A **spanning tree** of a (connected) graph **G** is a subgraph of **G** that is a tree
- Given a graph **G**, is it possible to have more than one spanning tree?

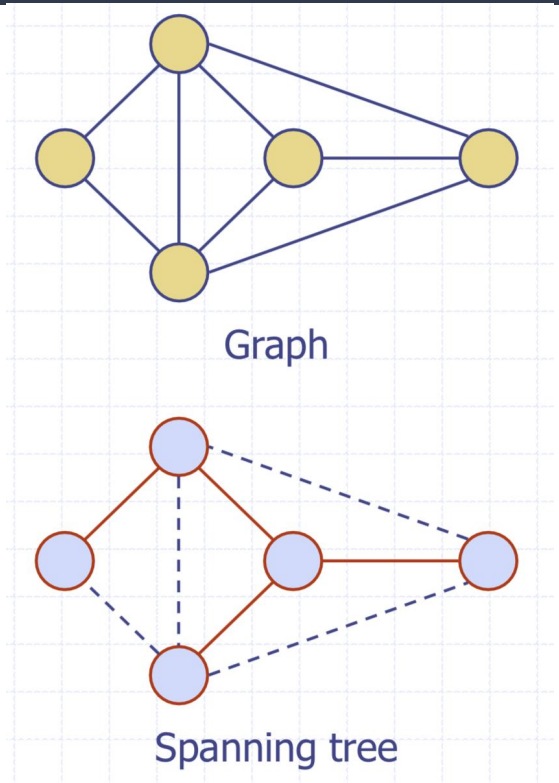


Picture from [2]

More Terminology...

- A **spanning tree** of a (connected) graph **G** is a subgraph of **G** that is a tree
- Given a graph **G**, is it possible to have more than one spanning tree?

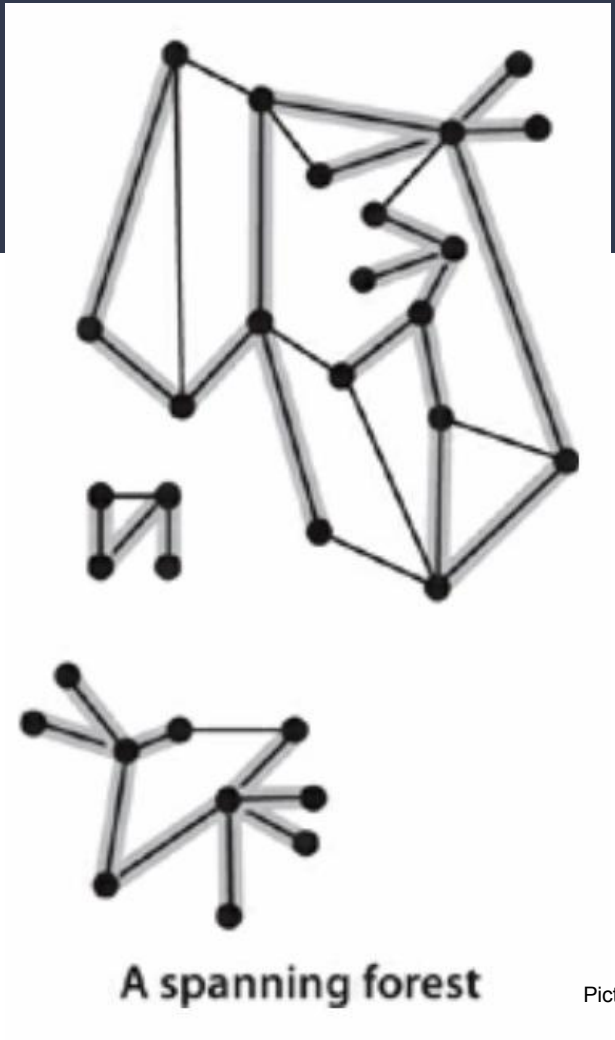
  Yes, unless G is already a tree.



