Discussion 10

DFS, BFS, SPs, MSTs

Announcements

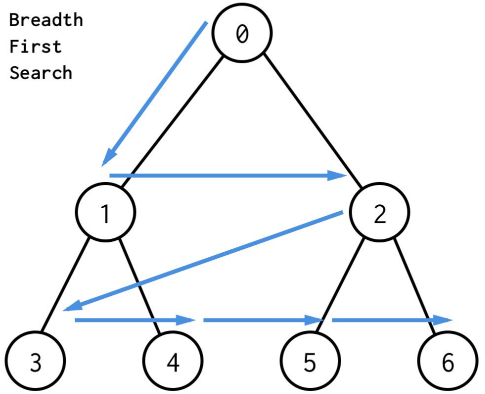
- Welcome back from spring break! Hope you had a restful week :)
- Reach out to your mentor GSI if you feel yourself falling behind, or are feeling overwhelmed -- we're here for you! We want to help you finish as strong as you can.
- We will be running the full autograder for project 2ab on ~4/2.
    - No extensions will be approved beyond this date except in case of **dire emergency**, and such extensions will need to go through Professor Hug

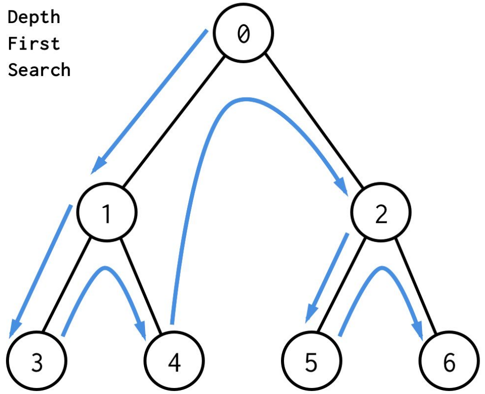
- **Midterm 2 this Friday, April 5th from 8-10pm**!
- Check out **@3764** for important information about the midterm & review sessions!
- **No lab due** this week! We'll be doing exam review in lab.

Table of Contents

Graph Traversals

- Level-Order/**Breadth** First Search (BFS):
  - Visit top to bottom, left to right, just like how you read!
- Depth-order/**Depth** First Search (DFS):
  - Traverse “deeper” nodes before shallow ones





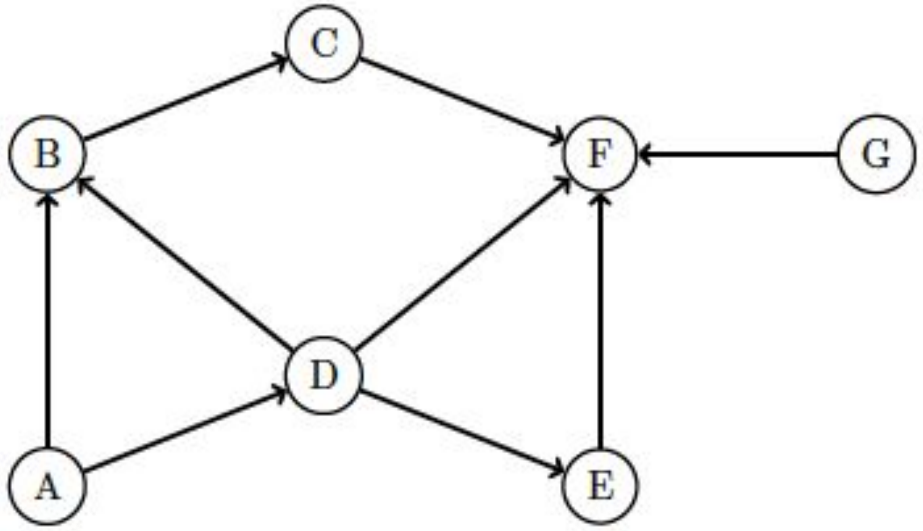
Preorder vs. Postorder vs. Inorder (all types of DFS)

- **Pre**order: “Visit” a node, then traverse its children
- **In**order: Traverse left child, “visit” node, then traverse right child
- **Post**order: Traverse children, then “visit” node.

- Note: The prefix to order (pre, in, post) refers to when a node is visited with respect to its children
  - ex) That’s why postorder means that we visit the node AFTER we visit the children (post is a prefix, meaning “behind,” “after,”)
- Note: Node and vertex are used interchangeably.

Q1.1: Graphs

Give the DFS preorder, DFS postorder, and BFS order of the graph traversals starting from vertex A. Break ties alphabetically.

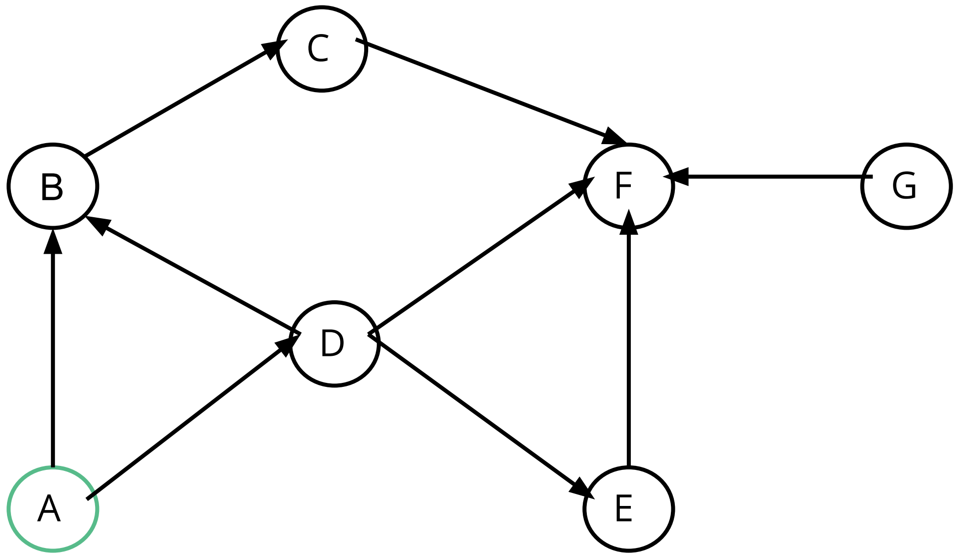


Q1.1: Implementing Post/Preorder

- Maintain a stack of nodes and a marked set.
- As soon as we add something to our stack, we note it for **preorder**.
    - The top node in our stack represents the node we are currently on, and the marked set represents nodes that have been visited.
- After we add a node to the stack, we visit its lexicographically next unmarked child.
- If there is none, we pop the topmost node from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution

remember: the node on the top of the stack is the node we're currently on



Stack: A

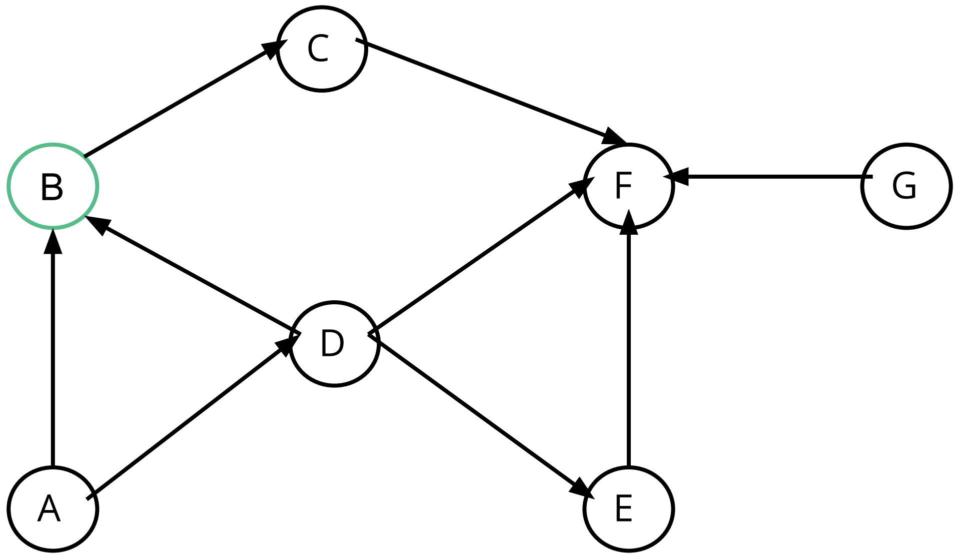
Marked Set: A

Preorder: A

(unmarked means the child is not in the Marked Set)

A has two unmarked children, but we're going in alphabetical order , so we're going add B to our stack, preorder, and marked set.

