Introduction to Pathfinding I

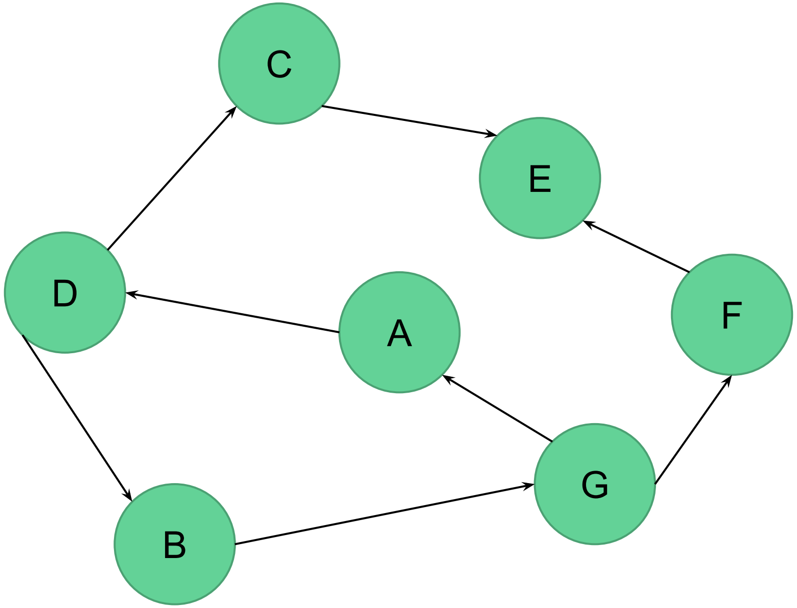
BFS and DFS

Ryan Toner

Basic Graph Pathfinding - Learning Objective

- Graphs are a data structure that consist of the following:
    - We define the graph G in the following way
    - $V(G) = \{v_1, v_2 \ldots v_k\}$, a set of *k* vertices, or nodes
    - $E(G) = \{e_1, e_2 \ldots e_j\}$, a set of *j* edges, where e_i defines a tuple $e_i = (v_a, v_b)$, the edge between two vertices V_a and V_b
- A path in the graph may be thought of simply as a route existing between two nodes by following all legal edge connections
    - They should not repeat vertices or edges (no cycles)
    - They should follow an ordered direction from start ➔ finish
- The objective of this lecture is to introduce the most fundamental graph searches - DFS and BFS.

Example Graph



- This is a directed graph.
- V(G) = {A,B,C,D,E,F,G}
- E(G) = {(A,D),(B,G),(C,E),(D,B),(D,C),(F,E),(G,A),(G,F)}
- Suppose we would like to know if we could find a path given an arbitrary starting and ending node.

Graph Search - DFS

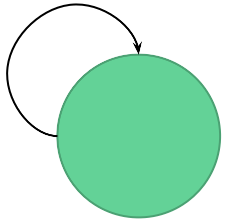
- The first algorithm for graph search we will look at is Depth-First search (DFS).
- This search will take a source node (root) and will find paths to an end node (terminus).
- This search can easily be extended to find all possible paths in the graph starting from the start (simply by not specifying an ending node).

High-Level DFS Algorithm:

1. Push root node to stack
2. Pop the top node off the stack
    a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node*
4. Push all unmarked neighbors of the top node to the stack
5. Repeat step 2-5, until stack is empty or terminus point is found

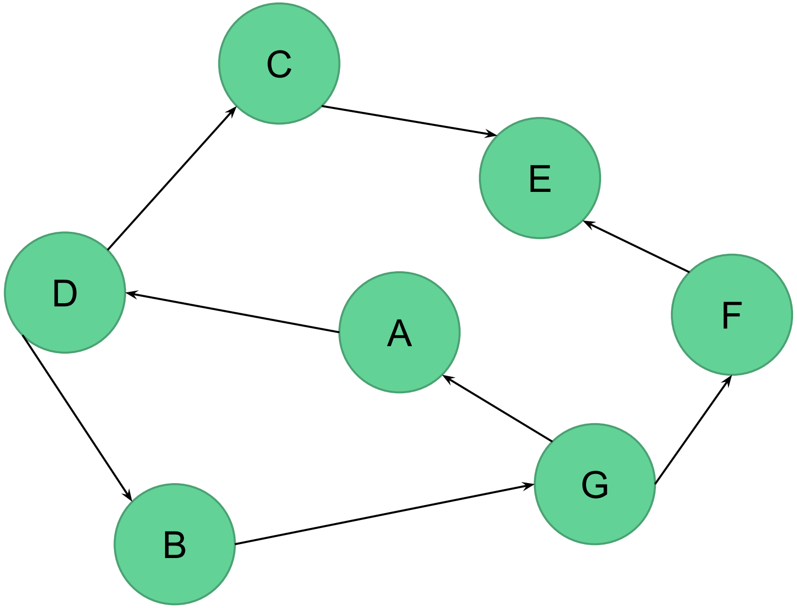
*A Note about marking:

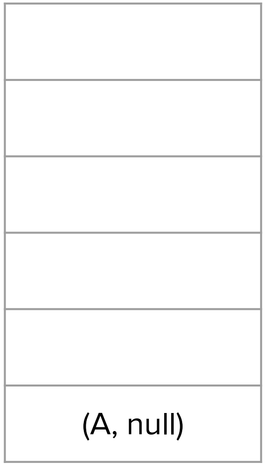
- You can represent marking with a hashset for efficiency e.g. marked items are in the set
- Marking the node at step 3 may save time in graphs with self-directed edges, where the edge connects a node to itself.



Graph Search - DFS

- Use DFS with a stack and marked list to find a path from A to F
- The stack will contain tuples of the form (node, path), where node indicates the node we have reached, and path indicates an augmenting path of how we got there.
- Start by placing the starting node on the stack.
    - Push (A, null) onto the stack. The null value indicates that this is the starting point.