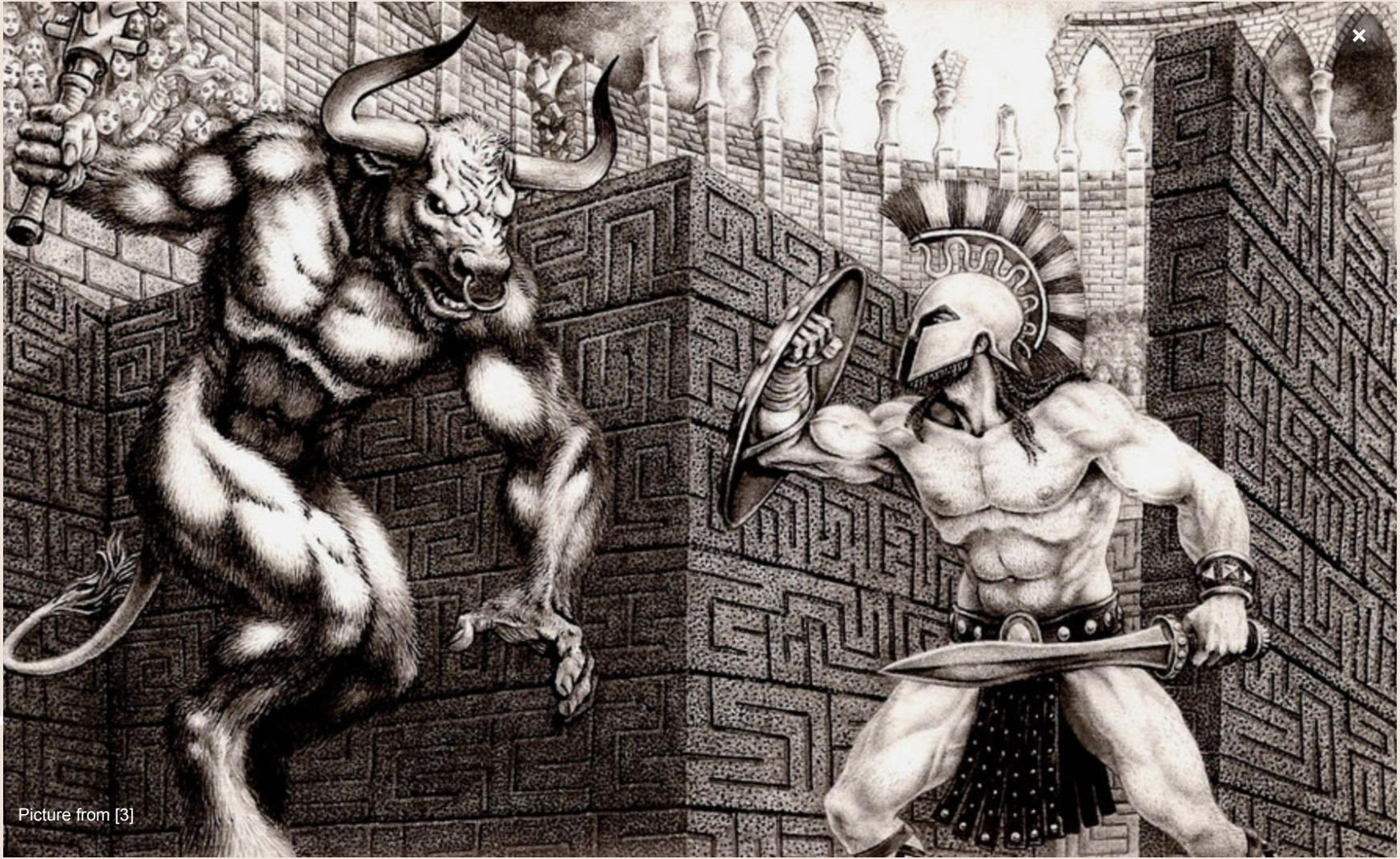
Picture from [2]

More Terminology...

