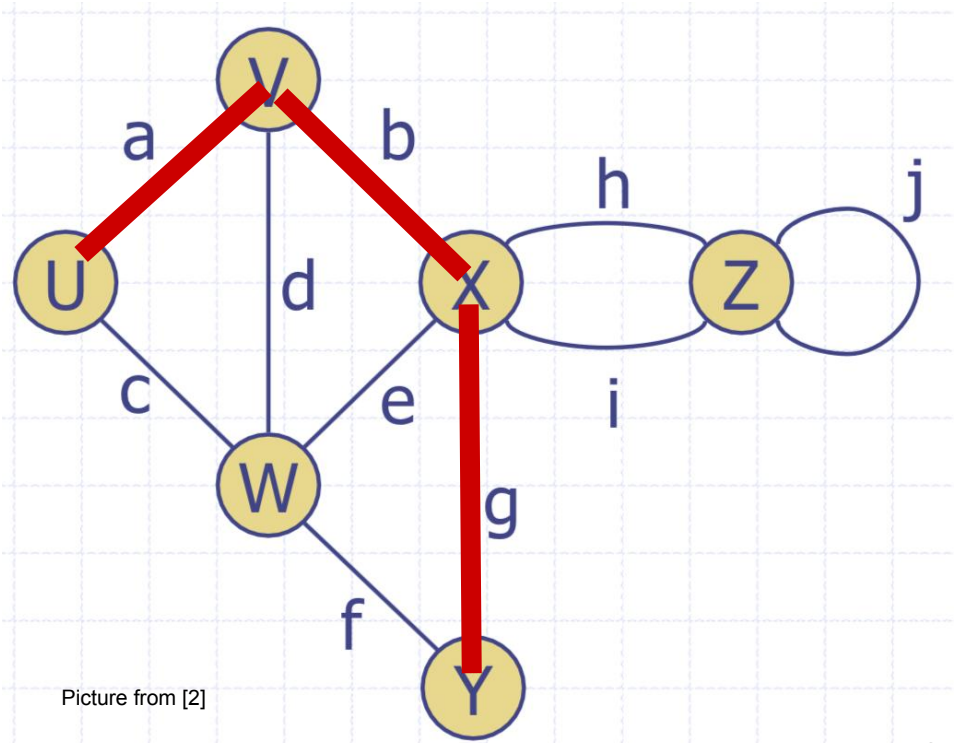
Graphs II

Part 1: DFS
Part 2: BFS

Part 1: Depth-First Search

More Terminology...

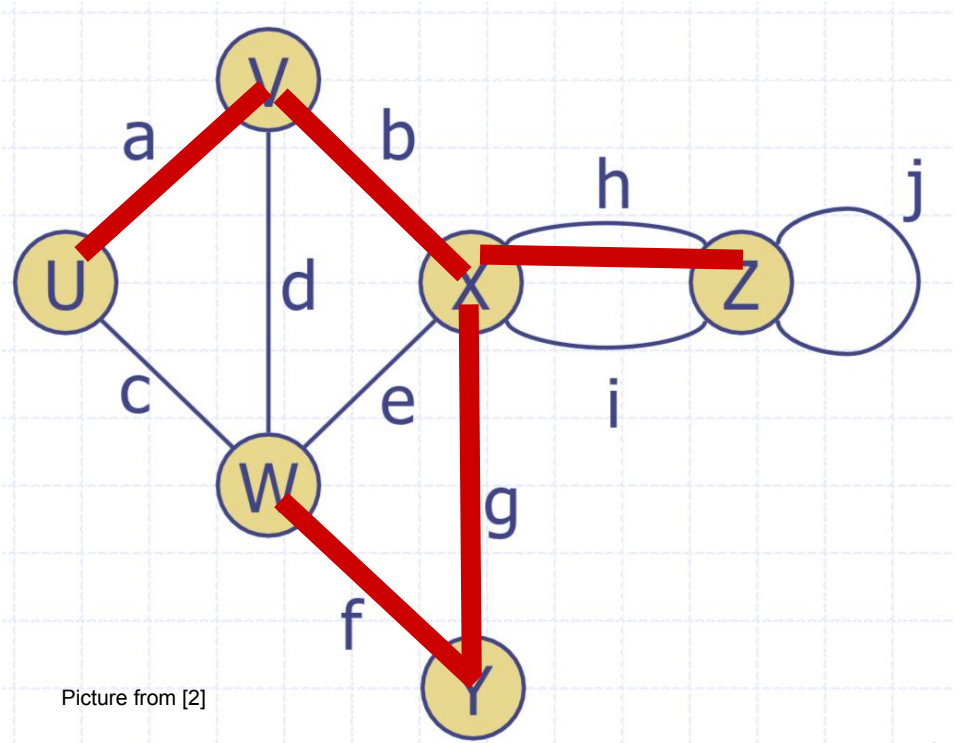
- A **subgraph** of a graph **G** is a graph made up of a subset of the vertices of **G** and a subset of the edges of **G**.
- EX: **S** = ({U, V, X, Y}, {a, b, g})