Q1.1: DFS Pre/postorder Solution



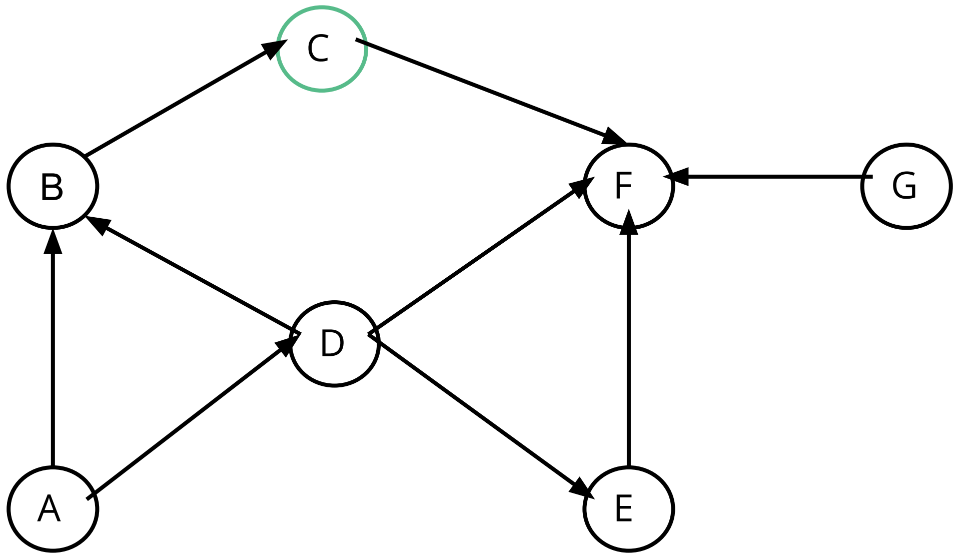
Stack: A B

Marked Set: A B

Preorder: A B

B has an unmarked child (C) , so we're going add it to our stack, preorder, and marked set.

Q1.1: DFS Pre/postorder Solution



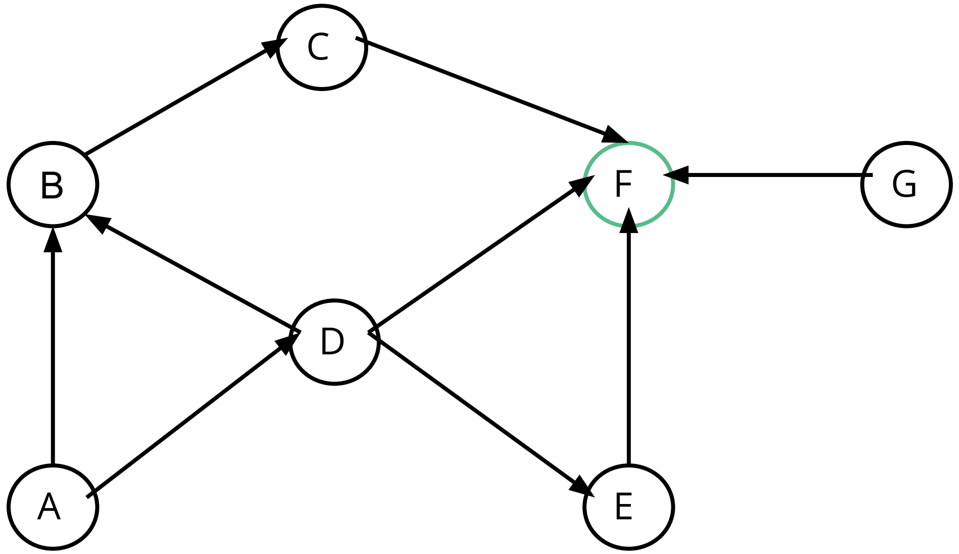
Stack: A B C

Marked Set: A B C

Preorder: A B C

C has an unmarked child (F) , so we're going add it to our stack, preorder, and marked set.

Q1.1: DFS Pre/postorder Solution

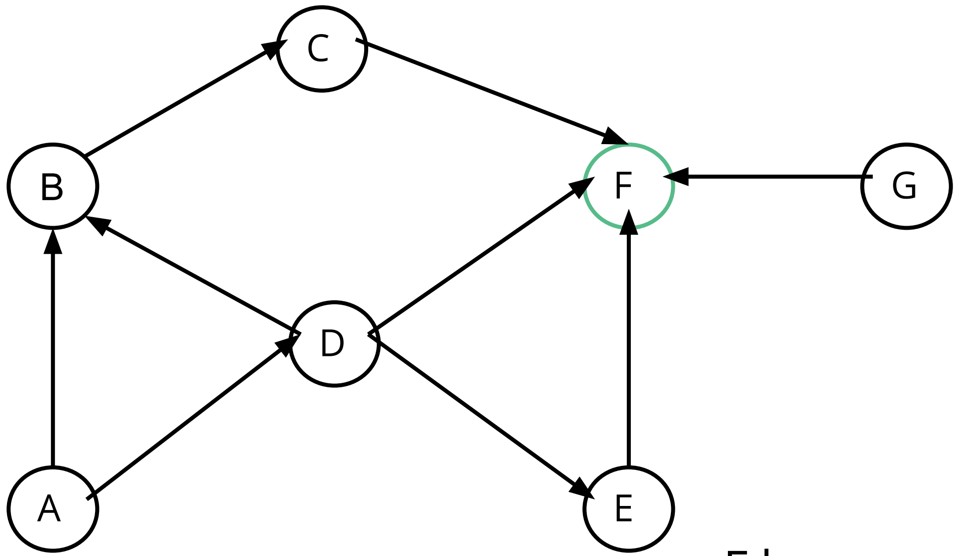


Stack: A B C F

Marked Set: A B C F

Preorder: A B C F

Q1.1: DFS Pre/postorder Solution



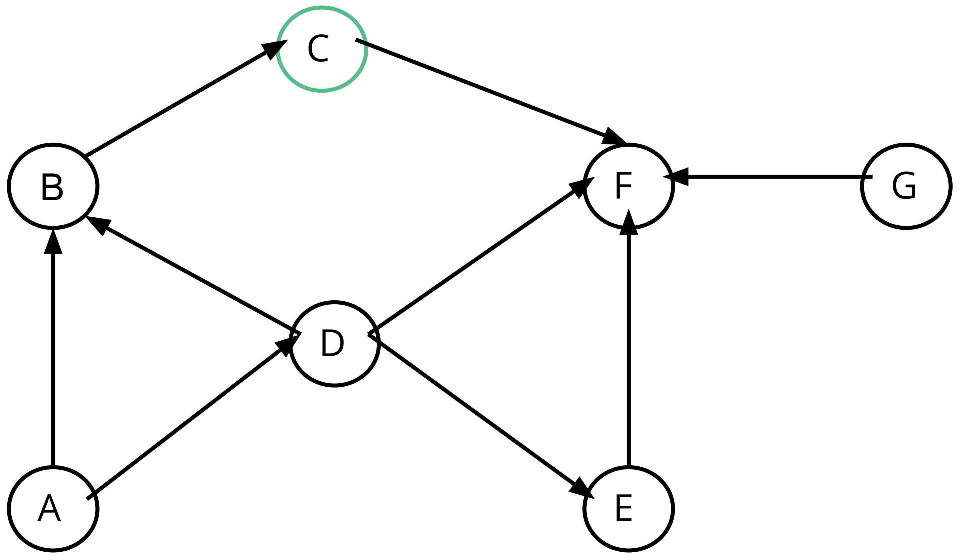
Stack: A B C F

Marked Set: A B C F

Preorder: A B C F

F has no unmarked children, so we're going to pop it from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution