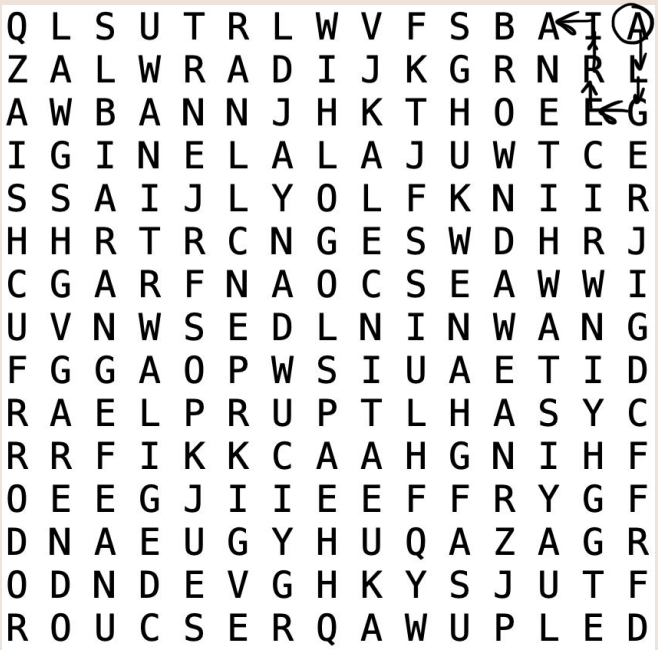
- A **spanning tree** of a (connected) graph **G** is a subgraph of **G** that is a tree
- A **spanning forest** is a subgraph that is a forest



Picture from [1]



Picture from [3]



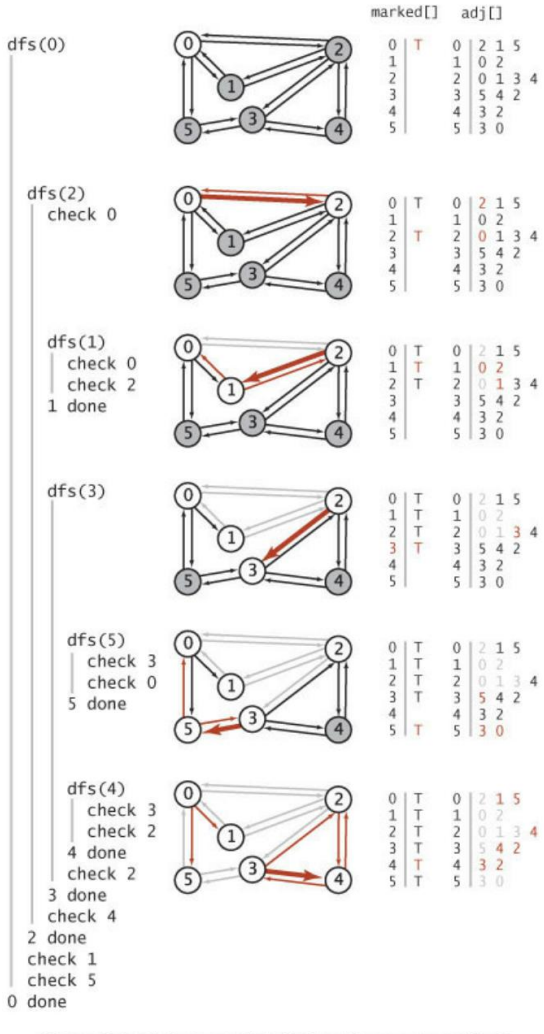

```
Algorithm dfs
Input: Grid G and starting position (row, col)
Output: path, a Stack representing the steps in the
path from Start to Finish
path := an empty Stack
push the current location onto the Stack
while you haven't reached the end of the word:
      Q := possible steps from the current position
      step := Q.dequeue()
      if there is no step (i.e. Q is empty)
      then backtrack (i.e. pop the last step from the
      stack, moving backward along that path
      accordingly)
      else make the step, pushing it onto path
      end if
end while

return path
```

DFS: Depth-First Search

- A general technique for traversing a graph
- Visits all the vertices and edges of the graph
- Determines whether or not the graph is connected
- Computes the connected components of the graph
- Computes a spanning forest of the graph (spanning tree if the graph is connected)
- Useful as basis for solving other problems (e.g. finding a path between two vertices)

Algorithm (single source)

Algorithm $dfs(G, v)$
Input: A graph $G = (V, E)$ and a source vertex v
    Mark v as visited
    for each vertex w adjacent to v
        if w has not been visited
        then $dfs(G, w)$



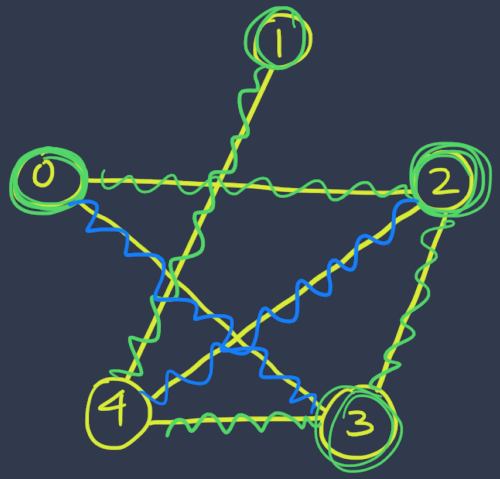
Trace of depth-first search to find vertices connected to 0