Picture from [2]

More Terminology...

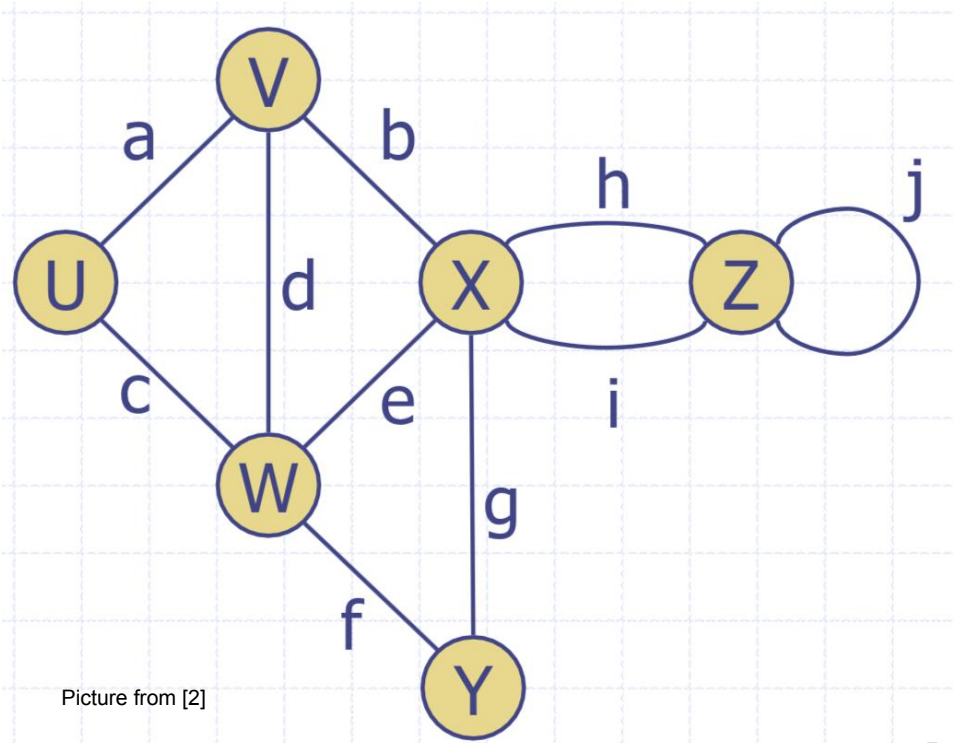
- A **subgraph** of a graph **G** is a graph made up of a subset of the vertices of **G** and a subset of the edges of **G**.
- EX: **S** = ({U, V, X, Y}, {a, b, g})
- A Spanning Subgraph of a graph is a subgraph that includes all the vertices of the original graph.
- EX: **S** = ({U, V, W, X, Y, Z},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



Picture from [2]

More Terminology...