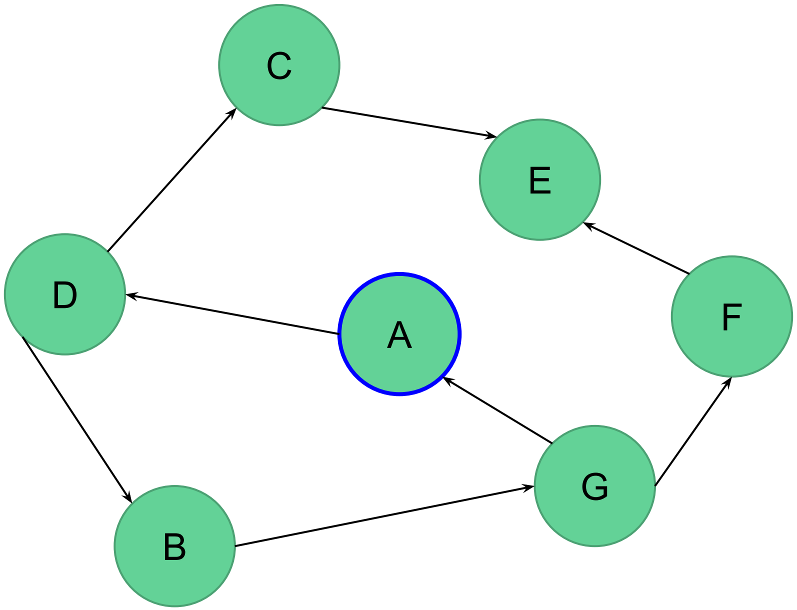


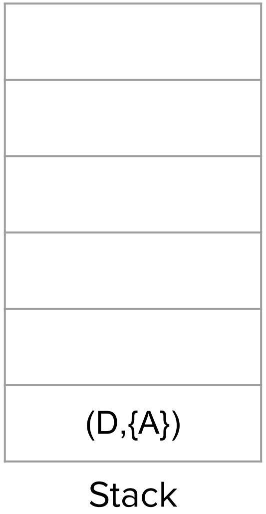


Stack

Graph Search - DFS

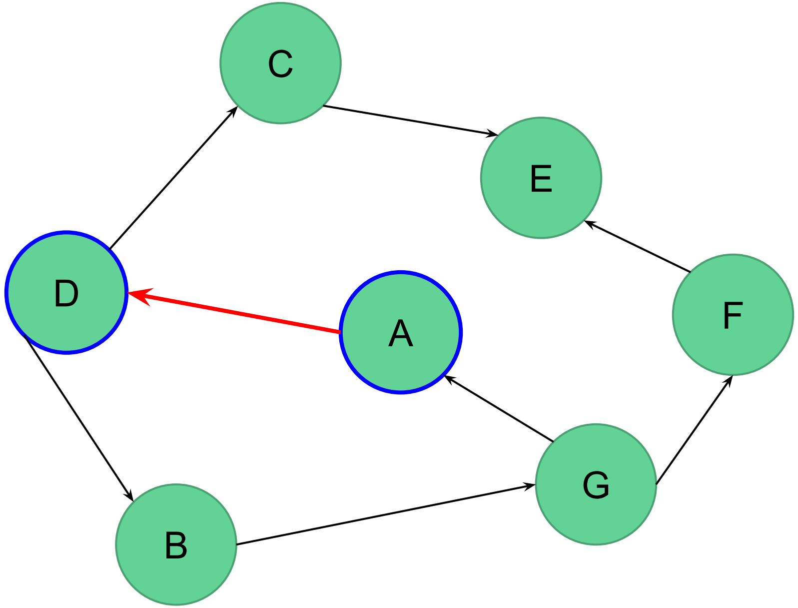
- Pop (A, null) from the top of the stack.
- Mark A (blue outline) to ensure it is only processed once.
- Next, process the unmarked neighbors of A.
- The *neighbors* of A are:
  - **all** edges **e** such that **e=(A,_)**, where **_** is any connected node.
- In this case, we only find D.
- Push (D, {A}) to the stack.

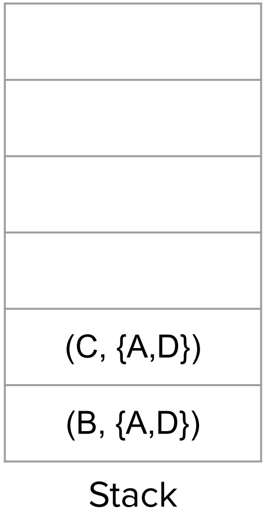




Graph Search - DFS

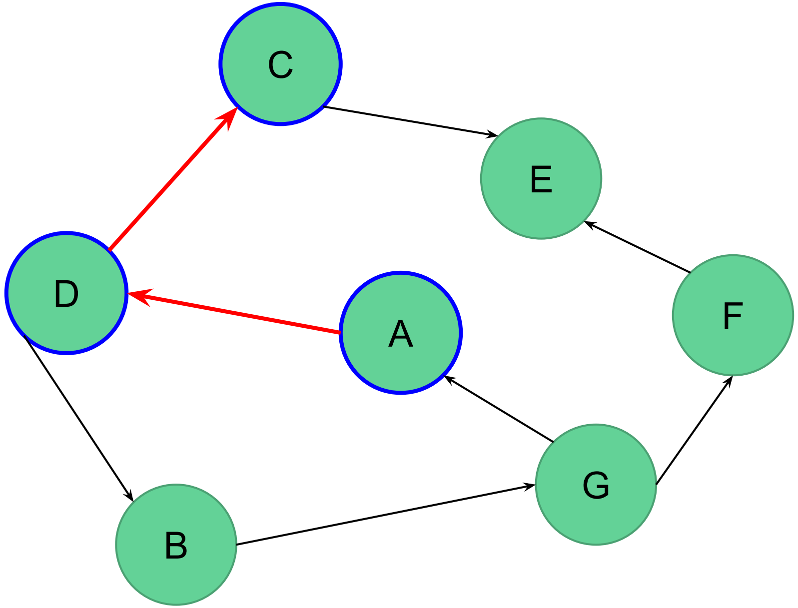
1. Pop (D, {A}) from the top of the stack.
2. Mark D
3. Process the unmarked neighbors of D:
   a. Push (B, {A,D}) to the stack.
   b. Push (C, {A,D}) to the stack.

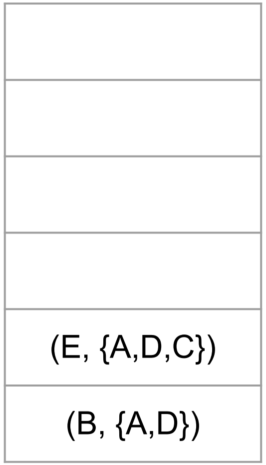




Graph Search - DFS

1. Pop (C, {A,D}) from the top of the stack.
2. Mark C
3. Process the unmarked neighbors of C:
   a. Push (E, {A,D,C}) to the stack.