- Note the trace of the recursive calls
- Note the way visited nodes are marked by changing their color to white
- Note how a recursive call is only made if the new node has just been discovered (i.e. it has not been visited before)

- Assuming an adjacency list representation, what is the runtime analysis for DFS(G, v)?

Example 1. Do DFS on the following graph starting at vertex 0.

0	2, 3
1	4
2	0, 3, 4
3	0, 2, 4
4	1, 2, 3



marked

0	T
1	T
2	T
3	T
4	T

dfs(0)
↳ dfs(2)
↳ dfs(3)
↳ dfs(4)
↳ dfs(1)

Analysis of DFS(G, v): n vertices and m edges

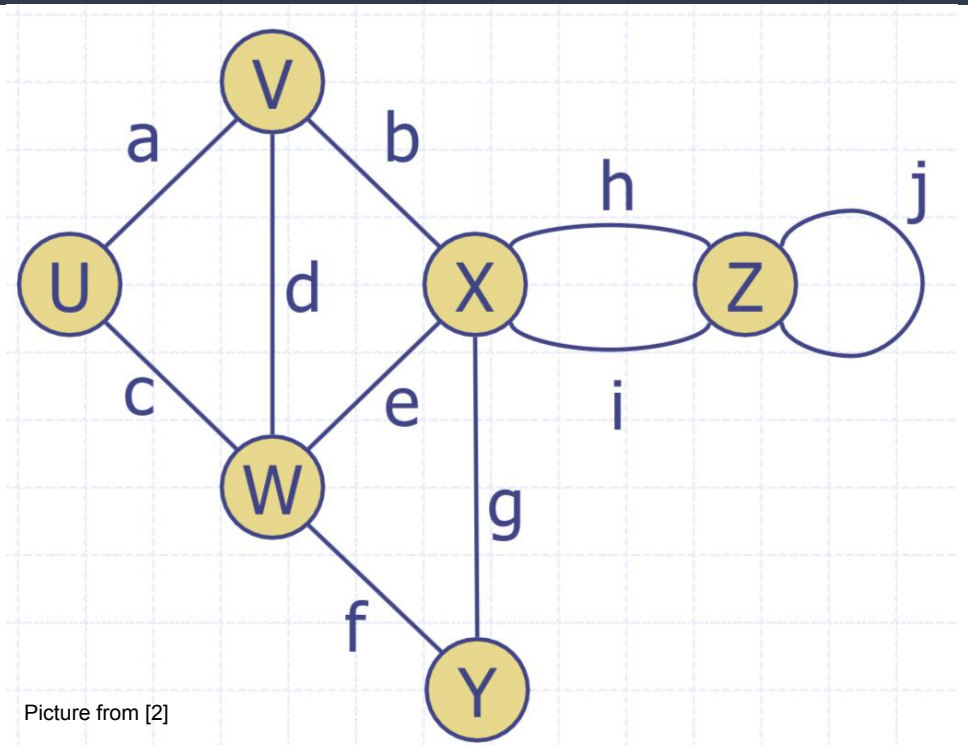
- By the end of the algorithm, we have visited all the nodes (**n**)
- And we have traversed every edge twice (**2m**)
- Total runtime with adjacency list representation: **O(n + m)**

Properties of DFS (G, v)

1. DFS(**G**, **v**) visits all edges and vertices in the connected component of **v.**
2. If the graph has multiple connected components, you can start DFS again with an unvisited node until everything has been visited.
3. The set of “discovery edges” labeled by DFS(**G**, **v**) form a spanning tree of the connected component of **v**. (A “discovery edge” is an edge that goes to a yet unvisited node.)

Problem 1: Finding a Path

Given a graph **G** = (**V**, **E**), a source vertex **v**, and a destination vertex **w**, determine if there is a path from **v** to **w**, and if there is, find the path.