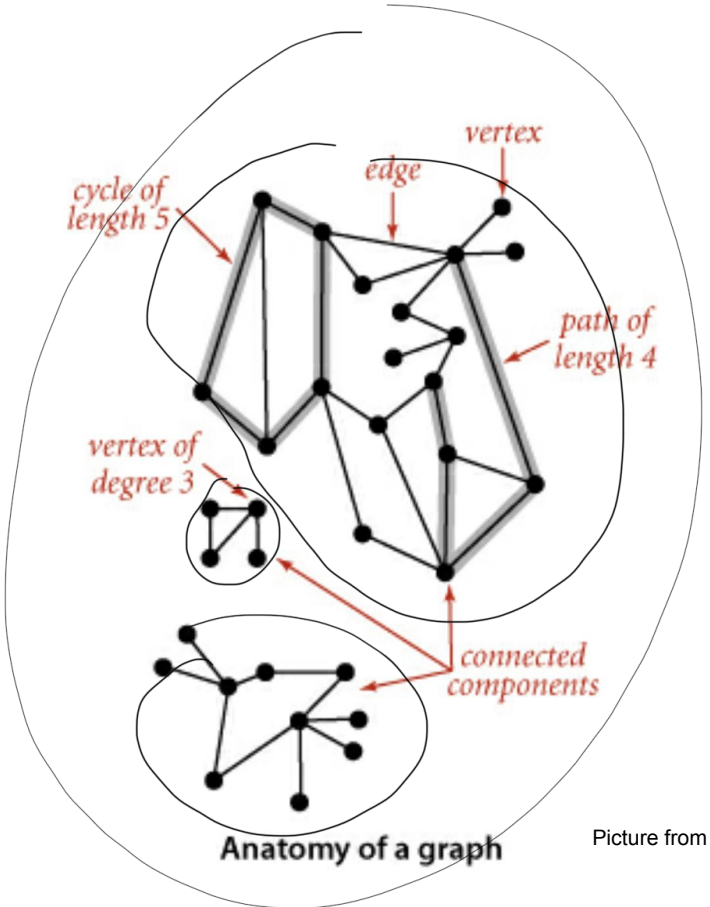
- A **connected** graph is a graph where there is a path from every vertex to every other vertex.



Picture from [2]

More Terminology...

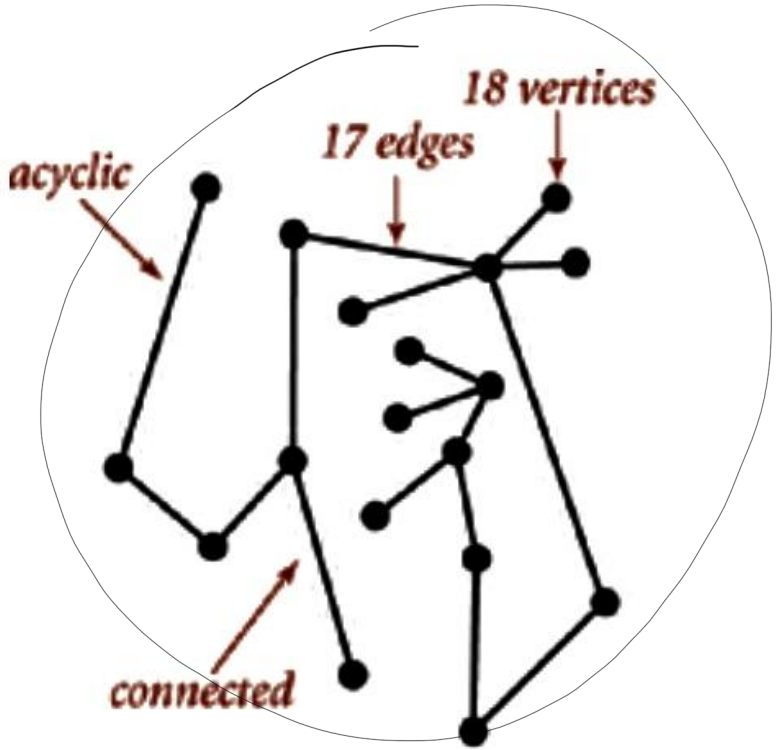
- A **connected** graph is a graph where there is a path from every vertex to every other vertex.
- A connected component of a graph **G** is a maximal connected subgraph of **G.**



Picture from [1]

More Terminology...

