Graphs

Edge/Node Series

Path

Edge/Node Series

Path

- A series of distinct Nodes

Edge/Node Series

Path

- A series of distinct Nodes
- Adjacent Node pairs are connected by a **distinct** edge
    - From the first node of the series to the second in the case of directed edges

Edge/Node Series

Path

- A series of distinct Nodes
- Adjacent Node pairs are connected by a **distinct** edge
  - From the first node of the series to the second in the case of directed edges
- Sometimes the first Node in the series can be the last Node

Edge/Node Series

Path

- A series of distinct Nodes
- Adjacent Node pairs are connected by a **distinct** edge
  - From the first node of the series to the second in the case of directed edges
- Sometimes the first Node in the series can be the last Node

Tour

Edge/Node Series

Path

- A series of distinct Nodes
- Adjacent Node pairs are connected by a **distinct** edge
  - From the first node of the series to the second in the case of directed edges
- Sometimes the first Node in the series can be the last Node

Tour

- A series of Nodes

Edge/Node Series

Path

- A series of distinct Nodes
- Adjacent Node pairs are connected by a **distinct** edge
  - From the first node of the series to the second in the case of directed edges
- Sometimes the first Node in the series can be the last Node

Tour

- A series of Nodes
- Adjacent Node pairs are connected by an edge (no edge is used twice)
  - From the first node of the series to the second in the case of directed edges

Edge/Node Series (cont.)

Walk

Edge/Node Series (cont.)

Walk

- A series of Nodes

Edge/Node Series (cont.)

Walk

- A series of Nodes
- Adjacent Node pairs are connected by an edge
  - From the first node of the series to the second in the case of directed edges

Edge/Node Series (cont.)

Walk

- A series of Nodes
- Adjacent Node pairs are connected by an edge
  - From the first node of the series to the second in the case of directed edges
- No other Constraints

Edge/Node Series (cont.)

Walk

- A series of Nodes
- Adjacent Node pairs are connected by an edge
  - From the first node of the series to the second in the case of directed edges
- No other Constraints

Cycle

Edge/Node Series (cont.)

Walk

- A series of Nodes
- Adjacent Node pairs are connected by an edge
  - From the first node of the series to the second in the case of directed edges
- No other Constraints

Cycle

- Path where the first and last Node are equal

Graph Types

Types we will see

- Directed/Undirected
  - If any edge has a direction, the graph is directed
  - Directed mixed with undirected edges is not uncommon
  - Directed Graphs are sometimes referred to as Digraphs
- Weighted/Unweighted
  - If an edge has a weight, the graph is weighted
  - Normally all edges will be weighted or unweighted
  - Unweighted graphs are sometimes referred to as Unit Distance Graphs
- Connected
  - A graph where a path exists between all pairs of nodes
  - The direction of the edges for the paths does not typically matter

Graph Types (cont.)

- Acyclic
  - A graph with no cycles
- Tree
  - Undirected, Connected, Acyclic Graph
- Rooted Tree
  - Directed, Connected, Acyclic Graph
  - There exists some node (root) that can reach all other nodes using a path
  - Additionally, if the edges were consider undirected, no cycle would exist
- DAG
  - Directed, Acyclic Graph

Searching Unit Distance Graphs

Suppose we are looking for our keys in a house. (hopefully not a strangers house)

We typically “search” the house for our keys.

Do we search the same location twice? I do, but it never works.

There are two common search methods:

- BFS - “Let’s split up and search for clues!”
- DFS - “No. That’s how people die in horror movies.”

BFS

Breadth First Search

BFS

Breadth First Search (Closest Nodes First)

BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

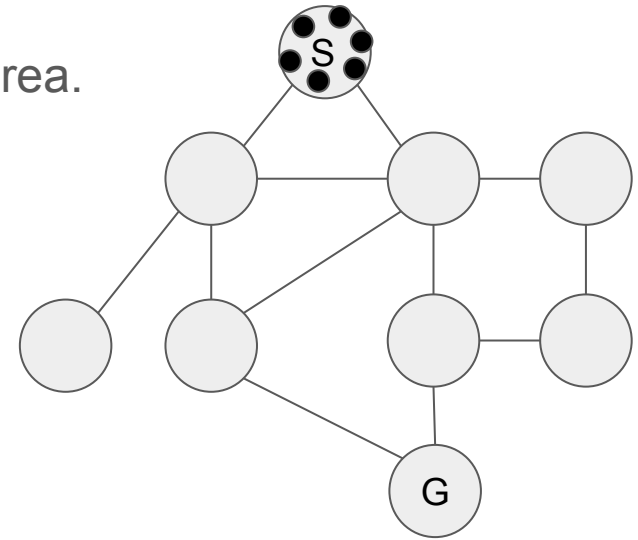
Think of it as groups of people sweeping an area.

BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.

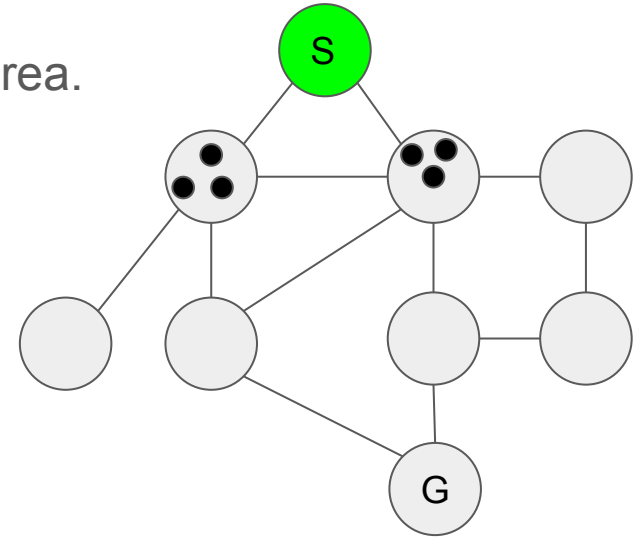


BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.