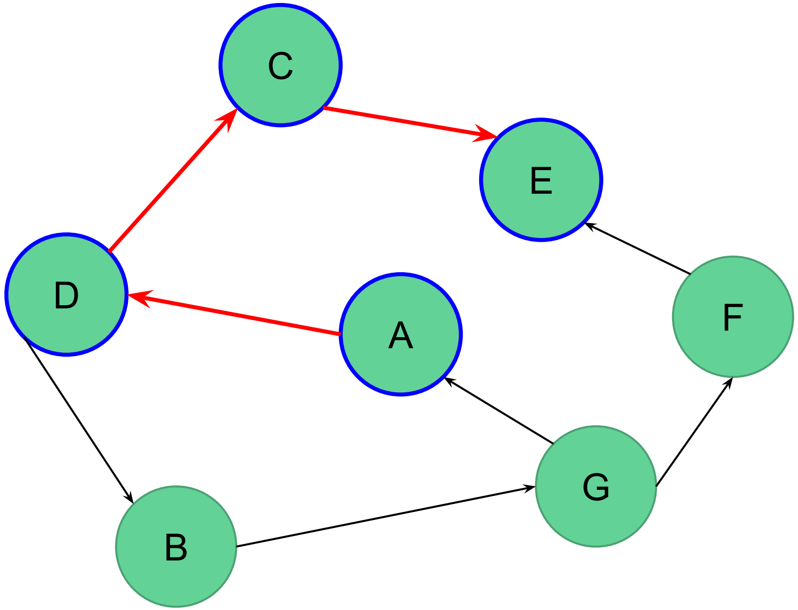


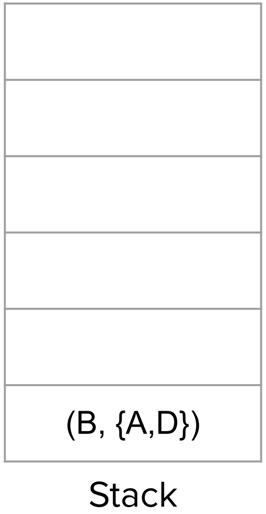


Stack

Graph Search - DFS

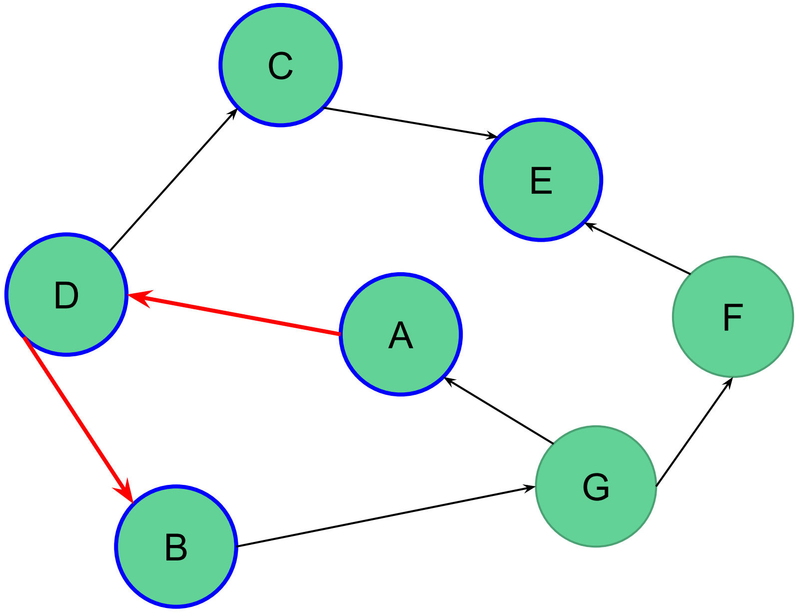
1. Pop (E, {A,D,C}) from the top of the stack.
2. Mark E
3. Process the unmarked neighbors of E:
    a. E has no neighbors

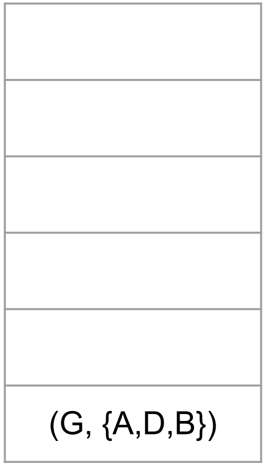




Graph Search - DFS

1. Pop (B, {A,D}) from the top of the stack.
2. Mark B
3. Process the unmarked neighbors of B:
    a. Push (G, {A,D,B}) to the stack.

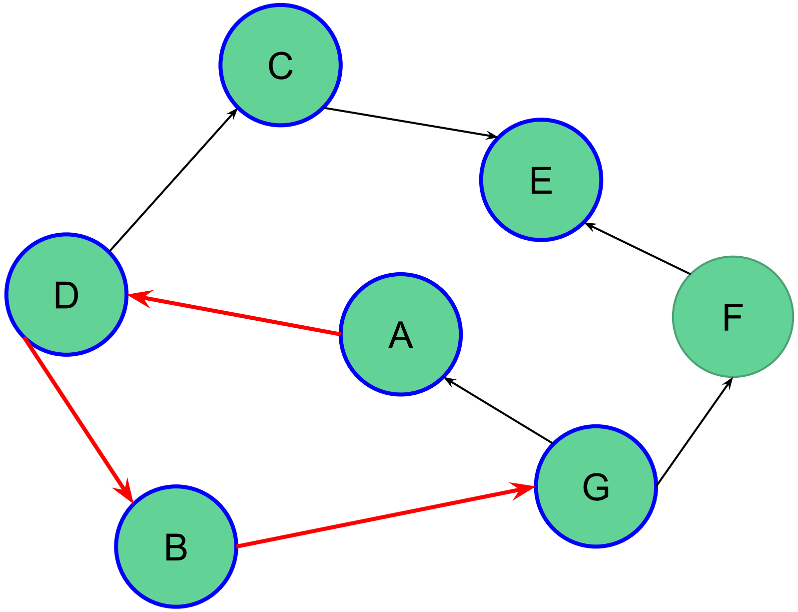


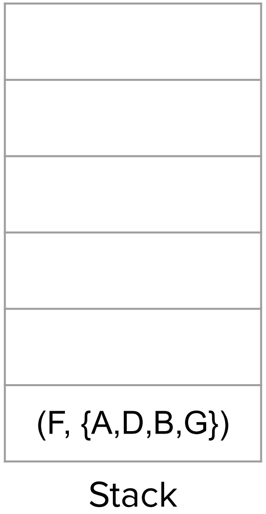


Stack

Graph Search - DFS

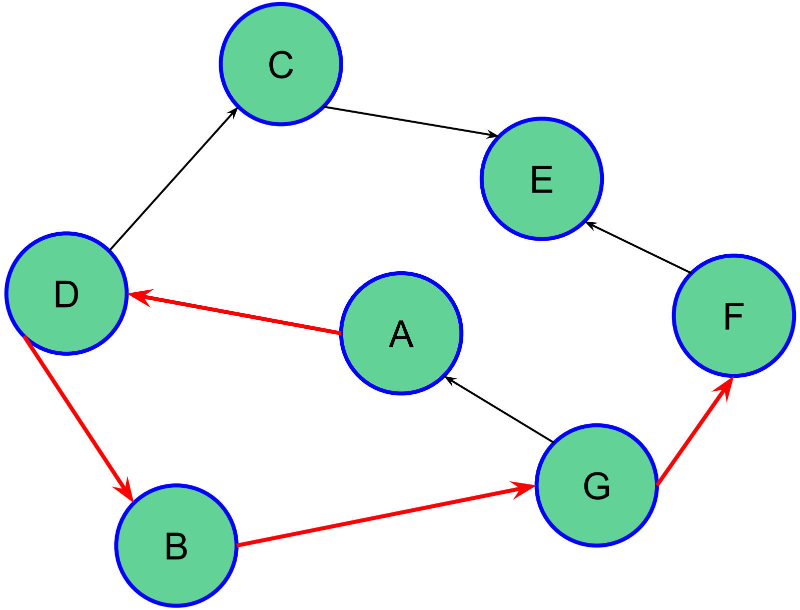
1. Pop (G, {A,D,B}) from the top of the stack.
2. Mark G
3. Process the unmarked neighbors of B:
   a. Push (F, {A,D,B,G}) to the stack.
   b. A is already marked, so do not push.