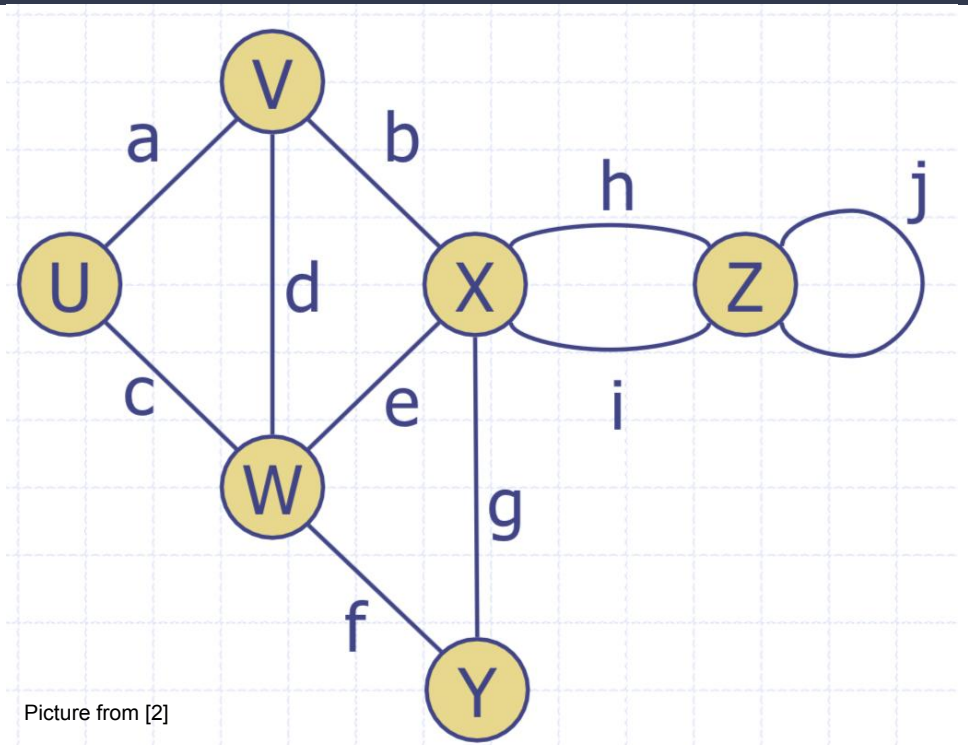


Picture from [2]

Problem 1: Finding a Path

Given a graph **G** = (**V**, **E**), a source vertex **v**, and a destination vertex **w**, determine if there is a path from **v** to **w**, and if there is, find the path.

Solution: Use DFS(**G**, **v**) and a stack **S** to keep track of the current path.



Picture from [2]

Example 2. Use DFS to find a path from vertex 0 to vertex 1 in the graph below.