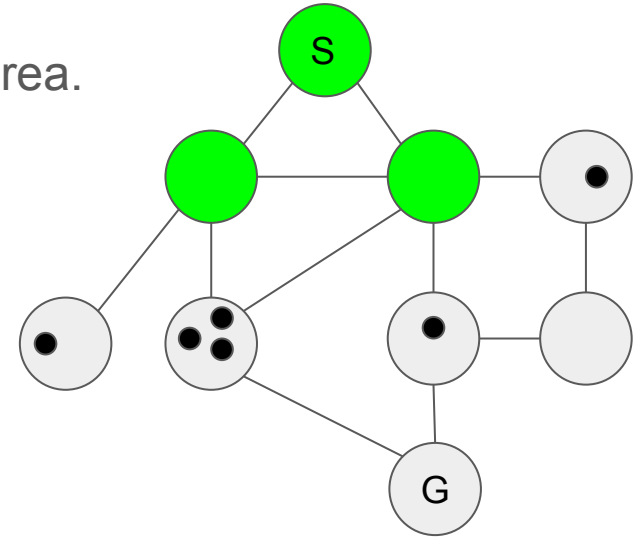


BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.

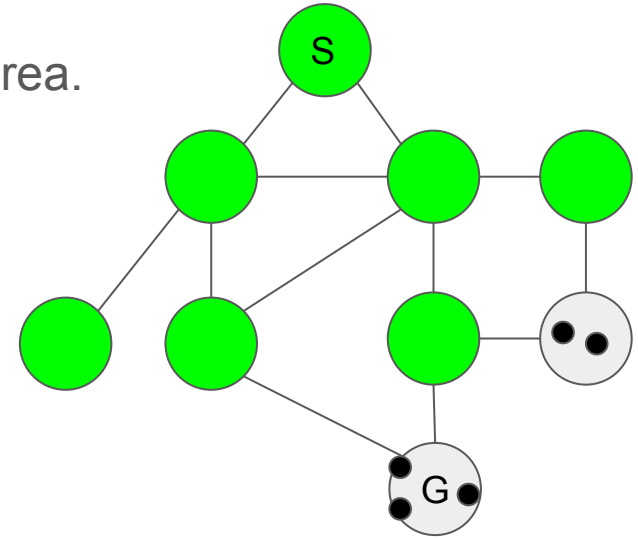


BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.

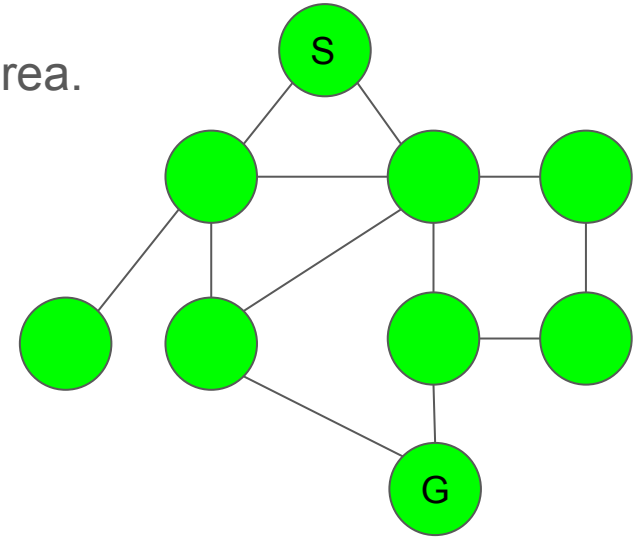


BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.



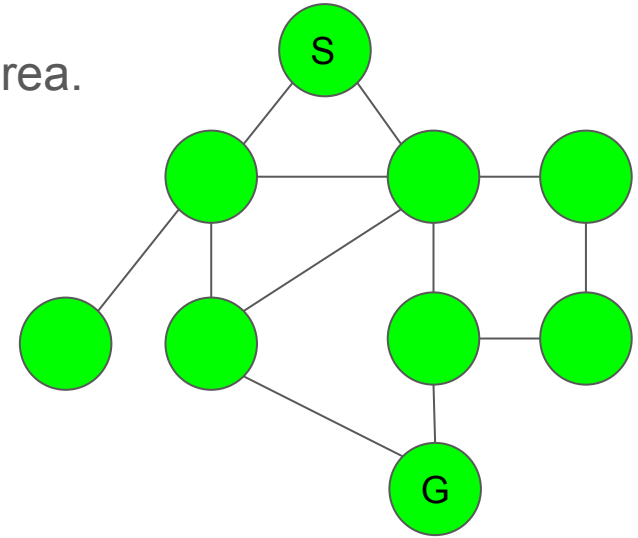
BFS

Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.

Somewhat unrealistic because,



BFS

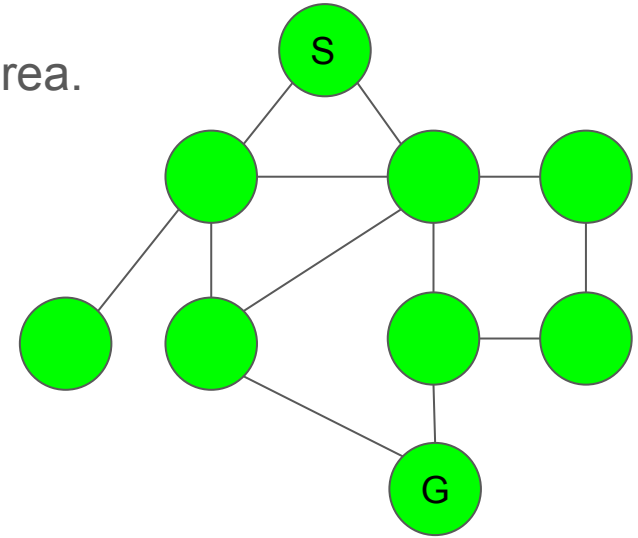
Breadth First Search (Closest Nodes First)

Search nodes in layers based on their distance from the start.

Think of it as groups of people sweeping an area.