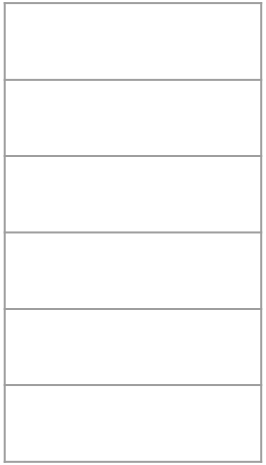




Graph Search - DFS

1. Pop (F, {A,D,B,G}) from the top of the stack.
2. Mark F
3. F is the terminus node, so terminate the algorithm here.

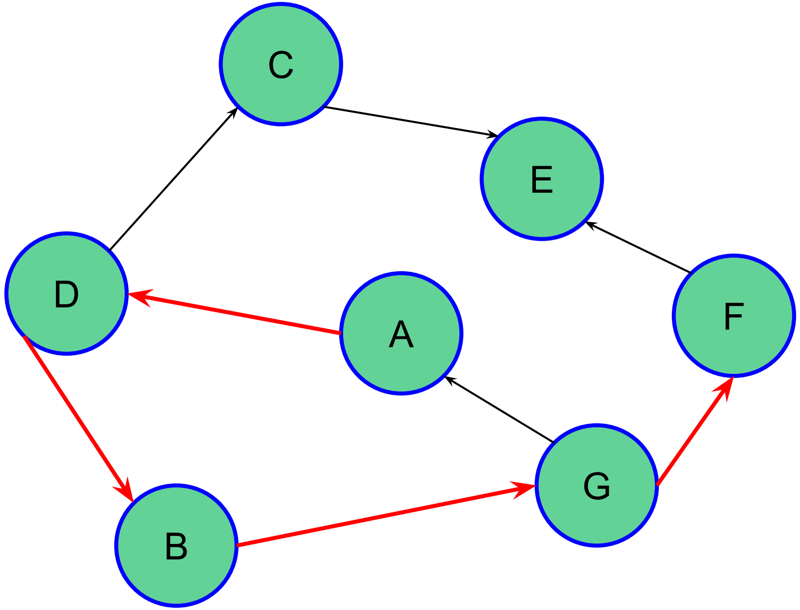


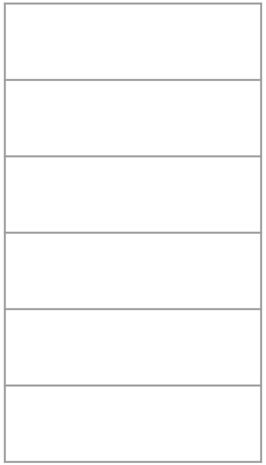


Stack

Graph Search - DFS

1. Our algorithm returns (F, {A,D,B,G}).
2. We can augment F to the path for the final result: {A,D,B,G,F} as our full path from A to F.
3. Another way this algorithm could work would be by simply storing the parents of each node in the stack.
4. An abstract data type is needed to store a node and its parent



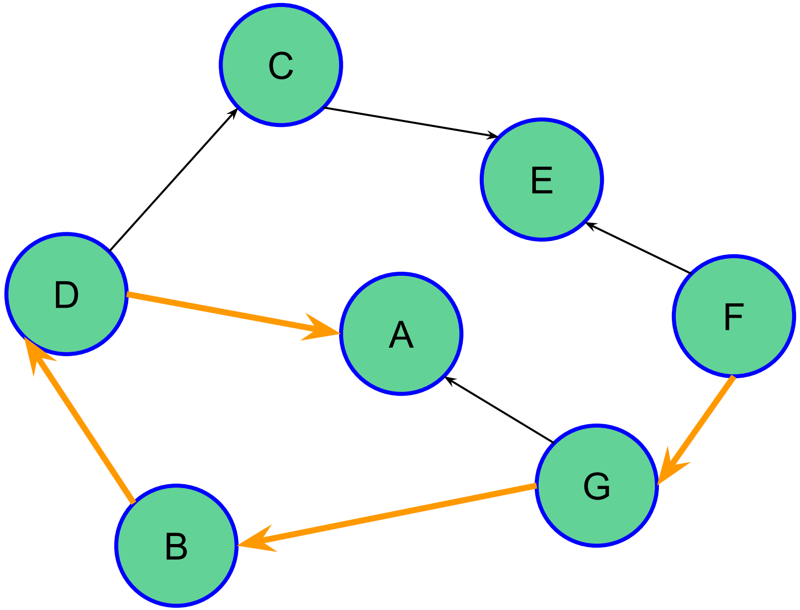


Stack

Simple Abstract Data Type (NodePath)

- This ADT will have two fields:
  1. Node n;
  2. NodePath parent;
- The NodePath will simply store the parent that got to Node n
- Suppose we ran our DFS using this method instead of (Node n, Path {})

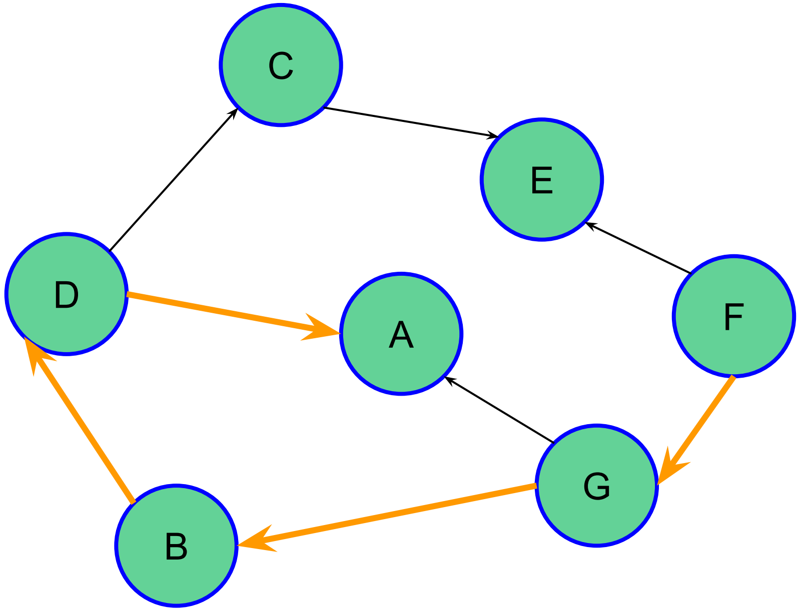
Graph Search - DFS with NodePath ADT



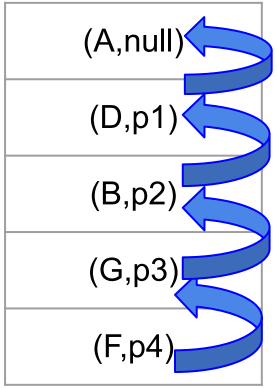
- Our algorithm would actually produce a linked-list!
- We would see something like this:
- F → G → B → D → A → Null
- Notice that the arrow direction has flipped!