Stack: A B C

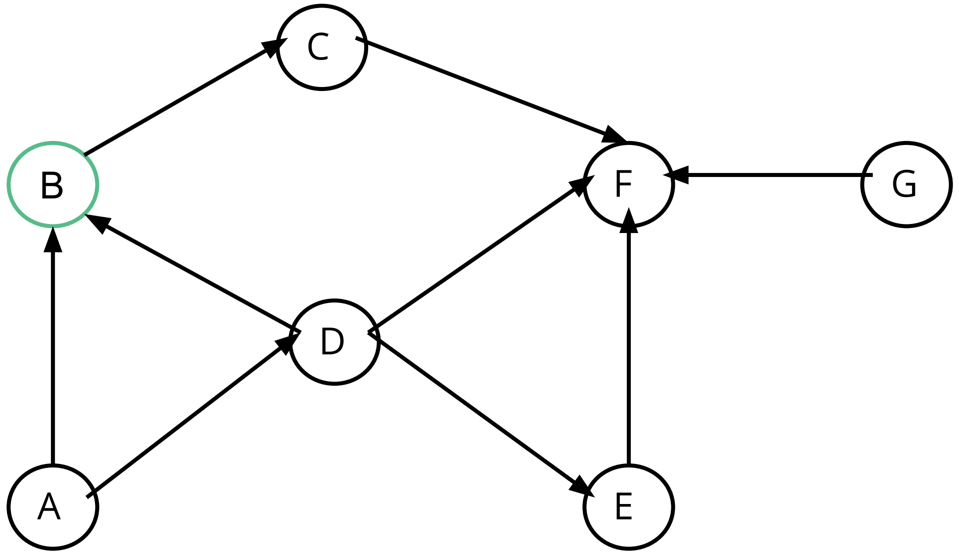
Marked Set: A B C F

Preorder: A B C F

Postorder: F

C has no unmarked children, so we're going to pop it from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution



Stack: A B

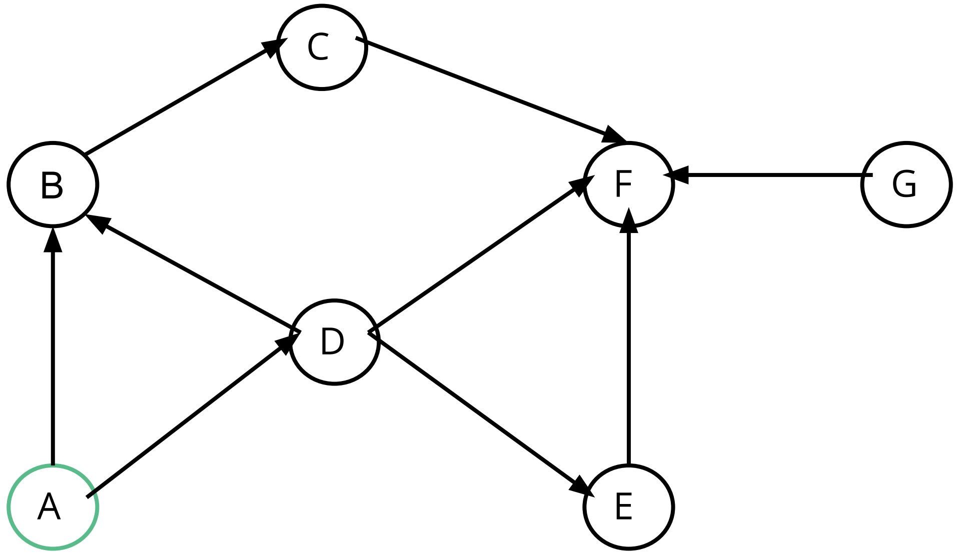
Marked Set: A B C F

Preorder: A B C F

Postorder: F C

B has no unmarked children, so we're going to pop it from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution



Stack: A

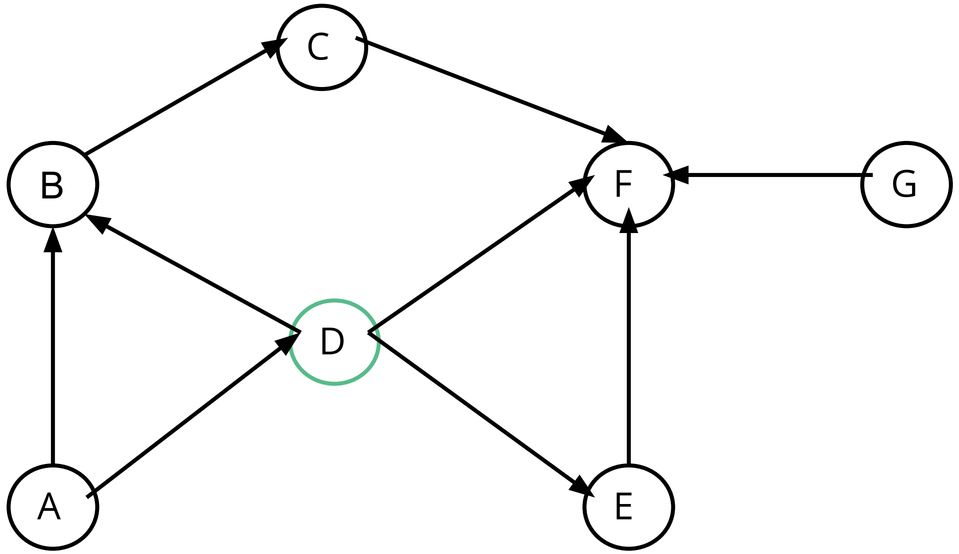
Marked Set: A B C F

Preorder: A B C F

Postorder: F C B

A has an unmarked child (D) , so we’re going add it to our stack, preorder, and marked set.

Q1.1: DFS Pre/postorder Solution



Stack: A D

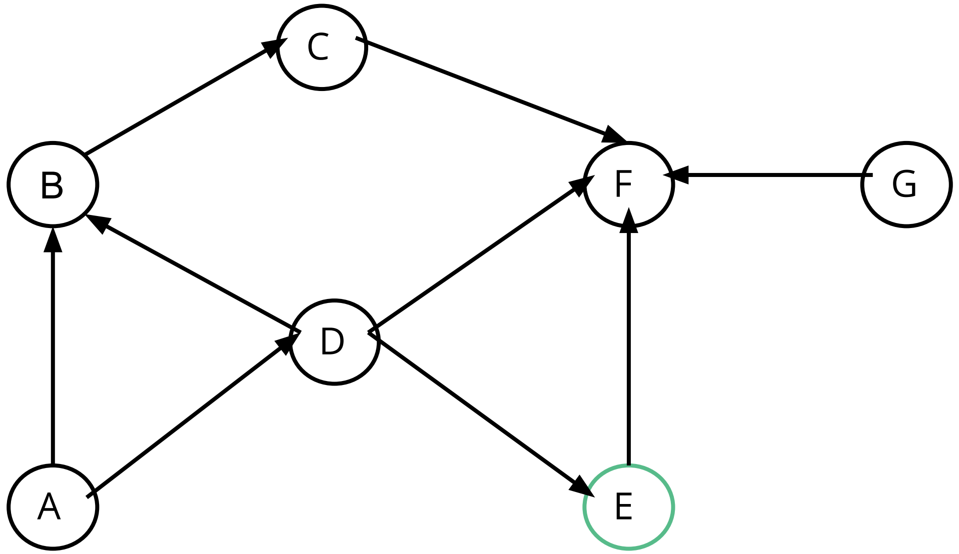
Marked Set: A B C F D

Preorder: A B C F D

Postorder: F C B

D has an unmarked child (E) , so we're going add it to our stack, preorder, and marked set.