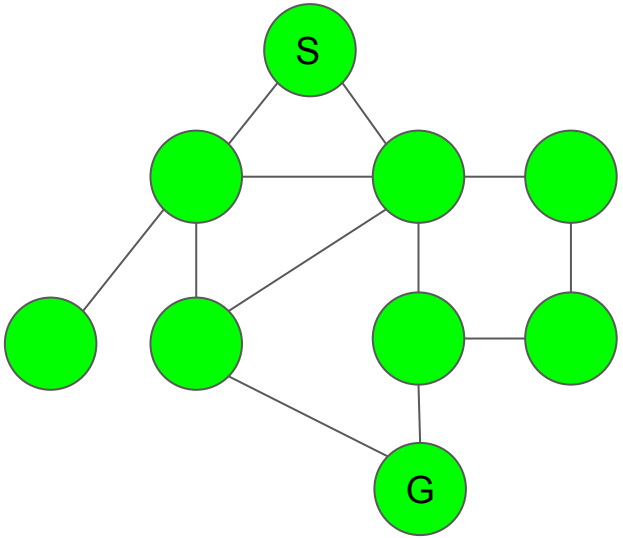
Somewhat unrealistic because,

- Typically not parallelized.
- Two different branches never merge.



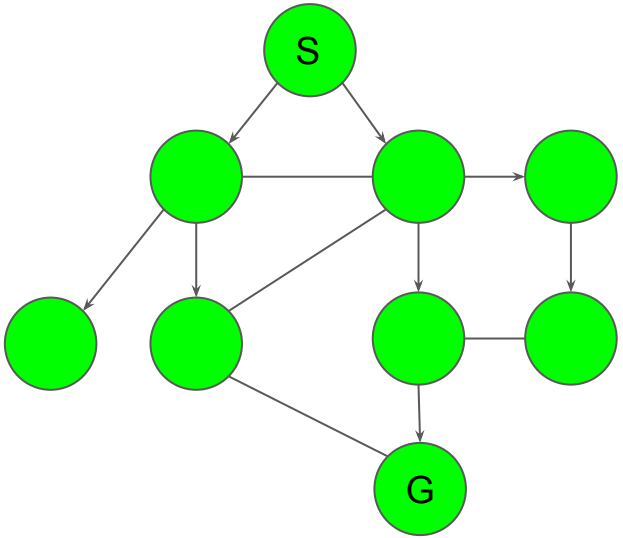
BFS (cont.)

We create a Tree of edges (the tree edges) in our search.



BFS (cont.)

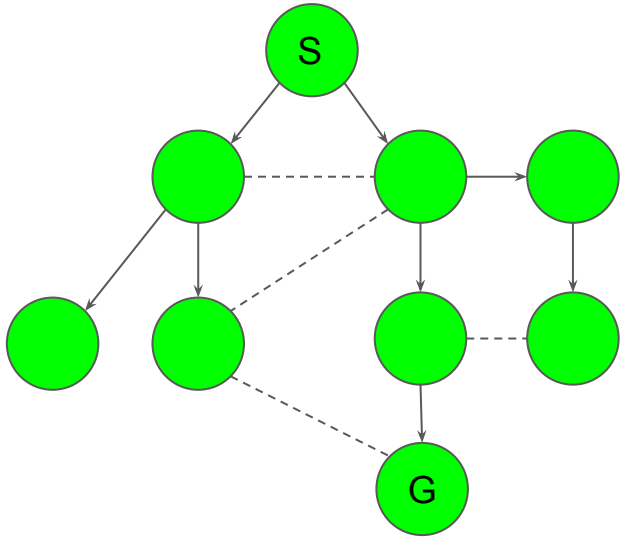
We create a Tree of edges (the tree edges) in our search.



BFS (cont.)

We create a Tree of edges (the tree edges) in our search.

Unused edges are considered either cross or back edges.

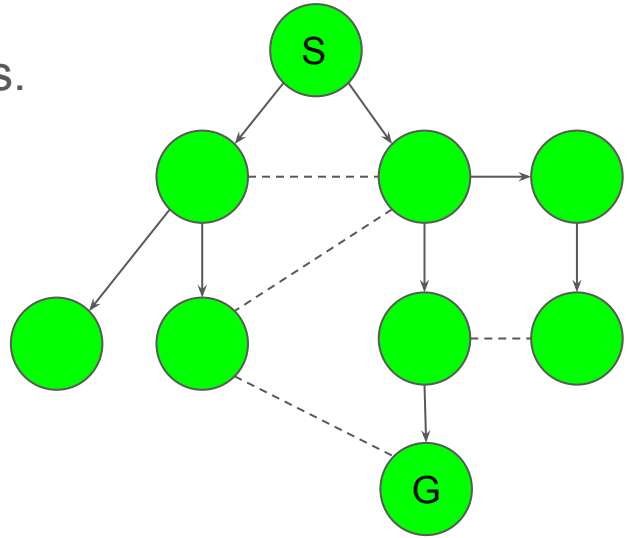


BFS (cont.)

We create a Tree of edges (the tree edges) in our search.

Unused edges are considered either cross or back edges.

Back edges are between nodes and ancestors.



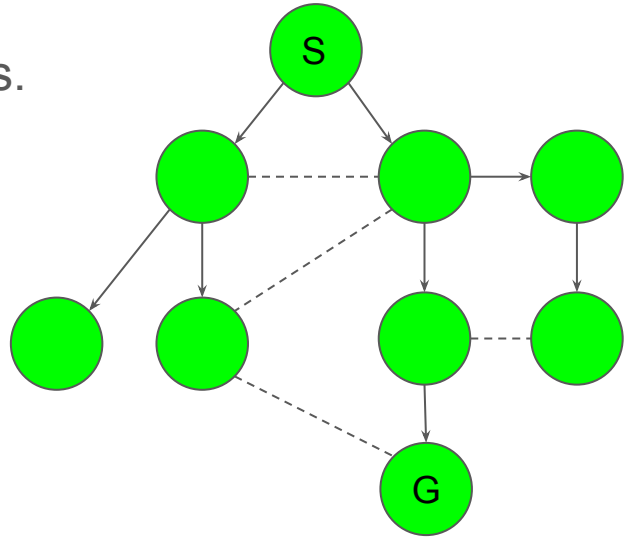
BFS (cont.)