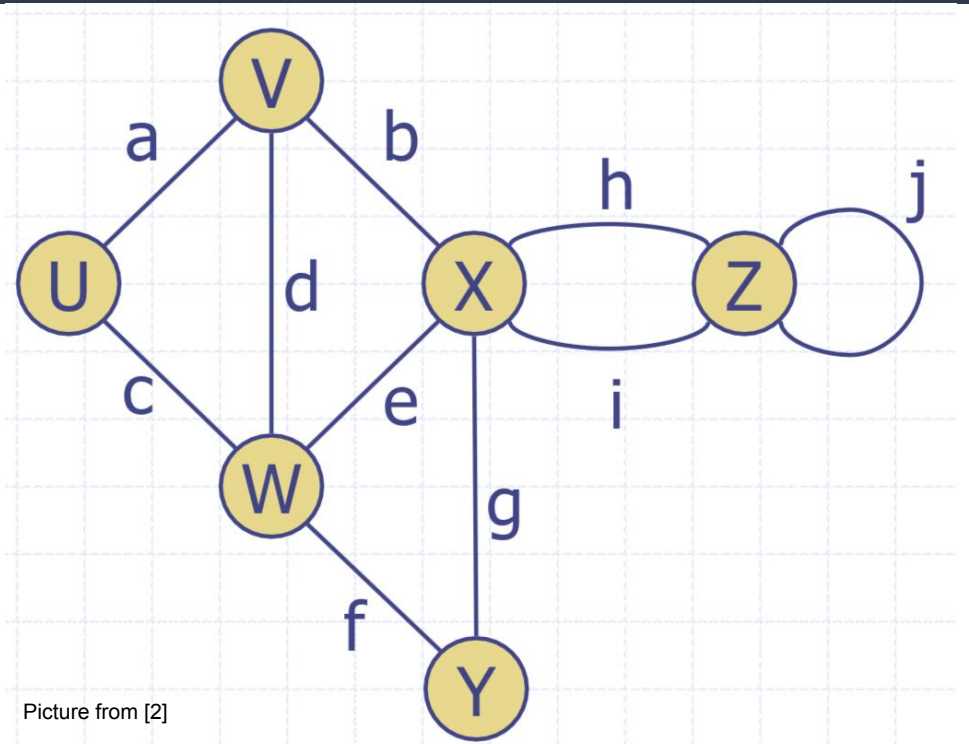
0	2, 3
1	4
2	0, 3, 4
3	0, 2
4	1, 2

S: 0 2 ~~3~~ 4 1

Problem 2: Finding a Simple Cycle

Given a graph **G** = (**V**, **E**), find a simple cycle

Solution: Use DFS(**G**, **v**) and a stack **S** to keep track of the current path—stop until the DFS algorithm is complete, or until you encounter a back edge —then return the stack from that edge back to the first occurence of the repeated vertex



Picture from [2]

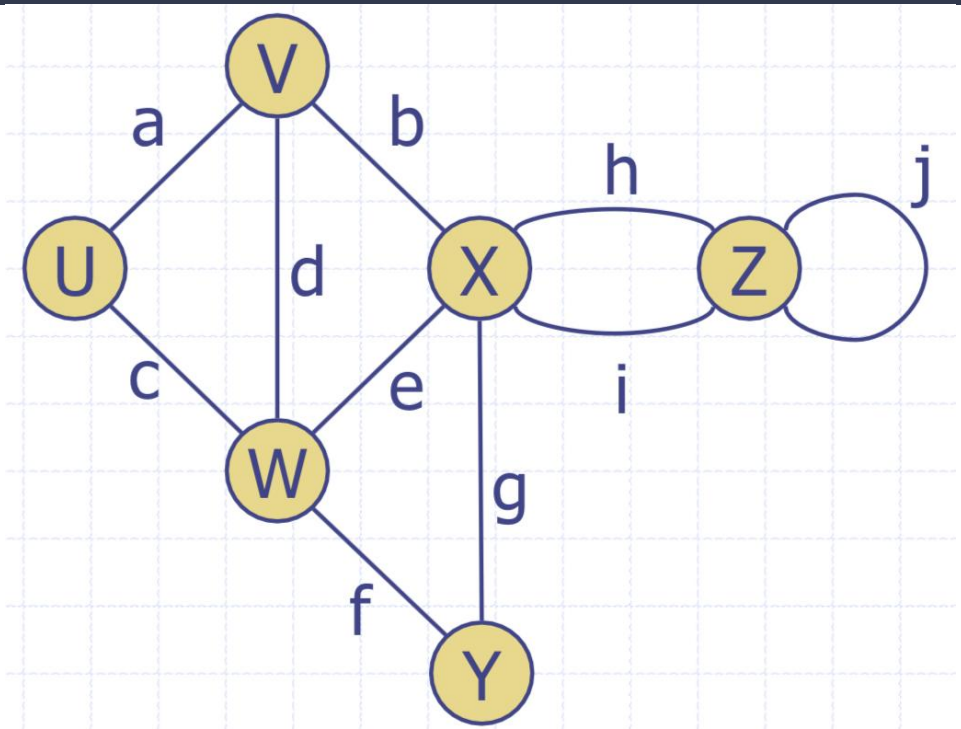
Part 2: Breadth-First Search

- Single-source shortest path
- BFS Algorithm and Code
- BFS vs. DFS

Single-source Shortest Path

Given a graph **G** = (**V**, **E**) and a source vertex **v**, find the shortest path (if a path exists) to a target vertex **w**.

Can we do this with DFS?