Q1.1: DFS Pre/postorder Solution



Stack: A D E

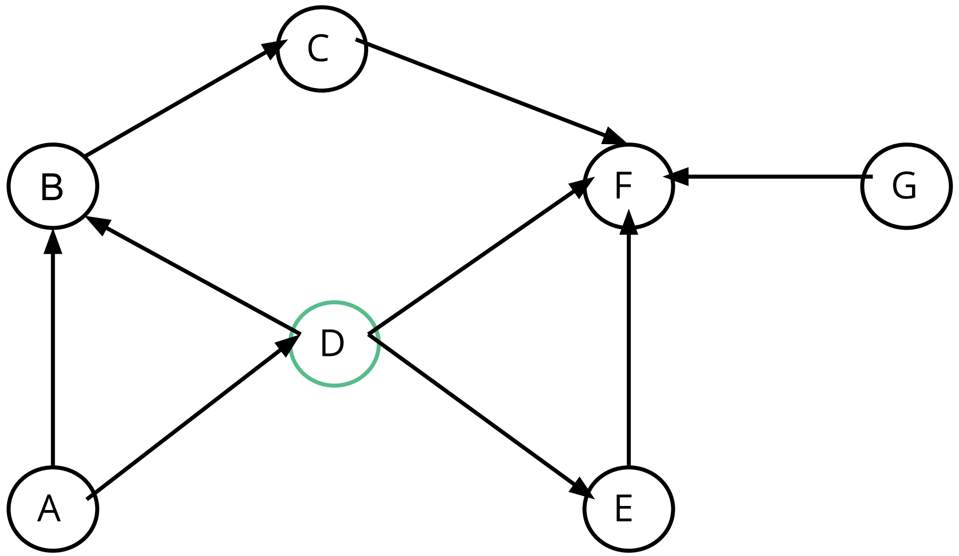
Marked Set: A B C F D E

Preorder: A B C F D E

Postorder: F C B

E has no unmarked children, so we're going to pop it from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution



Stack: A D

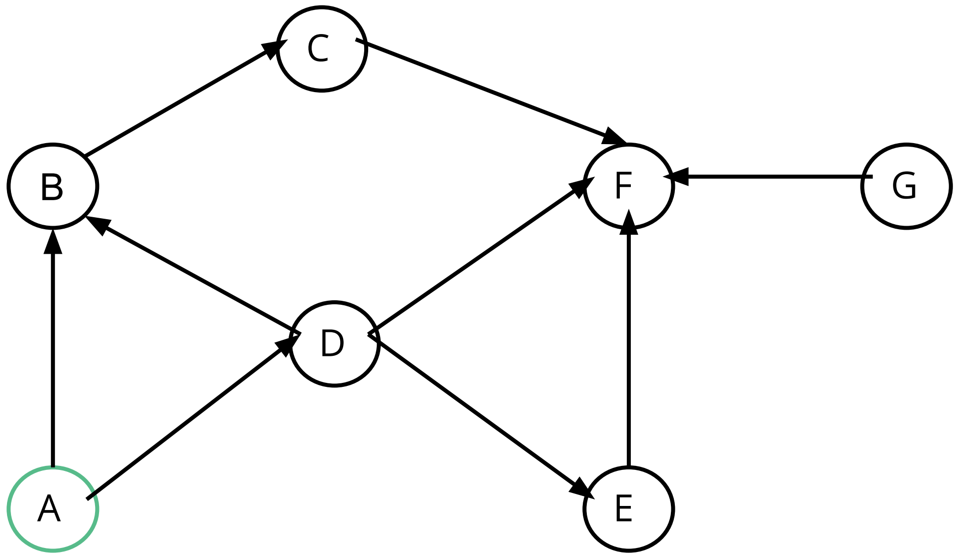
Marked Set: A B C F D E

Preorder: A B C F D E

Postorder: F C B E

D has no unmarked children, so we're going to pop it from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution



Stack: A

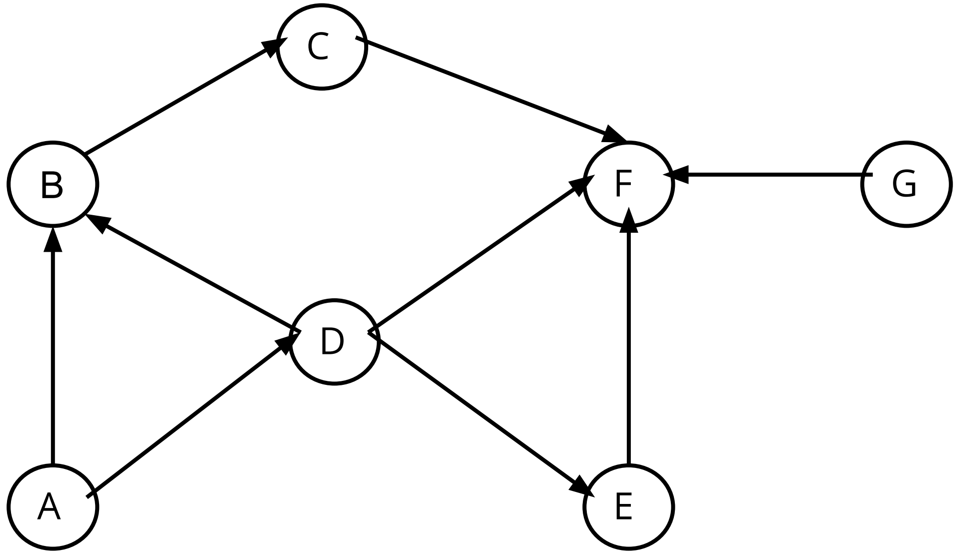
Marked Set: A B C F D E

Preorder: A B C F D E

Postorder: F C B E D

A has no unmarked children, so we're going to pop it from the stack and note it down for **postorder**.

Q1.1: DFS Pre/postorder Solution



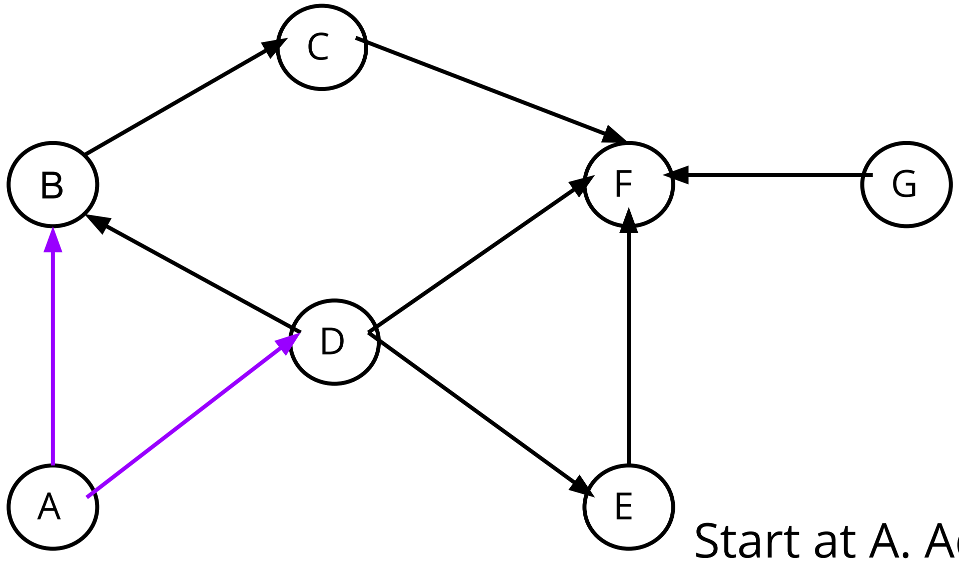
Stack: empty

Marked Set: A B C F D E (G)

Preorder: A B C F D E (G)

Postorder: F C B E D A (G)

Q1.1: BFS Solution



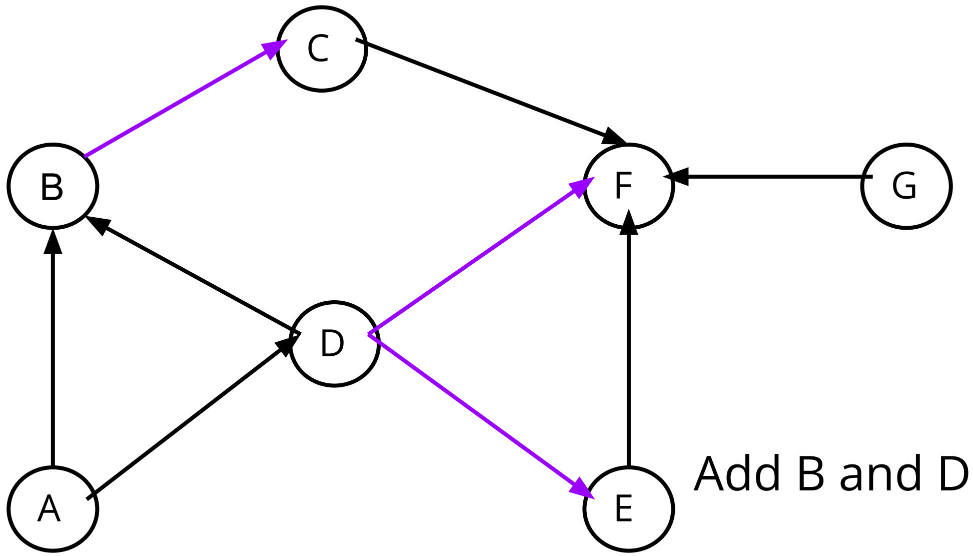
BFS: A

Marked Set: A

Start at A. Add to BFS order & mark it.