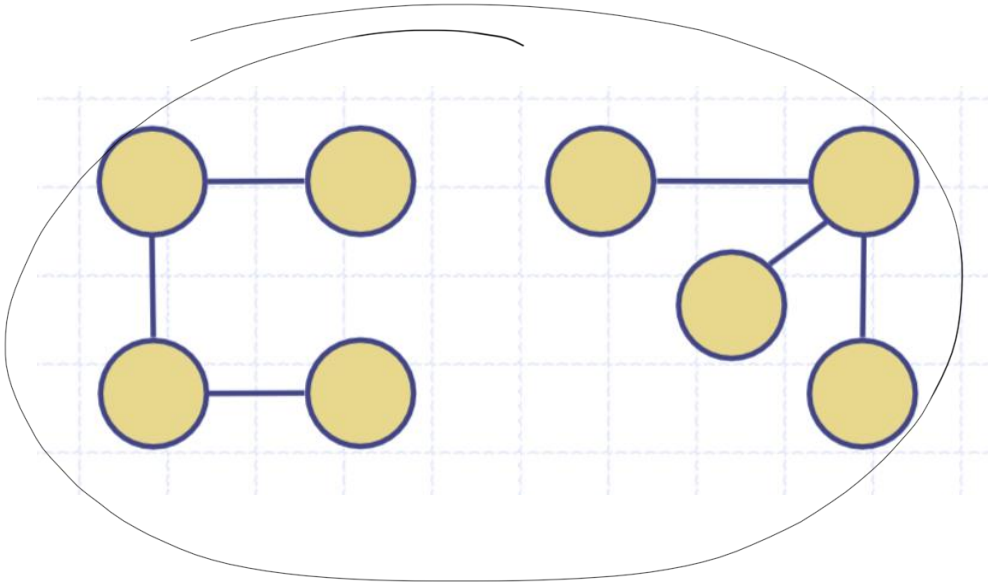
- A **tree** is an undirected graph **T** such that
  - **T** is connected
  - **T** has no cycles (acyclic)



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.



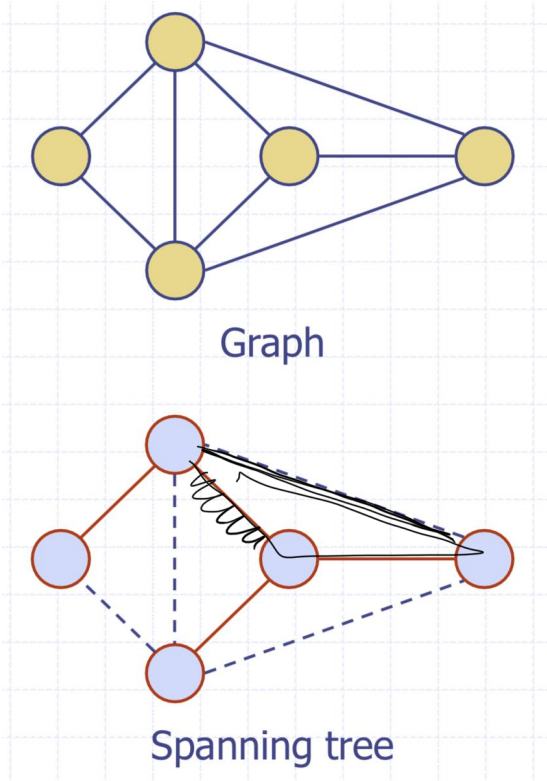
Picture from [2]

More Terminology...

Given a graph **G**, consider a subgraph of **G** that is *connected* and *acyclic* and includes all the vertices of **G**.

That subgraph is a Spanning tree.

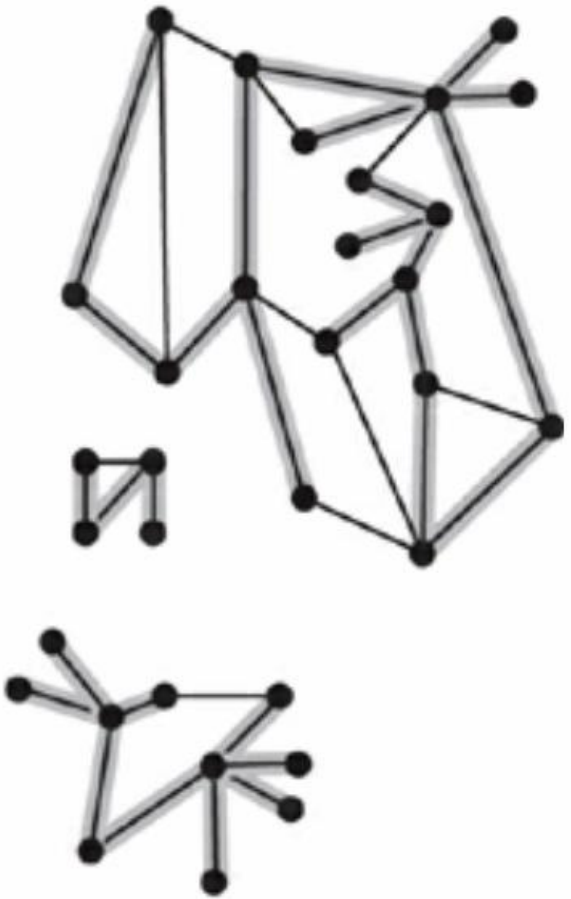
Is it possible for **G** to have more than one of these?