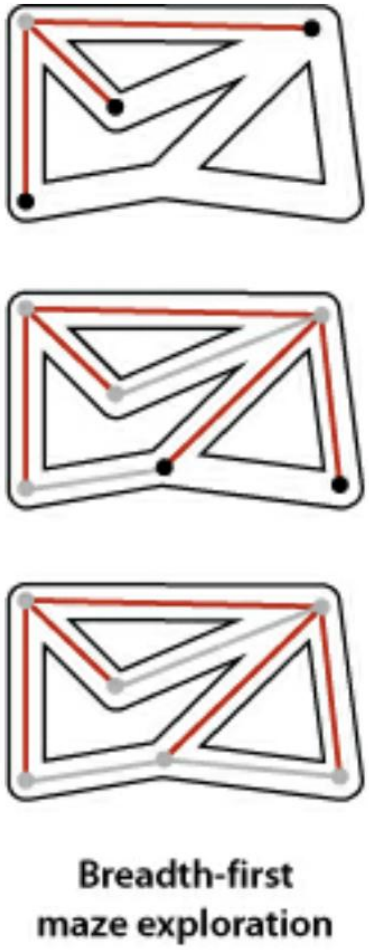
Picture from [2]

Single-source shortest path

- DFS doesn't work for this because it searches deep before wide
- We need an algorithm that searches wide before deep in a systematic way
- We need to search all paths of length 1, then all paths of length 2, etc. until we find what we are looking for or find there is nothing more to search
- Instead of a LIFO Stack like the DFS pathfinding algorithm, we will use a FIFO Queue, so we can easily explore vertices in order of their distance from the source

That algorithm is Breadth-first Search!


```
Algorithm bfs
Input grid G source location l
      Q := a queue
      Q.enqueue(l)
      while !Q.isEmpty()
            v := Q.dequeue()
            visit v
            for all w next to v:
                  if w is not in the queue and not already visited:
                        Q.enqueue(w)
```

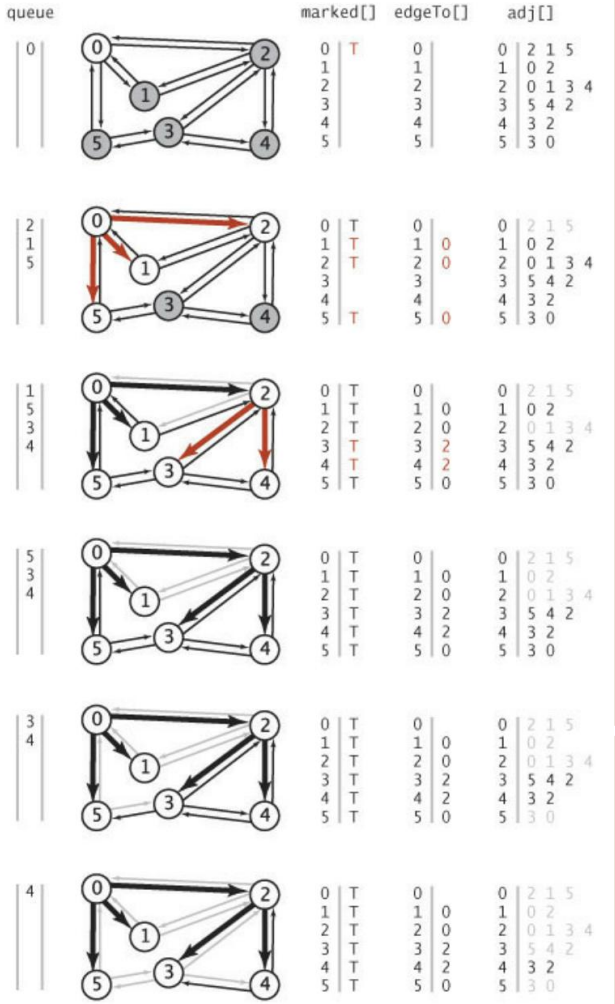


BFS is like solving a maze with a whole group of people who can spread out and try multiple paths at the same time.