We create a Tree of edges (the tree edges) in our search.

Unused edges are considered either cross or back edges.

Back edges are between nodes and ancestors.

Cross edges are other untaken edges.



BFS (cont.)

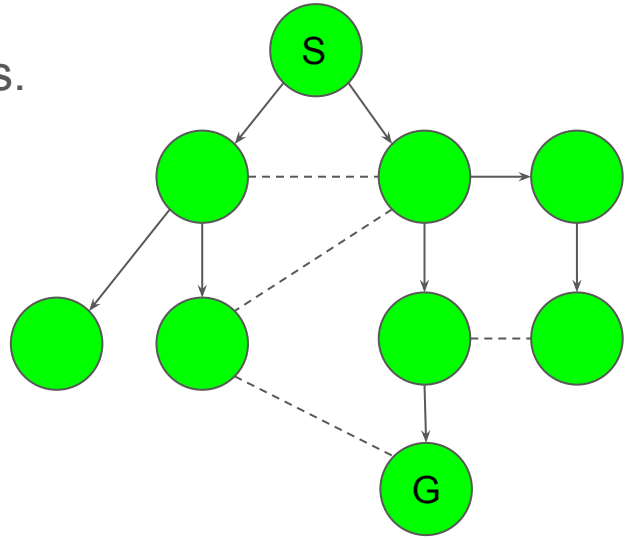
We create a Tree of edges (the tree edges) in our search.

Unused edges are considered either cross or back edges.

Back edges are between nodes and ancestors.

Cross edges are other untaken edges.

Undirected graphs without multi edges do not generate back edges in a BFS.



BFS Algorithm

```
Create an initially empty list of visited nodes
Create an initially empty queue of to process nodes
Add the start node to the visited nodes and the to process nodes
While Not Done
   Let the current node be the front of the to process nodes
   Remove the front of the to process nodes
   // DO WORK HERE (sometimes after or in the For loop)
   For all new node adjacent to current
      If visited nodes does not contain the new node
         Add the new node to the visited nodes
         Add the new node to the back of the to process nodes
      End If
   End For
End While
```

BFS Analysis

How many times will the While loop execute in the worst case?

How many times will the inner For loop execute in the worst case?

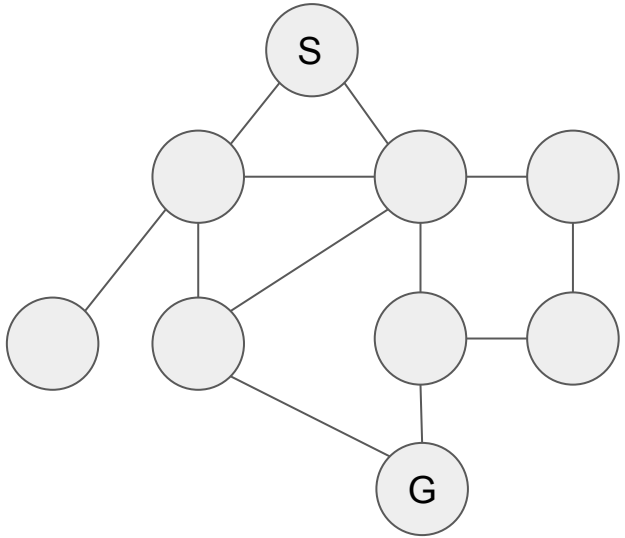
BFS Usage

- Shortest path in a unit graph
- **Diameter of a tree finding**
- Topological ordering
- Cycle detection
- 2 Coloring
- Connectivity check

Changing the queue to a priority queue enables using an algorithm called Dijkstra’s

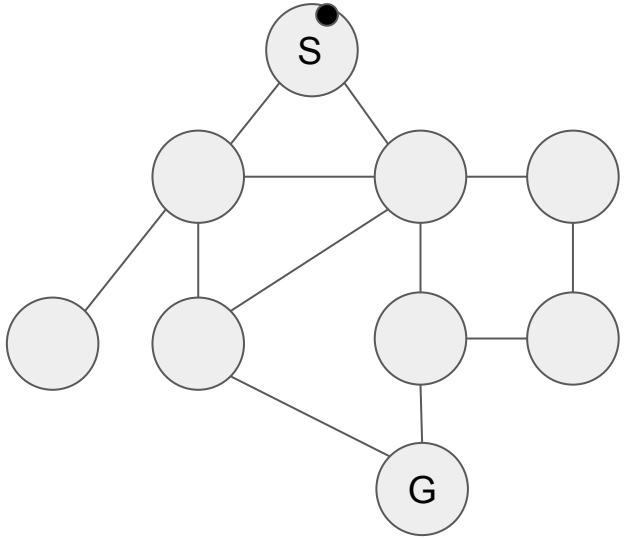
DFS

Depth First Search (Maze Solving)



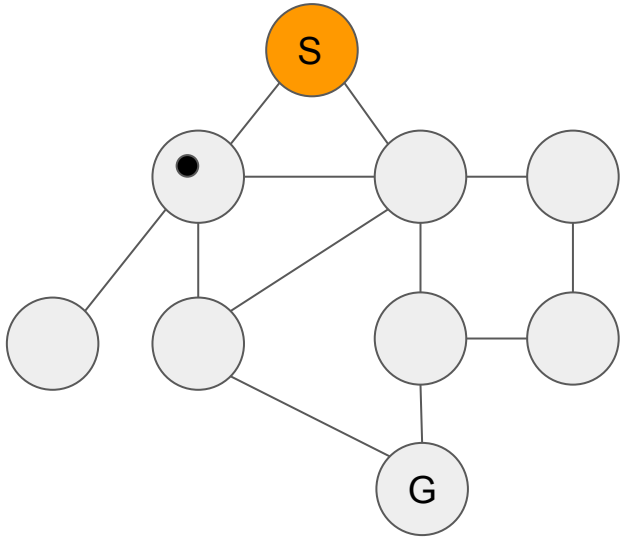
DFS

Depth First Search (Maze Solving)