Consider all unmarked nodes 1 edge away from A.

Q1.1: BFS Solution



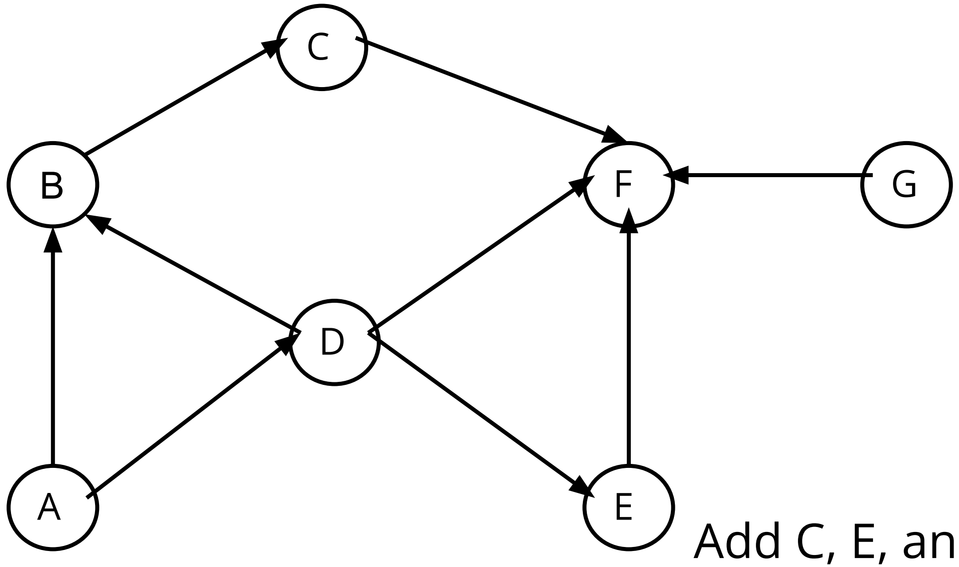
BFS: A B D

Marked Set: A B D

Add B and D to BFS order & mark them.

Consider all unmarked nodes 1 edge away from B and D. Note that B is 1 edge away but is marked.

Q1.1: BFS Solution



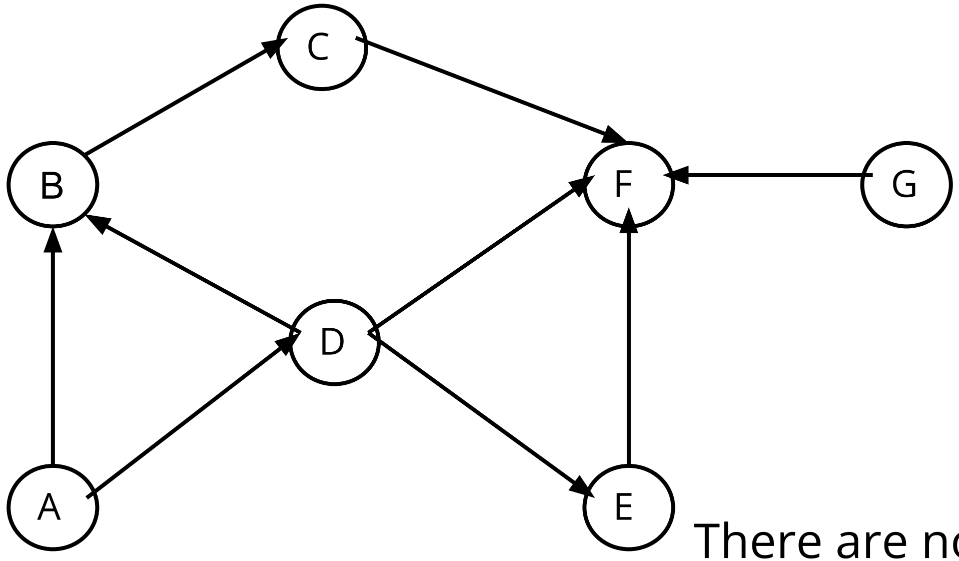
BFS: A BD CEF

Marked Set: A B D C E F

Add C, E, and F to BFS order & mark them.

Consider all unmarked nodes 1 edge away from C, E, and F.

Q1.1: BFS Solution



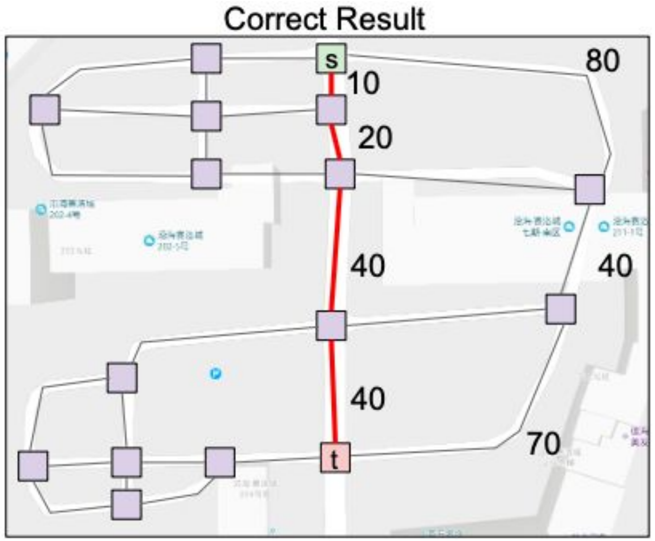
BFS: A BD CEF (G)

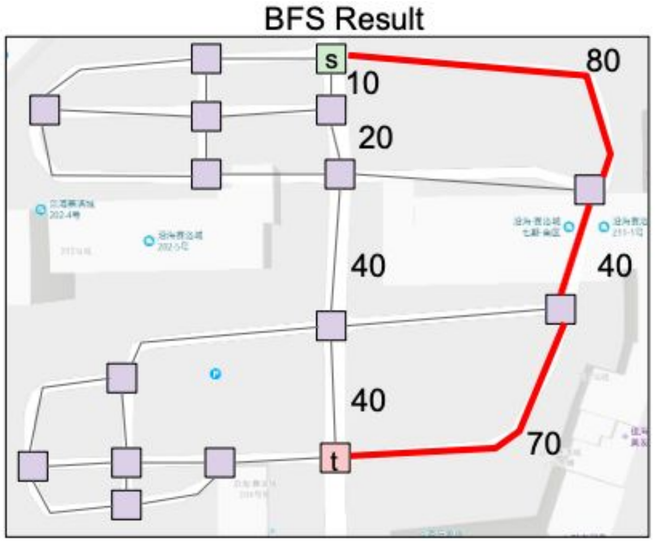
Marked Set: A B D C E F (G)

There are none, so we're done.

Dijkstra’s Algorithm

- Suppose we want the shortest path from vertex A to any vertex B.
    - BFS doesn’t always give us the shortest path (see below)
    - We need something that accounts for the actual distances or edge distances/weights between vertices.





Dijkstra's Algorithm

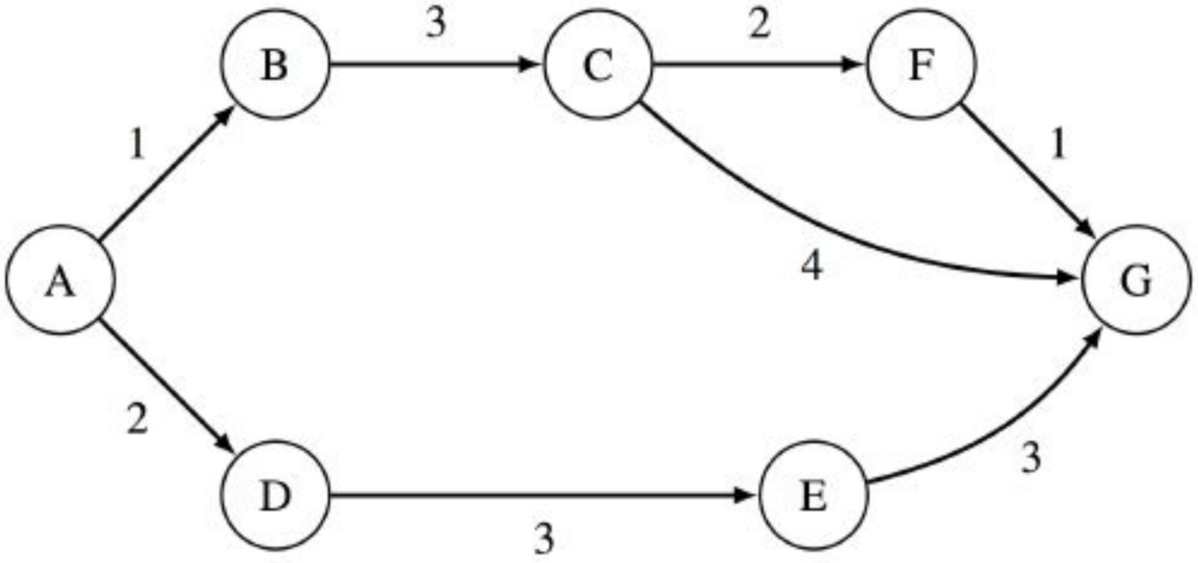
- Main Idea:
    - Insert all vertices into fringe PQ, storing vertices in order of distance from source.
    - Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.
    - Single source, multiple targets

- It's guaranteed to produce the correct result if all edges are non-negative.