- Consider why, starting from the beginning:
- The first element, p_1 = (A, null) is still the same since A has no parent
- Then p_2 = (D, p1) is the second
- Then p_3 = (C, p_2) is the third
- So, by the time we get to F, we have p_n= (F, p_{n-1}). However, we must traverse the linked list backwards now to determine the path. This technique is called "**backtracking**" and is critically important in Graph Theory.

Graph Search - DFS with NodePath ADT

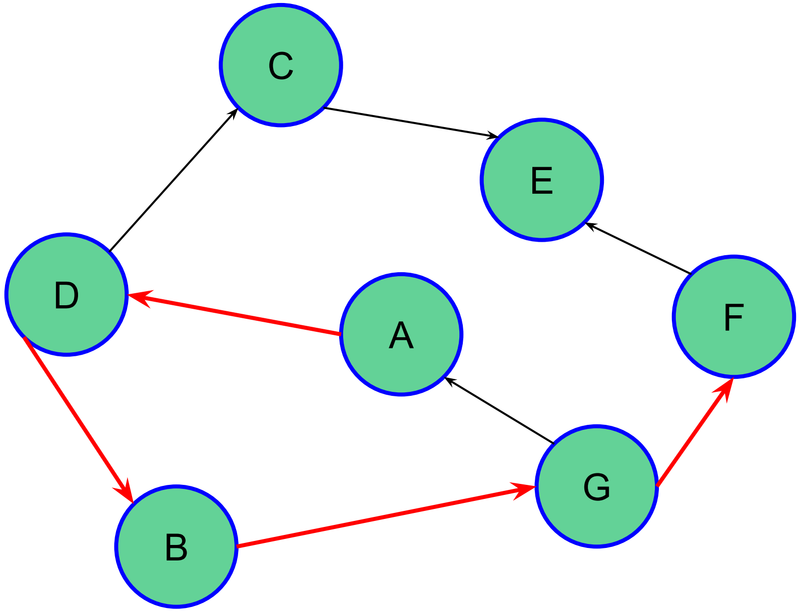


- We need to get the path in order from A→F
- First, add the entire linked-list path into a stack
- P = (F, p_{n-1})
- while(P != null):
  - stack1.push(P)
  - P = P.parent
- For simplicity, I will label all parents in numerical order (though this is not the actual order of traversal)



Stack

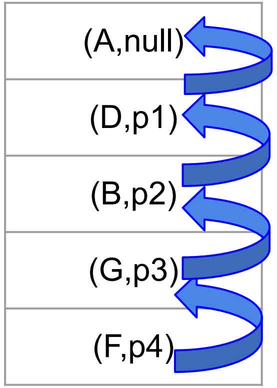
Graph Search - DFS with NodePath ADT



- Now, we can read off the stack in-order and get the actual path from A to F
- Path = list()
- while(stack is not empty):
    - Path .append(stack.pop().n)

⇒

- Path = {A,D,B,G,F}
- We have our path in-order again!



Stack

Overview - DFS

- The name of this pathfinding algorithm is Depth-First Search (DFS)
- Notice the way it looked through the first path (preferring depth) before going on to to the second path.
- DFS is good at making long gains quickly, but it may waste time if it checks paths in a inefficient ordering.
- The usage of the stack is necessary for DFS to happen.
- Using a “marking” system saves computation speed by making sure we do not recheck already processed or currently processing nodes. This can be accomplished easily with a HashSet.

Graph Search - BFS

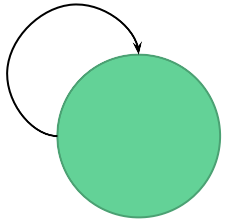
- The second algorithm for graph search we will look at is Breadth-First search (BFS).
- This search will take a source node (root) and will find paths to an end node (terminus).
- This search can easily be extended to find all possible paths in the graph starting from the start (simply by not specifying an ending node).

High-Level BFS Algorithm:

1. Enqueue root node to queue
2. Dequeue the top node from the queue
   a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node*
4. Enqueue all unmarked neighbors of the top node to the queue
5. Repeat step 2-5, until queue is empty or terminus point is found

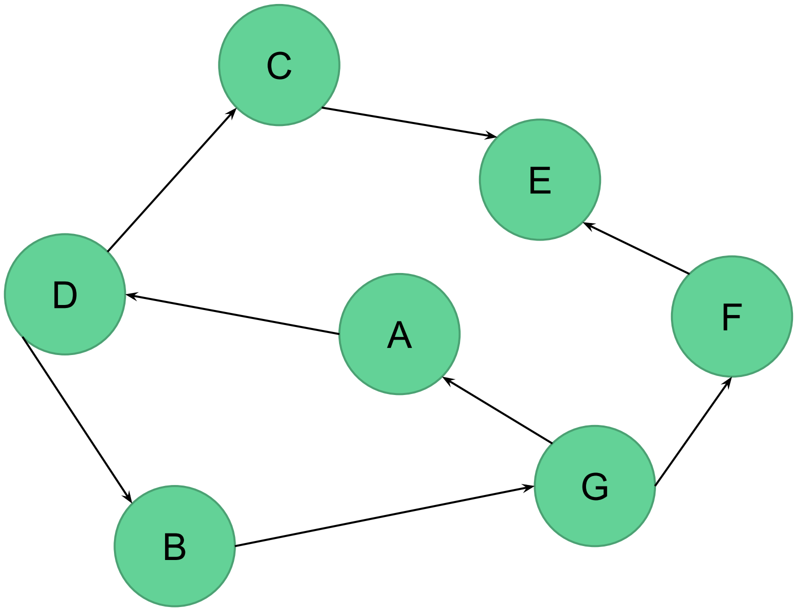
*The same note about marking still applies:

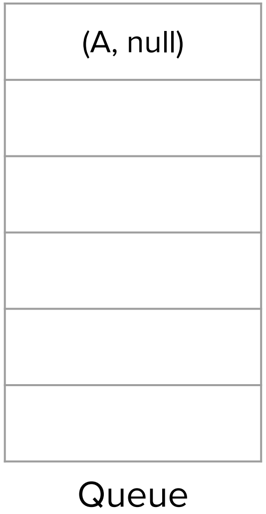
- You can represent marking with a hashset for efficiency e.g. marked items are in the set
- Marking the node at step 3 may save time in graphs with self-directed edges, where the edge connects a node to itself.