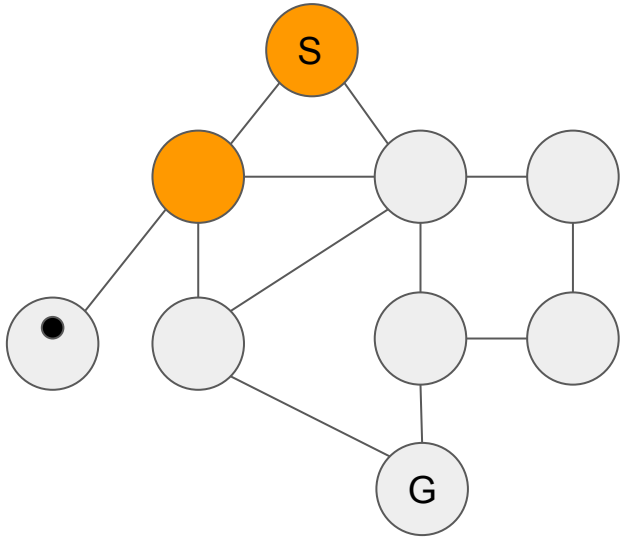
DFS

Depth First Search (Maze Solving)



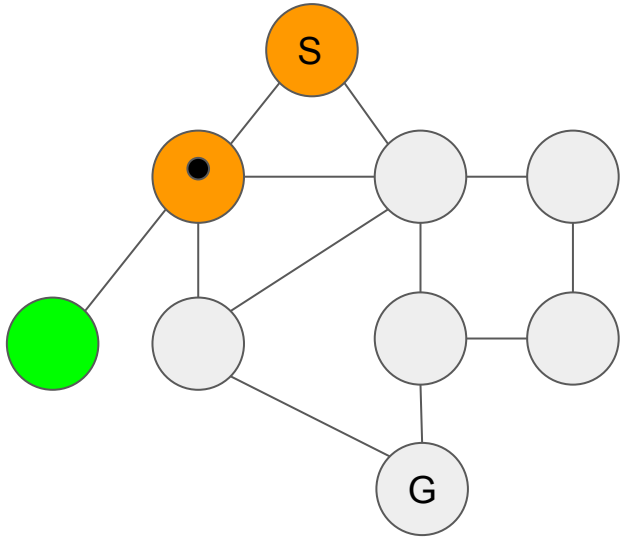
DFS

Depth First Search (Maze Solving)



DFS

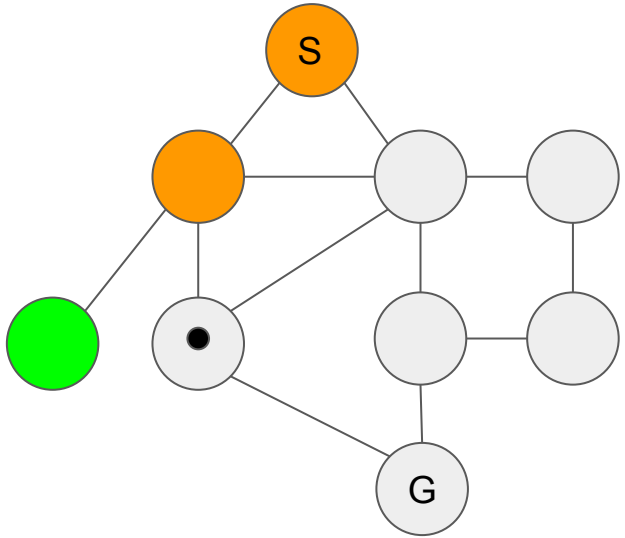
Depth First Search (Maze Solving)



DFS

Depth First Search (Maze Solving)

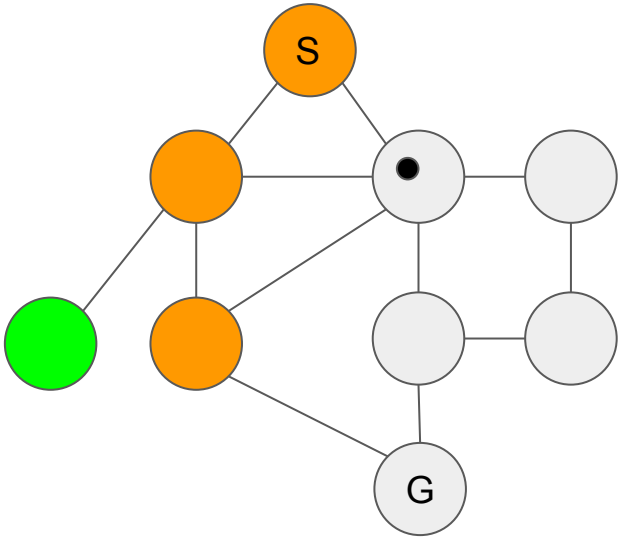
Almost done.



DFS

Depth First Search (Maze Solving)

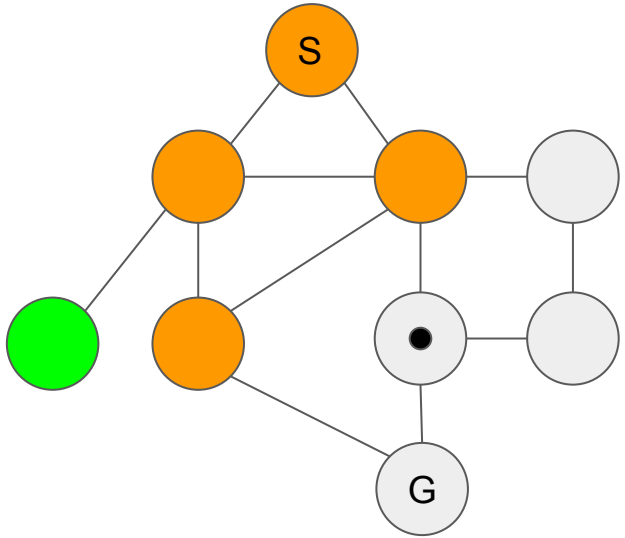
...