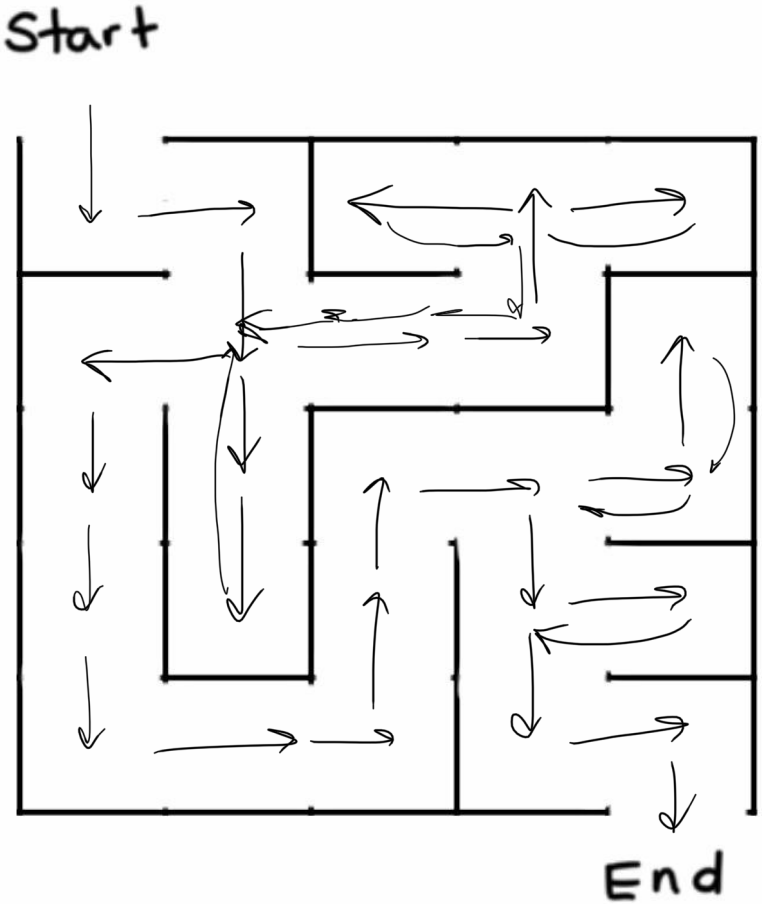
Picture from [2]

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]


Start
End

DFS: Depth-First Search

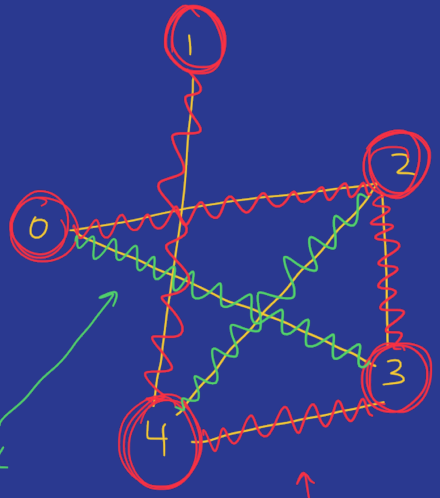
- A general technique for traversing a graph
- Visits all the vertices and edges of the graph
- Determines whether or not the graph is connected
- Computes the connected components of the graph
- Computes a spanning forest of the graph (spanning tree if the graph is connected)
- Useful as basis for solving 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 $dfs(G, w)$