Graph Search - BFS

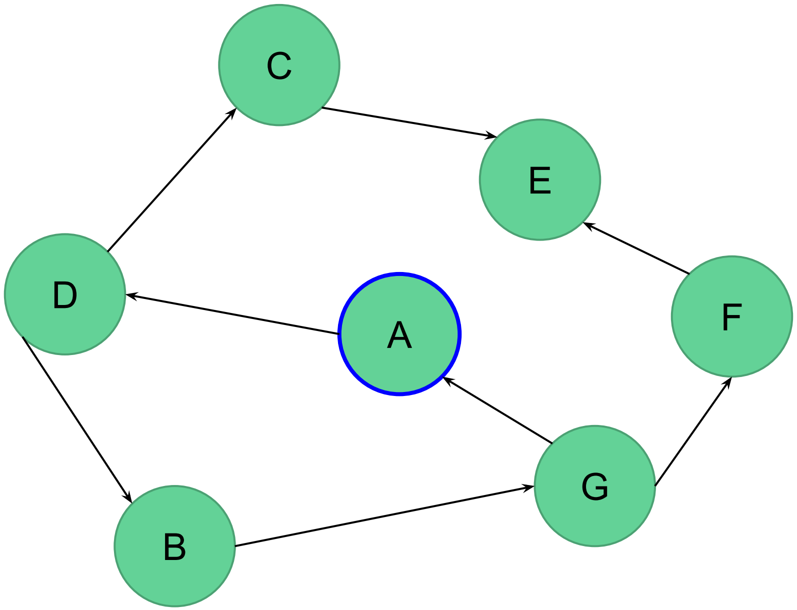
- Let's try this again, except simply swap out the Stack for a Queue data structure. Remember that queues are first-in, first-out (FIFO).
- This will process nodes in the order they appear.
- Enqueue (A, null)

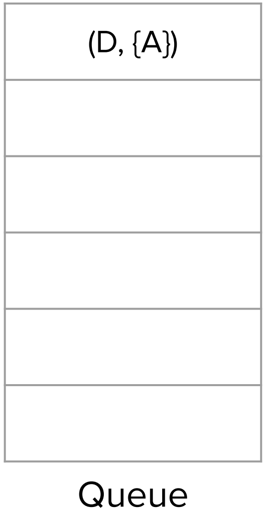




Graph Search - BFS

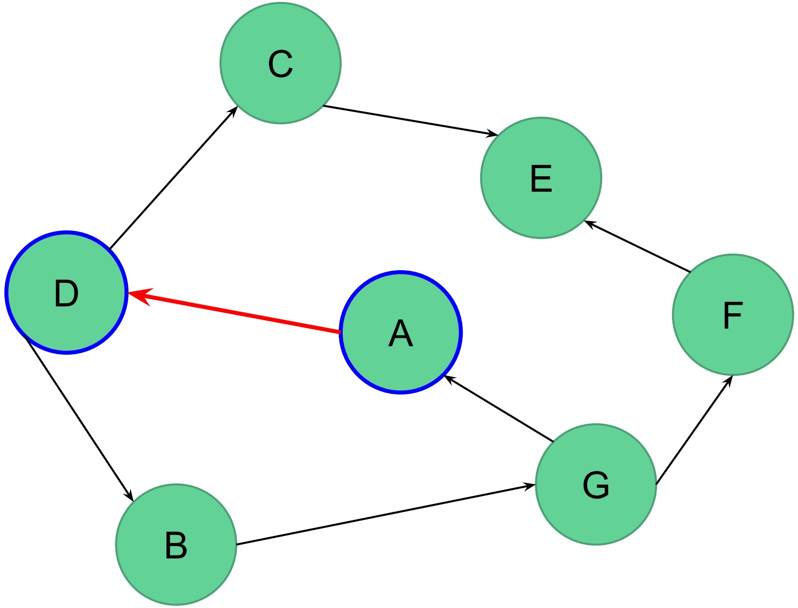
1. Dequeue (A, null) from the front of the queue.
2. Mark A
3. Process the unmarked neighbors of A:
   a. Enqueue (D, {A}).

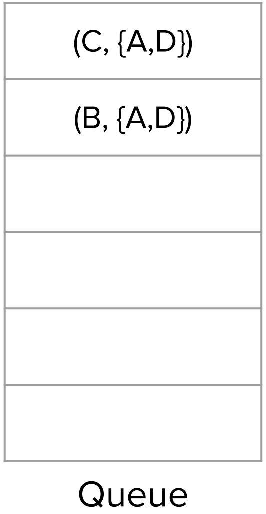




Graph Search - BFS

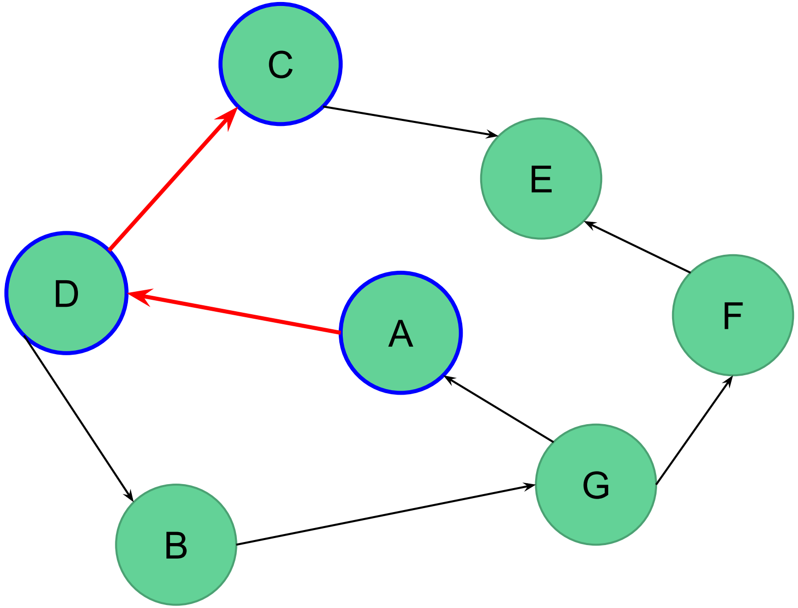
1. Dequeue (D, {A}) from the front of the queue.
2. Mark D
3. Process the unmarked neighbors of D:
   a. Enqueue (C, {A,D}).
   b. Enqueue (B, {A,D}).