DFS

Depth First Search (Maze Solving)

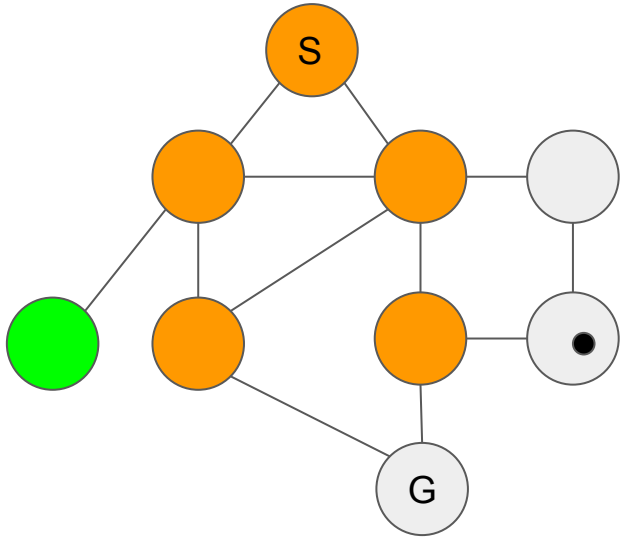
Almost There!!!



DFS

Depth First Search (Maze Solving)

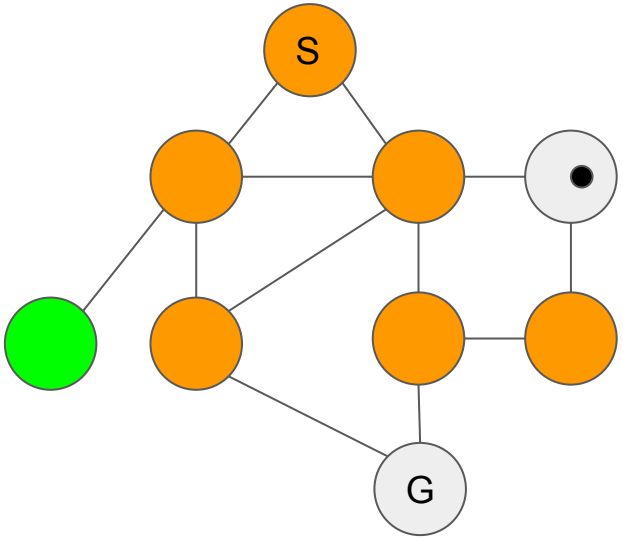
...



DFS

Depth First Search (Maze Solving)

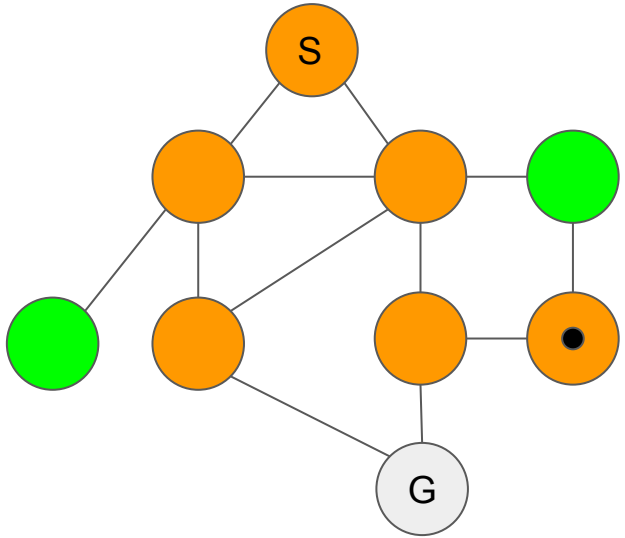
...



DFS

Depth First Search (Maze Solving)

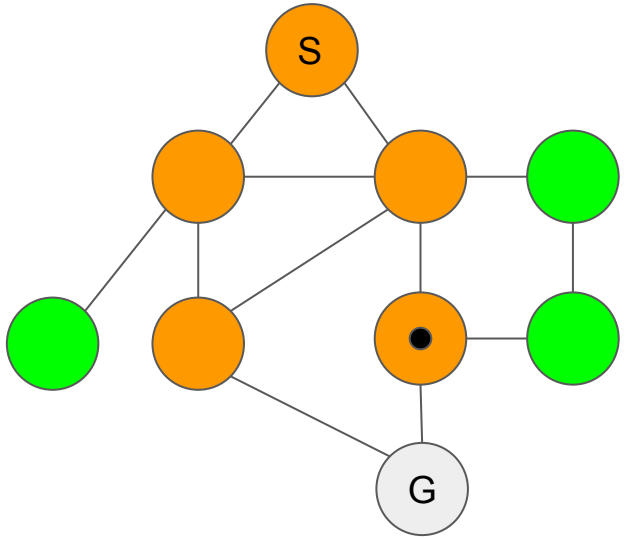
...



DFS

Depth First Search (Maze Solving)

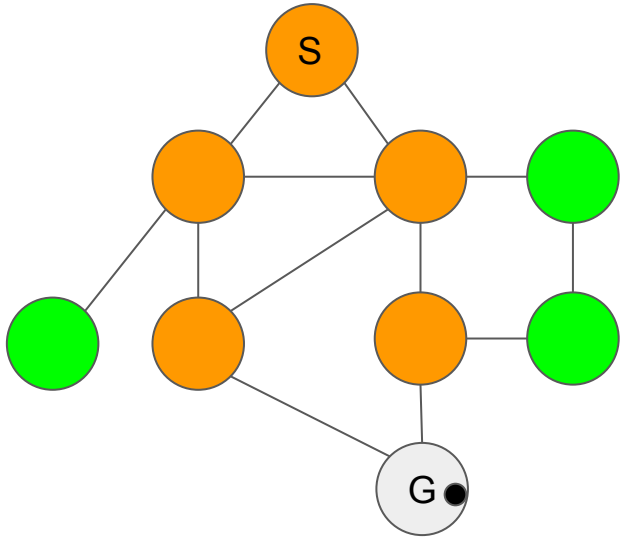
...