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



back edges

discovery edges

visited	
0	T
1	T
2	T
3	T
4	T

Connected

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)**

```
Algorithm DFS
Input: Graph G = (V, E), represented as adjList[] and source vertex s
a[], b[] ← arrays
S ← an empty Stack
for each v ∈ V:
        a[v] ← false
        b[v] ← nil
S.push(s)
while S.size() ≠ 0:
        u ← S.pop()
        if a[u] holds false:
                a[u] ← true
                for each n in adjList[u]:
                        if a[n] holds false:
                                S.push(n)
                                b[n] ← u
```

a[]

0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T

b[]

0	nil
1	~~nil~~ 4
2	~~nil~~ ~~0~~ ~~3~~ 4
3	~~nil~~ 0
4	~~nil~~ 3

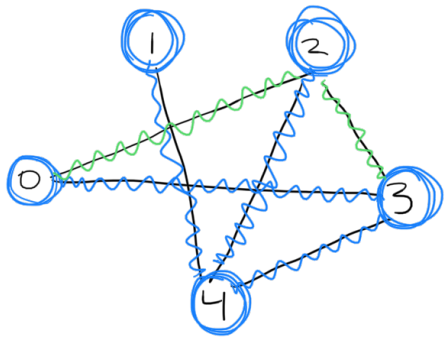
spanning tree

S: ~~0~~ 2 ~~3~~ 2 ~~4~~ ~~X~~ ~~2~~

u = ~~0~~ ~~3~~ ~~4~~ ~~2~~ 1

Input

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



Properties of DFS (G, v)

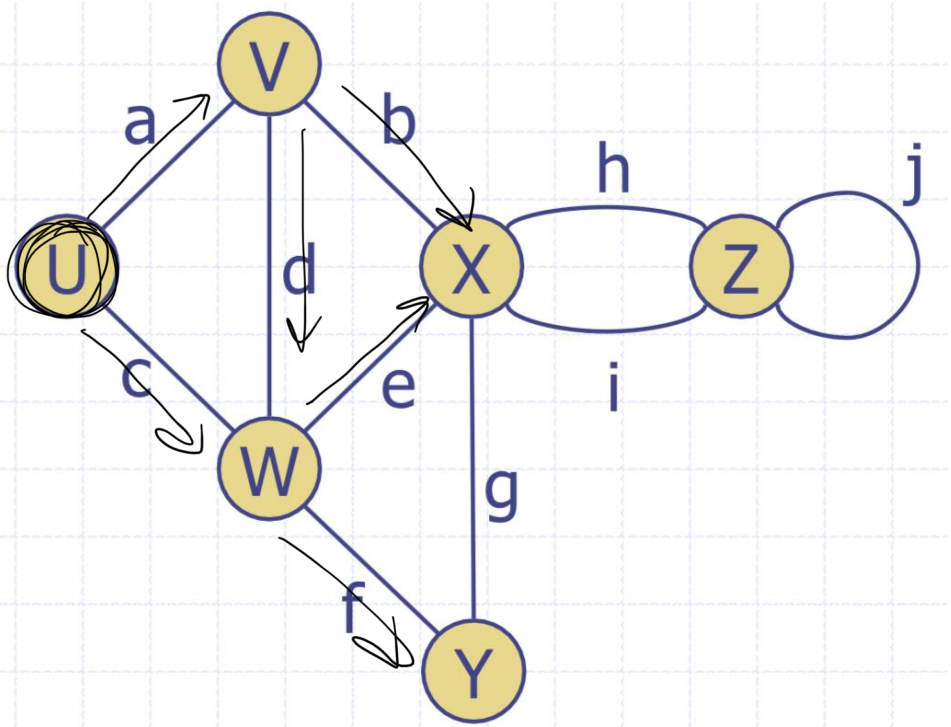
1. DFS(**G**, **v**) visits all edges and vertices in the connected component of **v.**
2. If the graph has multiple connected components, you can start DFS again with an unvisited node until everything has been visited.
3. The set of “discovery edges” labeled by DFS(**G**, **v**) form a spanning tree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

Part 2: Breadth-First Search

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?