- Dijkstra's Algorithm Demo Link (from lecture 25)

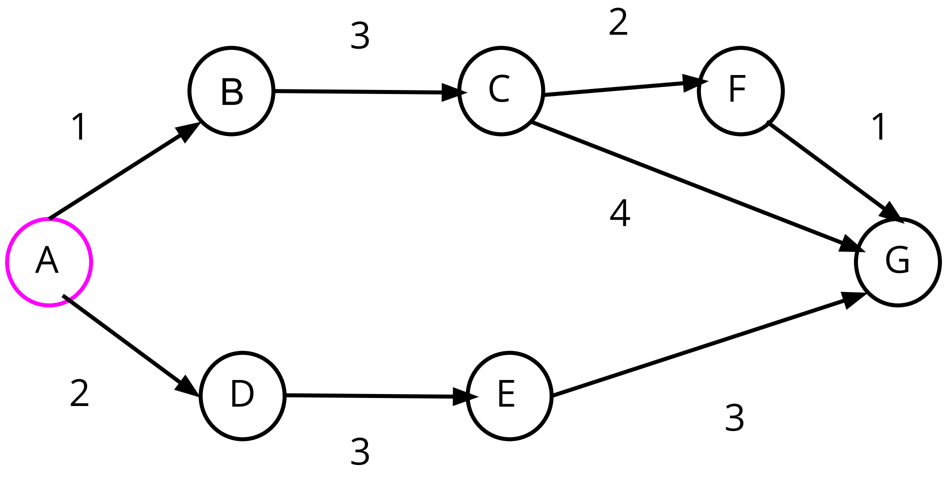
Q2.1: Dijkstra’s Algorithm

For the graph below, let `g(u, v)` be the weight of the edge between any nodes `u` and `v`. Let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`.



Edge weights	Heuristics
$g(A,B) = 1$	$h(A,G) = 8$
$g(B,C) = 3$	$h(B,G) = 6$
$g(C,F) = 4$	$h(C,G) = 5$
$g(C,G) = 4$	$h(F,G) = 1$
$g(F,G) = 1$	$h(D,G) = 6$
$g(A,D) = 2$	$h(E,G) = 3$
$g(D,E) = 3$	
$g(E,G) = 3$	

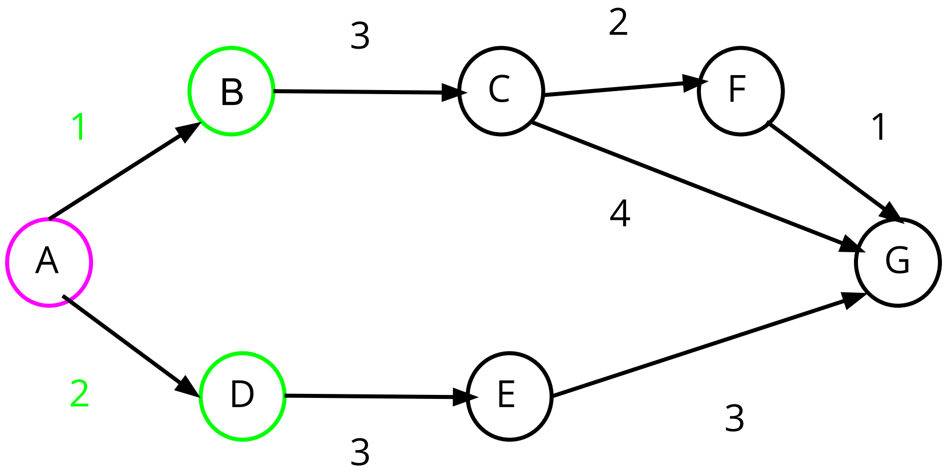
Q2.1: Dijkstra's Algorithm Solution



Pop A.

Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
B: inf		
C: inf		
D: inf		
E: inf		
F: inf		
G: inf		

Q2.1: Dijkstra's Algorithm Solution

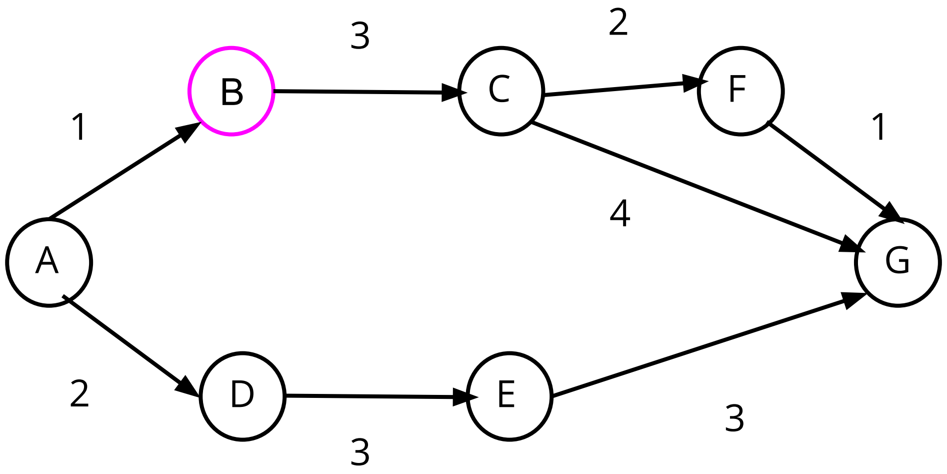


changePriority(B, 1),
changePriority(D, 2)

Priority Queue	**Distances**	**edgeTo**
~~A: 0~~	A: 0	—
B: 1	B: 1	A
D: 2	D: 2	A
C: inf		
E: inf		
F: inf		
G: inf		

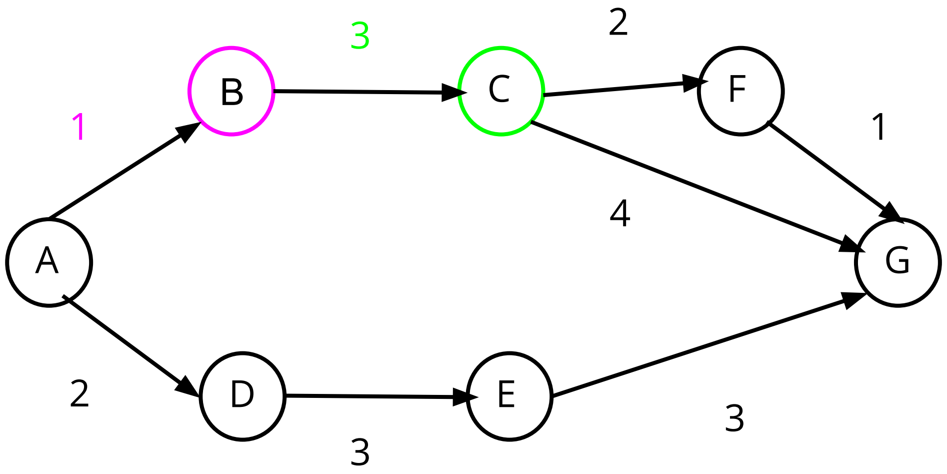
Q2.1: Dijkstra's Algorithm Solution

Pop B.



Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
D: 2	D: 2	A
C: inf		
E: inf		
F: inf		
G: inf		

Q2.1: Dijkstra's Algorithm Solution

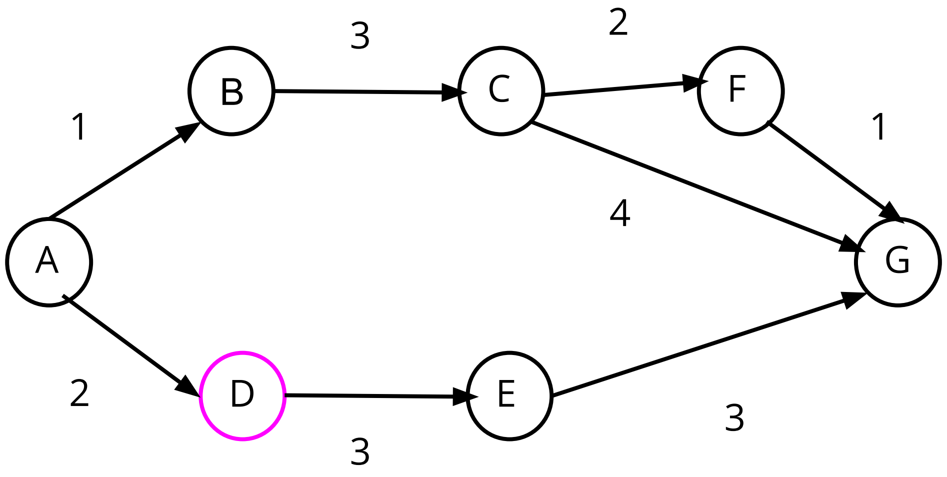


changePriority(C, 4)

Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
D: 2	D: 2	A
C: 4	C: 4	B
E: inf		
F: inf		
G: inf		

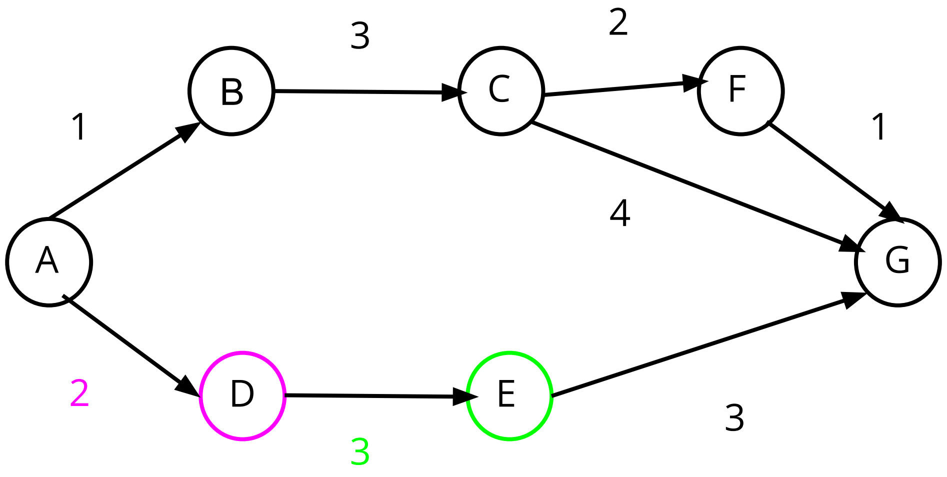
Q2.1: Dijkstra's Algorithm Solution

Pop D.



Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
C: 4	C: 4	B
E: inf		
F: inf		
G: inf		

Q2.1: Dijkstra's Algorithm Solution

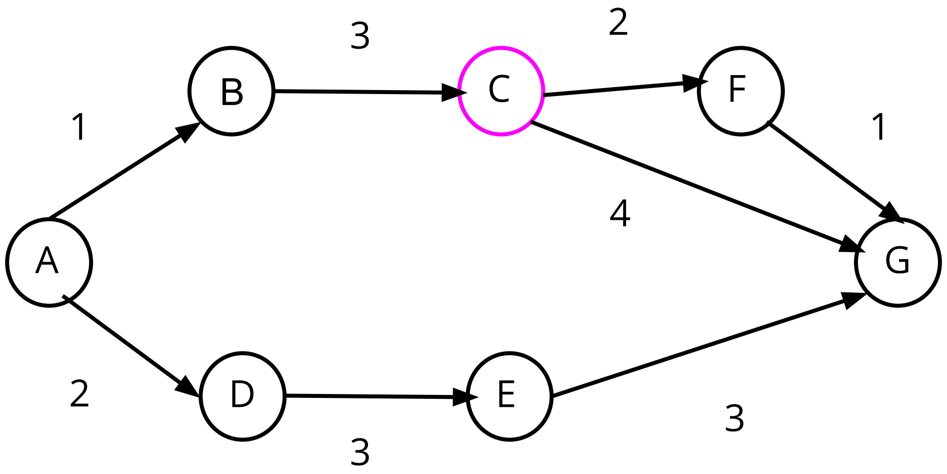


changePriority(E, 5)

Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
C: 4	C: 4	B
E: 5	E: 5	D
F: inf		
G: inf		

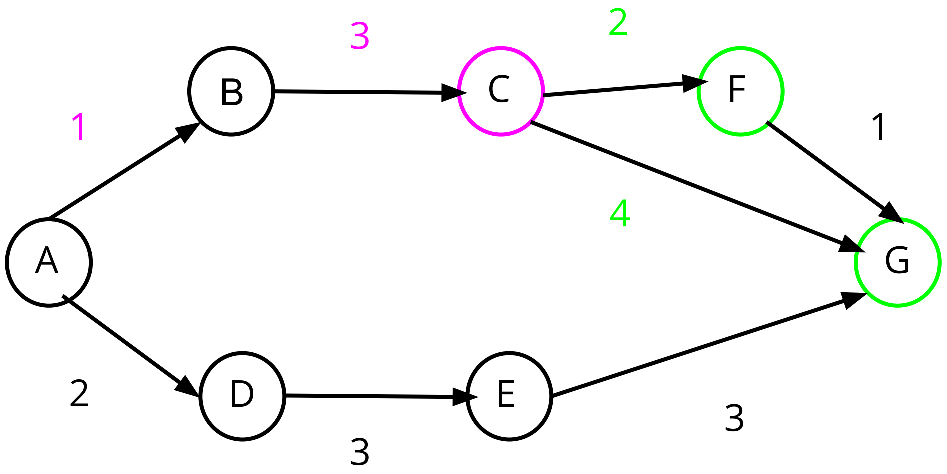
Q2.1: Dijkstra's Algorithm Solution

Pop C.



Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
E: 5	E: 5	D
F: inf		
G: inf		

Q2.1: Dijkstra’s Algorithm Solution

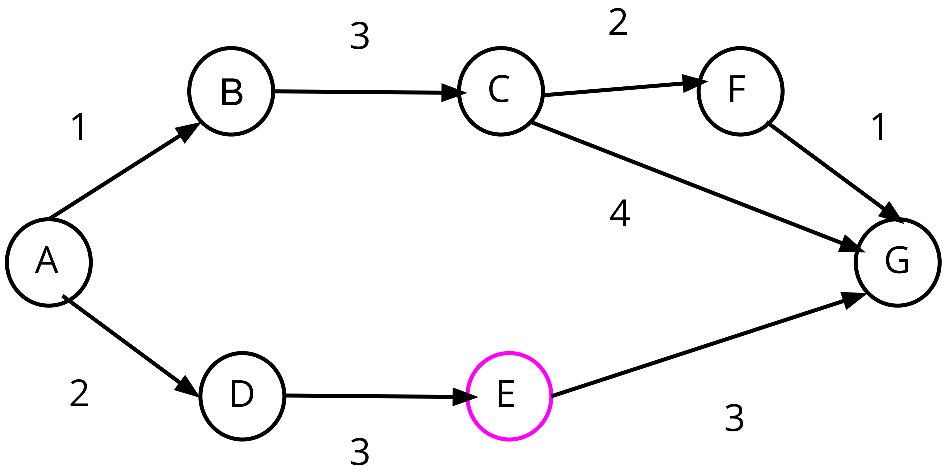


changePriority(F, 6),
changePriority(G, 8)

Priority Queue	**Distances**	**edgeTo**
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
E: 5	E: 5	D
F: 6	F: 6	C
G: 8	G: 8	C