DFS

Depth First Search (Maze Solving)

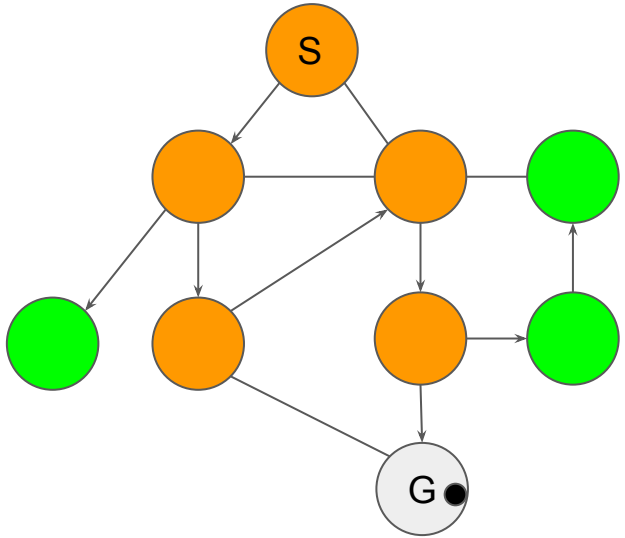
Finally!



DFS

Depth First Search (Maze Solving)

This also produces a set of Tree Edges.

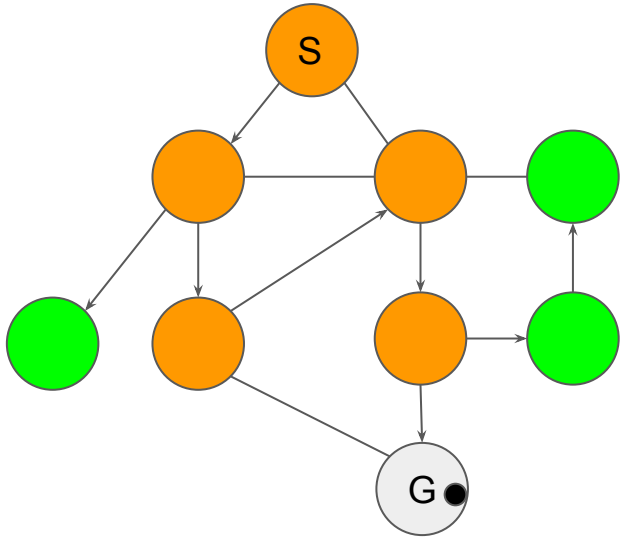


DFS

Depth First Search (Maze Solving)

This also produces a set of Tree Edges.

Unused edges are mostly back edges.

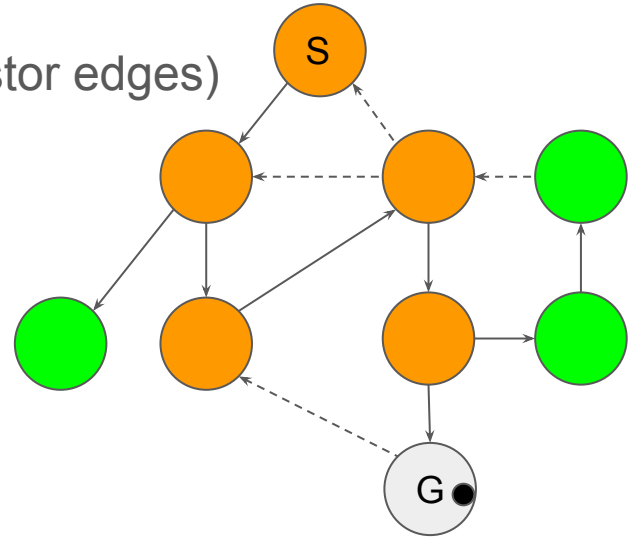


DFS

Depth First Search (Maze Solving)

This also produces a set of Tree Edges.

Unused edges are mostly back edges. (ancestor edges)



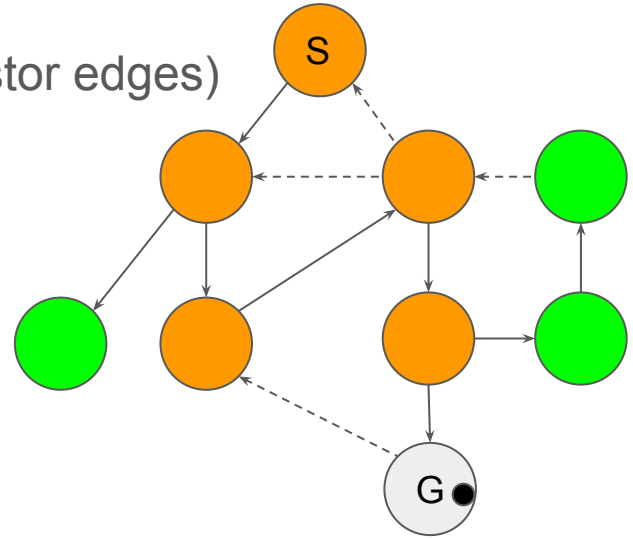
DFS

Depth First Search (Maze Solving)

This also produces a set of Tree Edges.

Unused edges are mostly back edges. (ancestor edges)

Cross edges can appear in directed graphs.