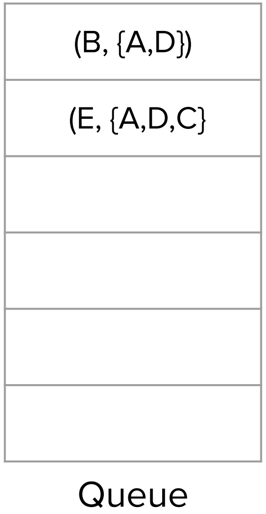




Graph Search - BFS

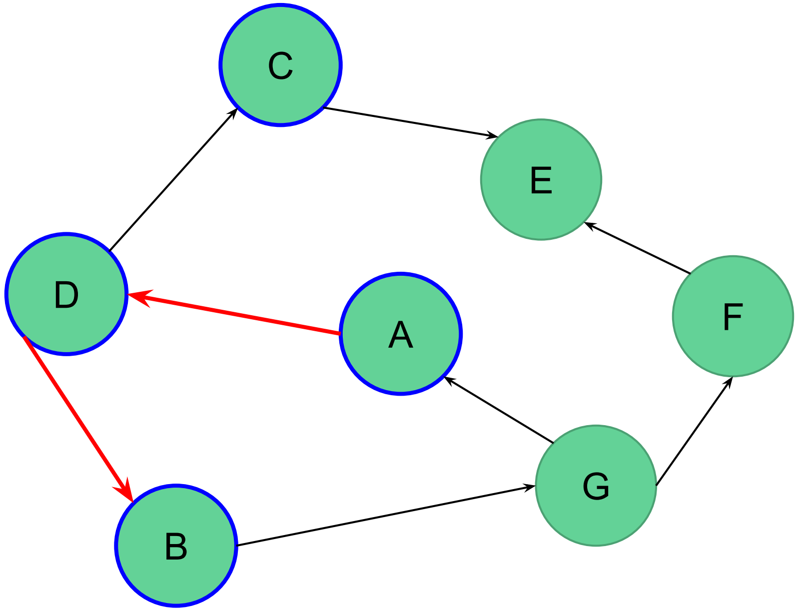
1. Dequeue (C, {A,D}) from the front of the queue.
2. Mark C
3. Process the unmarked neighbors of C:
   a. Enqueue (E, {A,D,C}).

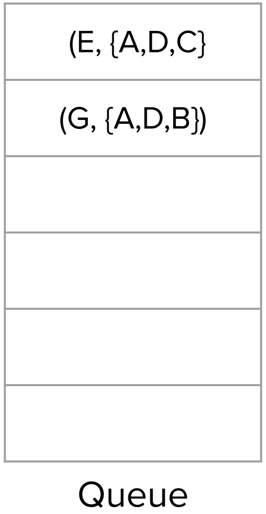




Graph Search - BFS

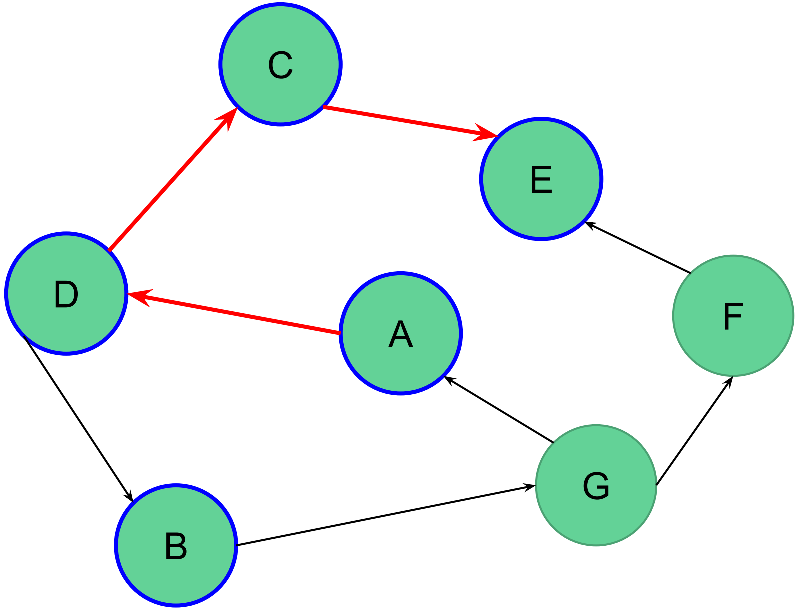
1. Dequeue (B, {A,D}) from the front of the queue.
2. Mark B
3. Process the unmarked neighbors of B:
    a. Enqueue (G, {A,D,B}).

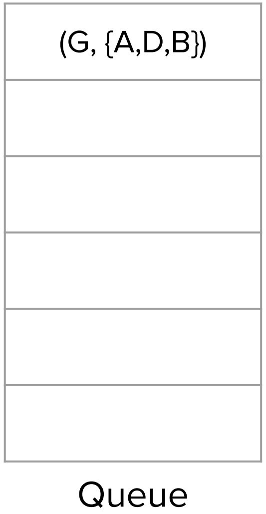




Graph Search - BFS

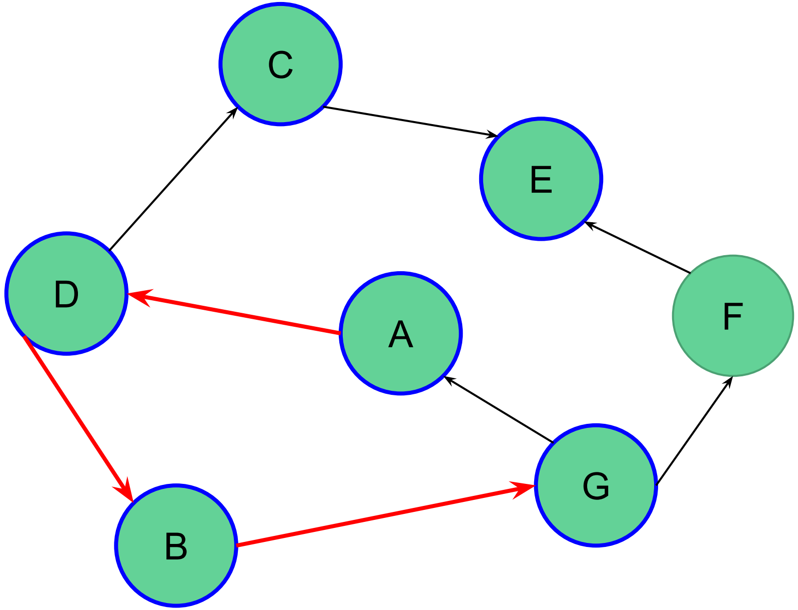
1. Dequeue (E, {A,D,C} from the front of the queue.
2. Mark E
3. Process the unmarked neighbors of E:
    a. E has no neighbors