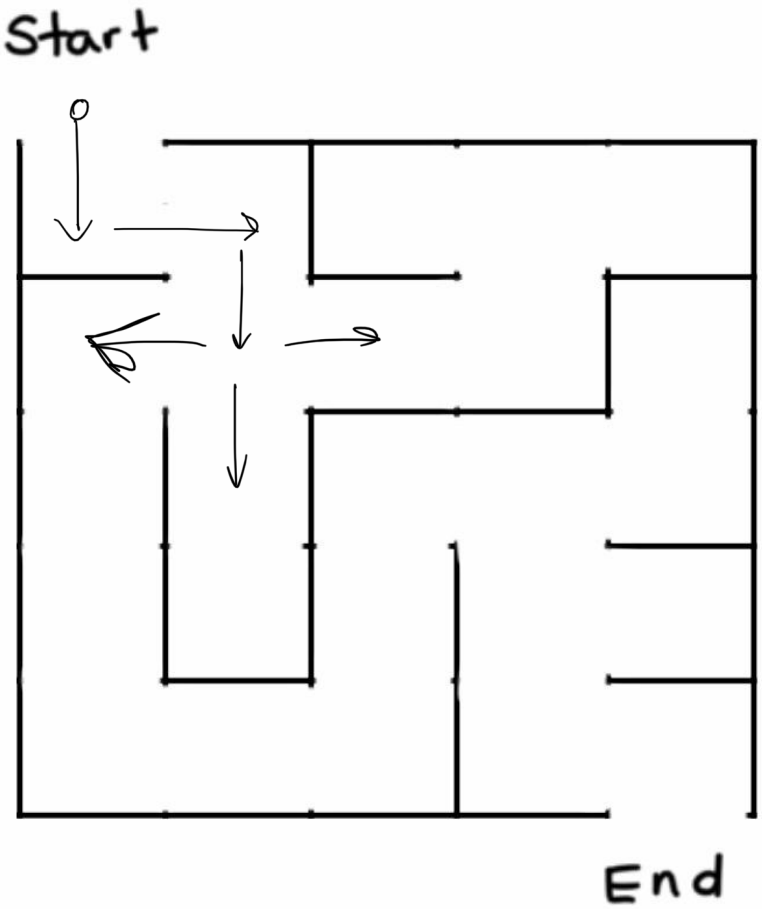


Picture from [2]

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!


Start
End

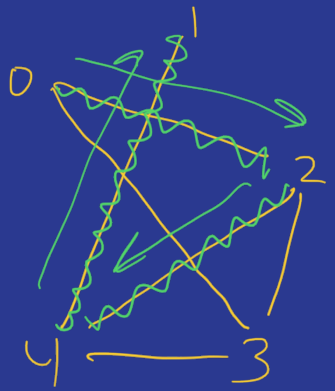
```
Algorithm BFS
Input: Graph G = (V, E), represented as adjList[] and source vertex s
a[], b[], c[] ← arrays
Q ← an empty Queue
for each v ∈ V:
        a[v] ← 'U'
        b[v] ← ∞
        c[v] ← nil
a[s] ← 'Q'
b[s] ← 0
Q.enqueue(s)
while Q.size() ≠ 0:
        v ← Q.dequeue()
        for each n in adjList[v]:
                if a[n] holds 'U':
                        a[n] ← 'Q'
                        b[n] ← b[v] + 1
                        c[n] ← v
                        Q.enqueue(n)
        a[v] ← 'V'
```