DFS

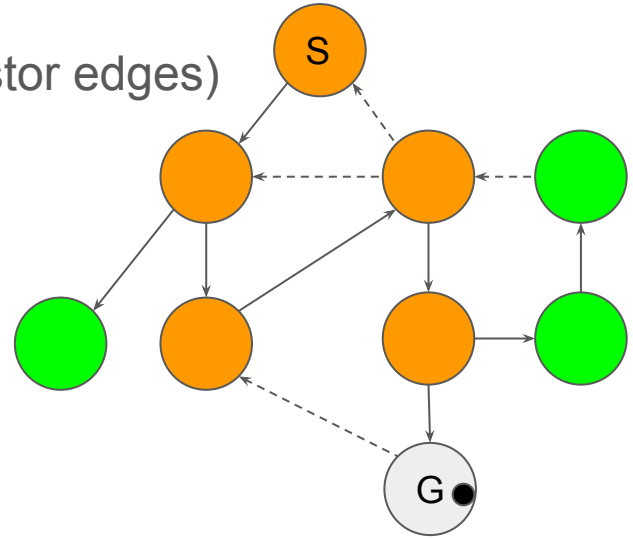
Depth First Search (Maze Solving)

This also produces a set of Tree Edges.

Unused edges are mostly back edges. (ancestor edges)

Cross edges can appear in directed graphs.

We are required to know the orange nodes.



DFS

Depth First Search (Maze Solving)

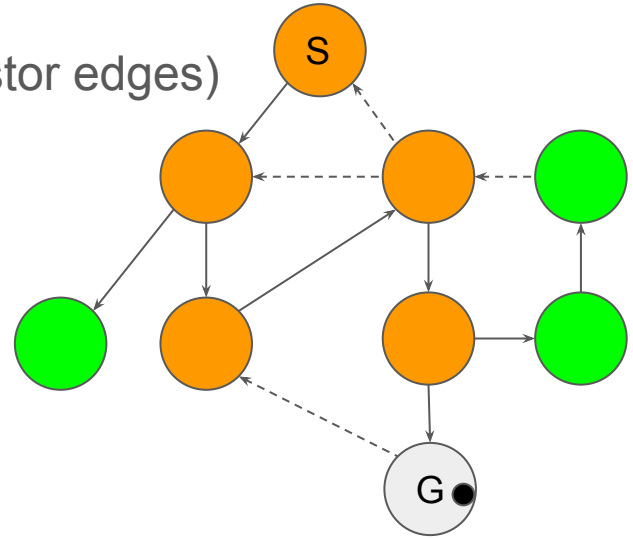
This also produces a set of Tree Edges.

Unused edges are mostly back edges. (ancestor edges)

Cross edges can appear in directed graphs.

We are required to know the orange nodes.

Keep track of them using a stack.



DFS Algorithm

```
Create an initially empty set of visited nodes
DFS(start node)
Function DFS(current node)
   // DO WORK HERE (sometimes in or after the For loop)
   Add current node to the set of visited nodes
   For all next node adjacent to the current node
      If next node is not in the set of visited nodes
         DFS(next node)
      End If
   End For
End Function
```

DFS Analysis

How many times will DFS be called?

How many times will the inner for loop be called?

What is the worst case depth of the program stack?

DFS Usage

- Low Link
  - Strongly Connected Component Finding
  - Bridge (cut edge) Finding
  - Articulation point (cut node) Finding
- Maze solving
- Topological ordering
- Cycle detection
- 2 Coloring
- Connectivity check

Problem Talk

Suppose we have a famous actor we want to get in contact with through a series of friends. We scraped the data from a popular social media website, and we have a gigantic network of friendships.

Problem Talk

Suppose we have a famous actor we want to get in contact with through a series of friends. We scraped the data from a popular social media website, and we have a gigantic network of friendships.

How could we quickly determine the shortest sequence of friendships to reach the famous actor?

Problem Talk

Suppose we have a famous actor we want to get in contact with through a series of friends. We scraped the data from a popular social media website, and we have a gigantic network of friendships.

How could we quickly determine the shortest sequence of friendships to reach the famous actor?

BFS is pretty good, but the data is quite large and suppose we want to do this for many pairs of people.

Problem Talk

Suppose we have a famous actor we want to get in contact with through a series of friends. We scraped the data from a popular social media website, and we have a gigantic network of friendships.

How could we quickly determine the shortest sequence of friendships to reach the famous actor?

BFS is pretty good, but the data is quite large and suppose we want to do this for many pairs of people.

Divide and conquer can give slight performance improvements depending on the network structure.

Problem Talk

Suppose we have a famous actor we want to get in contact with through a series of friends. We scraped the data from a popular social media website, and we have a gigantic network of friendships.

How could we quickly determine the shortest sequence of friendships to reach the famous actor?

BFS is pretty good, but the data is quite large and suppose we want to do this for many pairs of people.

Divide and conquer can give slight performance improvements depending on the network structure. Search from both ends and meet in the middle.

Another Problem

Class scheduling is tricky. Suppose we have many classes in our degree program. Some classes have prerequisites. We want to know if we can actually take all the classes while abiding by the class prerequisite. We also want to know what is the fastest we can graduate (fewest semesters) while abiding by the class prerequisite.

Another Problem

Class scheduling is tricky. Suppose we have many classes in our degree program. Some classes have prerequisites. We want to know if we can actually take all the classes while abiding by the class prerequisite. We also want to know what is the fastest we can graduate (fewest semesters) while abiding by the class prerequisite.

Why might it not be possible?

Another Problem

Class scheduling is tricky. Suppose we have many classes in our degree program. Some classes have prerequisites. We want to know if we can actually take all the classes while abiding by the class prerequisite. We also want to know what is the fastest we can graduate (fewest semesters) while abiding by the class prerequisite.

Why might it not be possible?

The prerequisite graph needs to be a DAG.

Another Problem

Class scheduling is tricky. Suppose we have many classes in our degree program. Some classes have prerequisites. We want to know if we can actually take all the classes while abiding by the class prerequisite. We also want to know what is the fastest we can graduate (fewest semesters) while abiding by the class prerequisite.

Why might it not be possible?

The prerequisite graph needs to be a DAG.

We can find the minimum by using a topological ordering algorithm.