Picture from [1]

Algorithm *bfs(G, s)*
Input graph *G =(V, E)* and source vertex *s*

```
    Q := a queue of size |V|
    edgeTo := an array of size |V|
    Q.enqueue(s)
    while !Q.isEmpty()
        v := Q.dequeue()
        Mark v
        for all w adjacent to v
        if w is not in the queue and not
        marked:
                Q.enqueue(w)
                edgeTo[w] = v
```



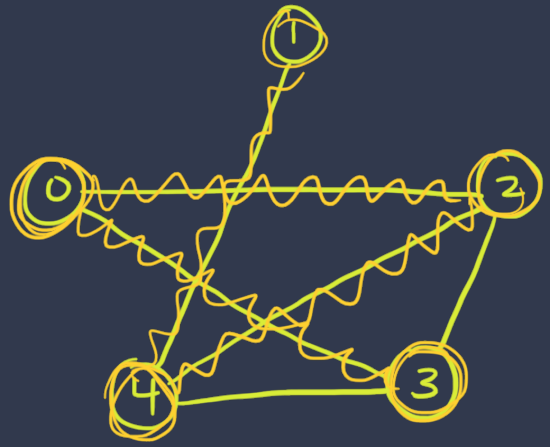
Trace of breadth-first search to find all paths from 0

- Note that unlike DFS, this is not typically a recursive algorithm
- This demonstrates finding *all* the shortest paths (all the paths from 0 to each of the other vertices)
- Once the algorithm is complete, you can determine if a path from the source to a specific target exists by examining the **marked** array…
- ...and you can determine the actual path by examining the **edgeTo** array.
- Runtime: **O(n + m)**

Picture from [1]

Example 3. Do BFS on the following graph starting at vertex 0. Compare the paths found from 0 to 3 using DFS and using BFS.

0	2, 3
1	4
2	0, 3, 4
3	0, 2, 4
4	1, 2, 3

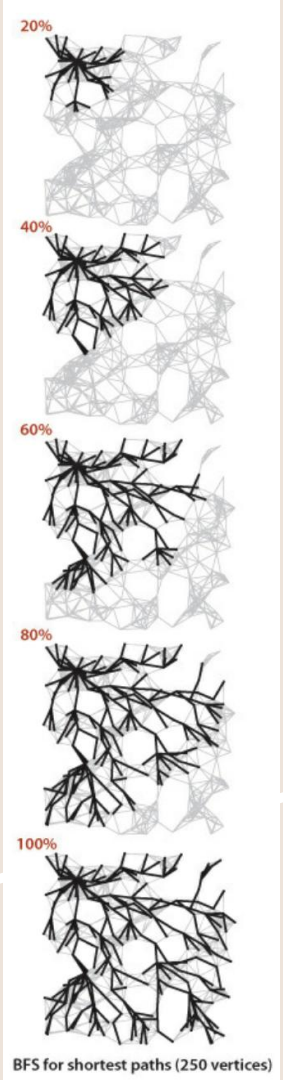


marked	
0	T
1	T
2	T
3	T
4	T

edgeTo	
0	-1
1	4
2	0
3	0
4	2

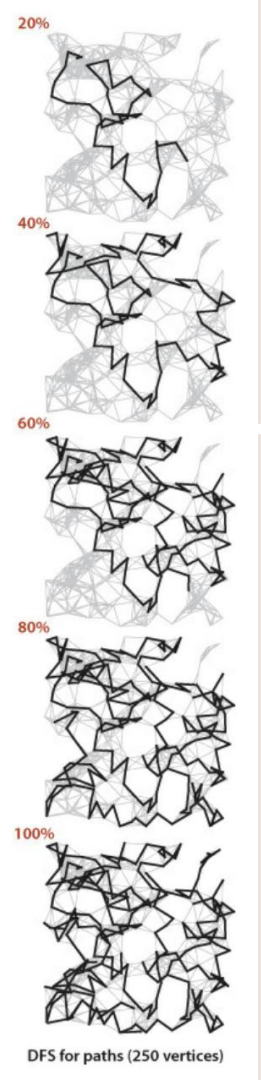
0 - 2 - 3 - 4 - 1

BFS vs. DFS



- DFS goes to the end of a path before coming back to an intersection
- BFS tries multiple paths on edge at a time

- Given an adjacency list representation of a graph with **n** vertices and **m** edges, the runtime for both is: **O(n + m)**



Pictures from [2]

References

[1] Sedgewick and Wayne
[2] Tamassia and Goodrich
[3] https://www.streamfinancial.com.au/the-myth-of-theseus-and-the-minotaur-lessons-from-the-ancient-greeks/

BFS: 1-2-3-4-7-5-6
DFS: 1-2-4-7-3-5-6
pre-order: 1-2-4-7-3-5-6
post-order: 4-7-2-5-6-3-1
in-order: 4-2-7-1-5-3-6