	a	b	c
0	V	0	nil
1	V	3	4
2	V	1	0
3	V	1	0
4	V	2	2

Q: ∅ 2 3 4 1

v = ∅ 2 3 4

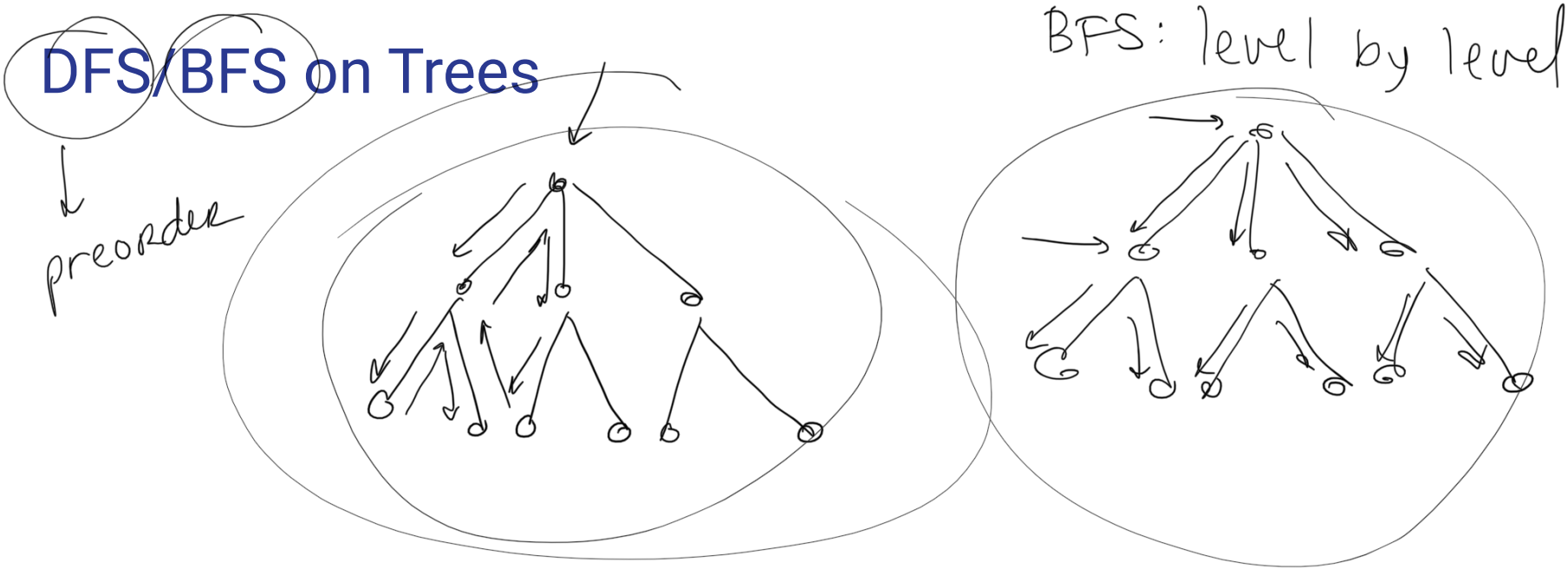
0	2, 3
1	4
2	0, 3, 4
3	0, 2, 4
4	1, 2, 3



Analysis of BFS(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)**

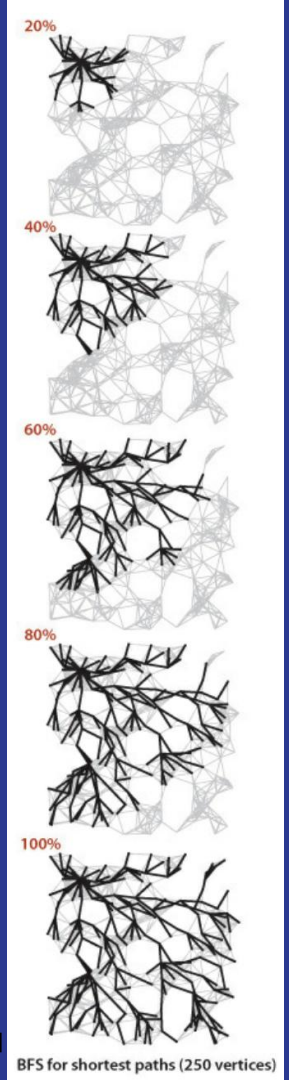
DFS/BFS on Trees



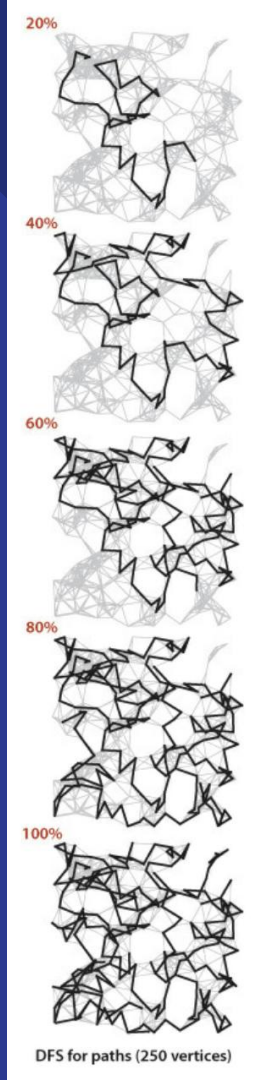
BFS vs. DFS

- 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)**



BFS for shortest paths (250 vertices)



DFS for paths (250 vertices)

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/