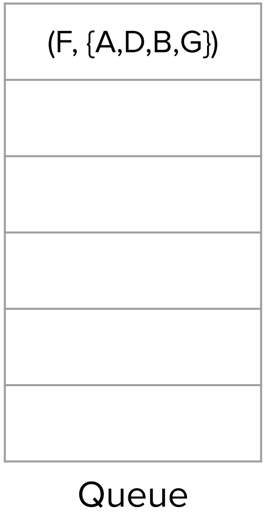




Graph Search - BFS

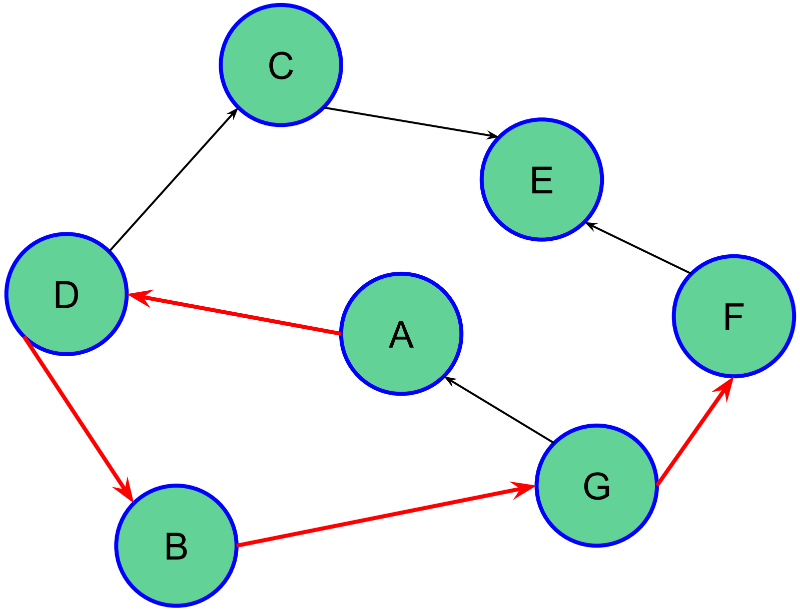
1. Dequeue (G, {A,D,B}) from the front of the queue.
2. Mark G
3. Process the unmarked neighbors of G:
   a. Enqueue (F, {A,D,B,G}).
   b. A is already marked.

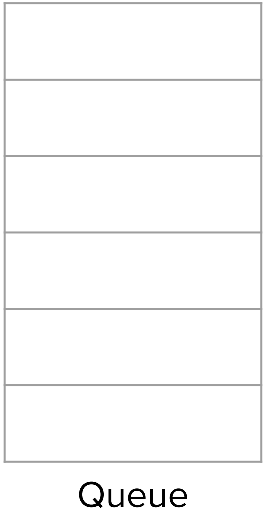




Graph Search - BFS

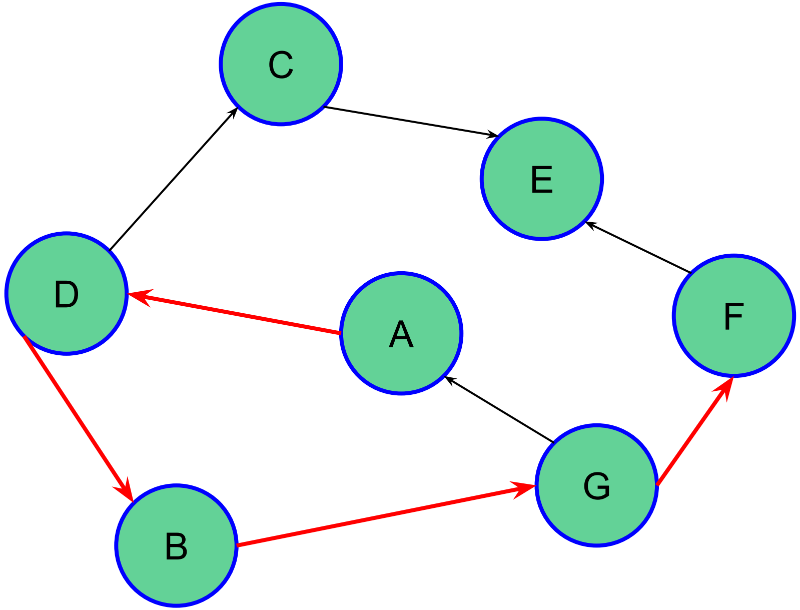
1. Dequeue (F, {A,D,B,G}) from the front of the queue.
2. Mark F
3. F is the terminus node, so terminate the algorithm here.

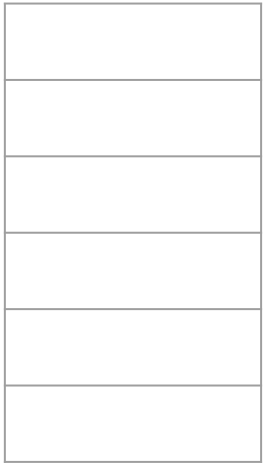




Graph Search - BFS

- Our algorithm returns (F, {A,D,B,G}).
- We can augment F to the path for the final result: {A,D,B,G,F} as our full path from A to F.
- Like DFS, we could also use the NodePath ADT approach for BFS and use backtracking to get the ordered solution path.





Queue

Overview - BFS

- BFS may be inefficient when there are many neighbors that require checking before progress further away may be made.
- BFS tends to be the basis for more advanced pathfinding algorithms, but it will use a weight-based approach for selecting neighbors connected by edges with the least-weight.
    - This would use a priority-queue data structure, which returns neighbors in order of highest priority.

BFS vs DFS

- In terms of implementation, the only major difference between BFS and DFS is Queue and Stack usage respectively.
- They are functionally equivalent, but one may outperform the other depending on the graph construction.
- Both may use an augmenting path (list), or NodePath (linked-list) approach to constructing the final-path once the algorithm terminates.
- Either may be used for finding all possible paths in a network from a single-source/root, simply by not specifying a terminal node.
    - The respective algorithm will terminate once the stack or queue is empty.

Pathfinding and Mazes

- DFS tries to make gains to the end as fast as possible
- Humans tend to scan mazes by trying to follow single paths – just like DFS!
- Think about it:
  - You will scan a path until you hit a wall, and then trace back to the next possible path.
  - This is much easier than thinking about solving a maze with BFS, as it is too much information to consider several paths branching out at once!
- Problems that involve lengthy paths are much better with DFS.
  - BFS will take a long time, having to explore all paths node-by-node
  - Try not to use a BFS with a maze! You are better off using informed search algorithms you will learn later.
- However, BFS will find shorter path lengths than DFS.

Conclusions

- BFS and DFS are not too difficult to implement, and a simple change in data structure from Queue to Stack, or vice-versa, will flip your search technique.
- Although they are functionally equivalent, it is sometimes more appropriate to use one over the other
- BFS and DFS are “uninformed” searches. This is because they explore all neighbor nodes without any notion of which is better.