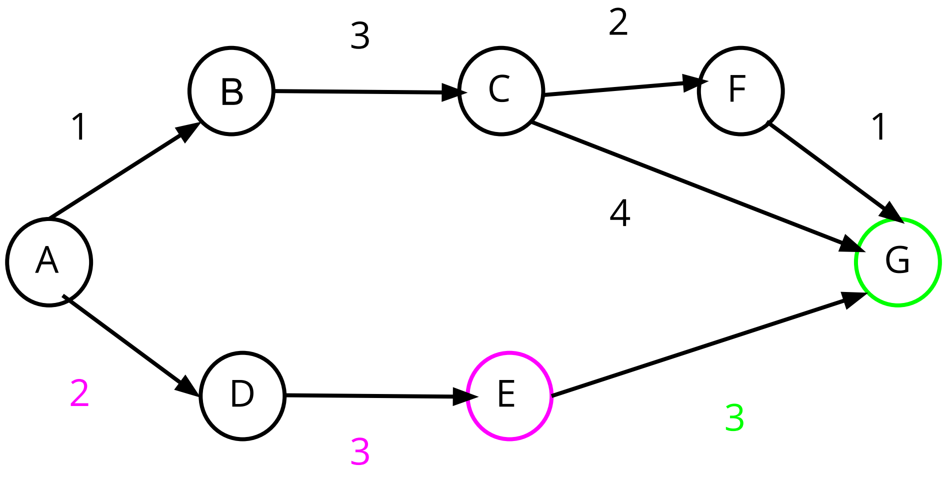
Q2.1: Dijkstra's Algorithm Solution

Pop E.



Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
~~E: 5~~	E: 5	D
F: 6	F: 6	C
G: 8	G: 8	C

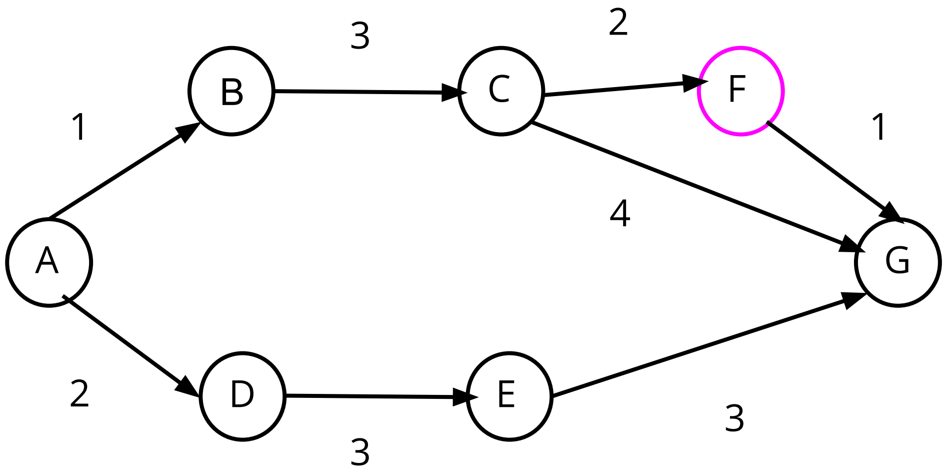
Q2.1: Dijkstra's Algorithm Solution



Potential distance to G is 8, which is the same. Don't change priority.

Priority Queue	**Distances**	**edgeTo**
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
~~E: 5~~	E: 5	D
F: 6	F: 6	C
G: 8	G: 8	C

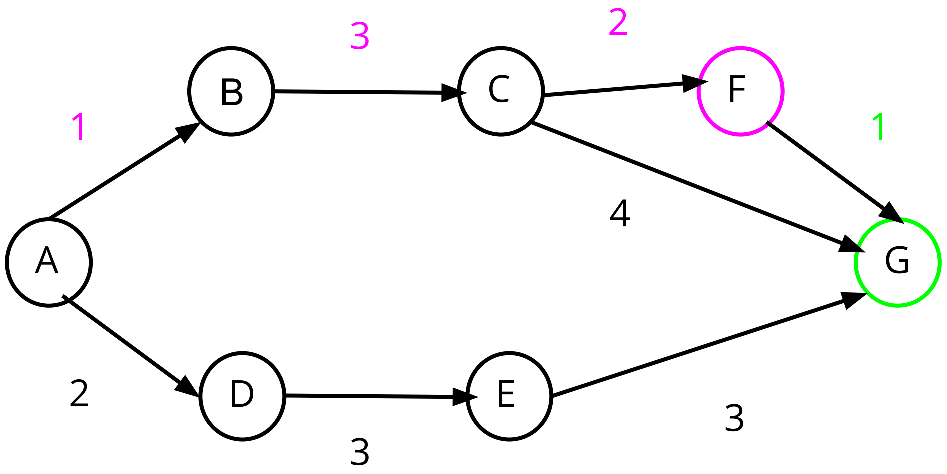
Q2.1: Dijkstra's Algorithm Solution



Pop F.

Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
~~E: 5~~	E: 5	D
~~F: 6~~	F: 6	C
G: 8	G: 8	C

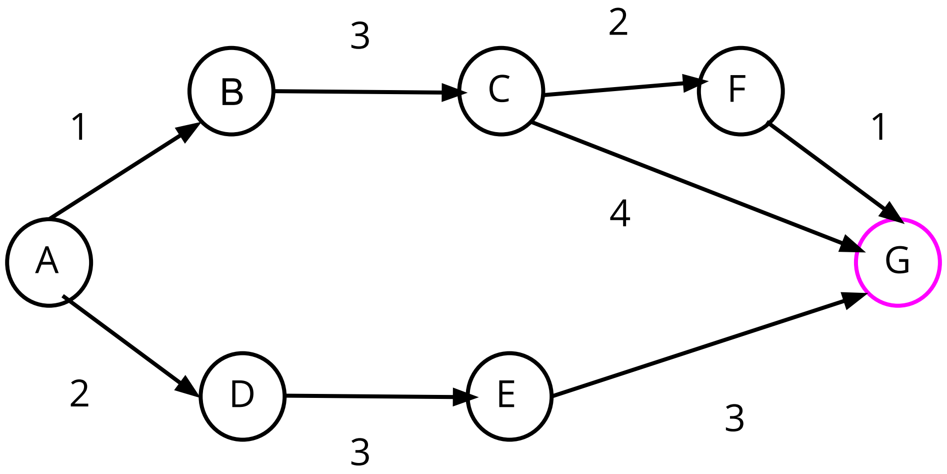
Q2.1: Dijkstra's Algorithm Solution



Potential distance to G is 7, changePriority(G, 7).

Priority Queue	**Distances**	**edgeTo**
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
~~E: 5~~	E: 5	D
~~F: 6~~	F: 6	C
G: 8	G: 8	C

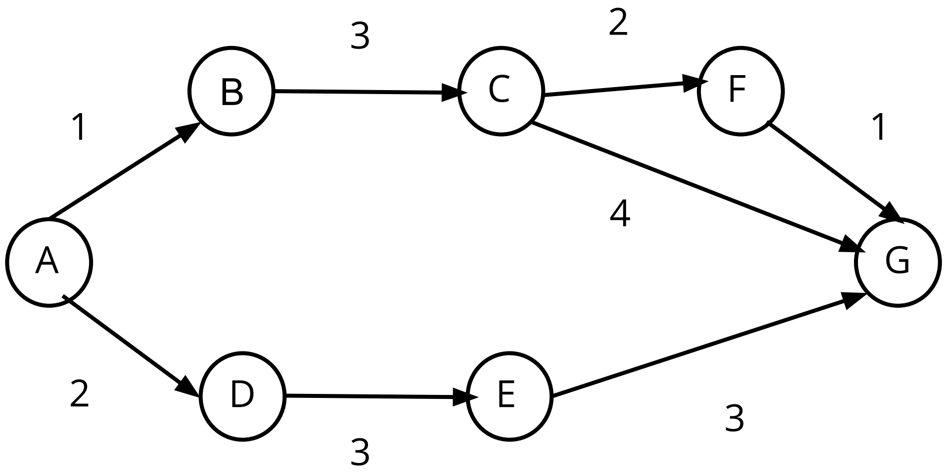
Q2.1: Dijkstra's Algorithm



Pop G. Now we have the shortest paths from A to every other vertex!

Priority Queue	Distances	edgeTo
~~A: 0~~	A: 0	—
~~B: 1~~	B: 1	A
~~D: 2~~	D: 2	A
~~C: 4~~	C: 4	B
~~E: 5~~	E: 5	D
~~F: 6~~	F: 6	C
~~G: 7~~	G: 7	F

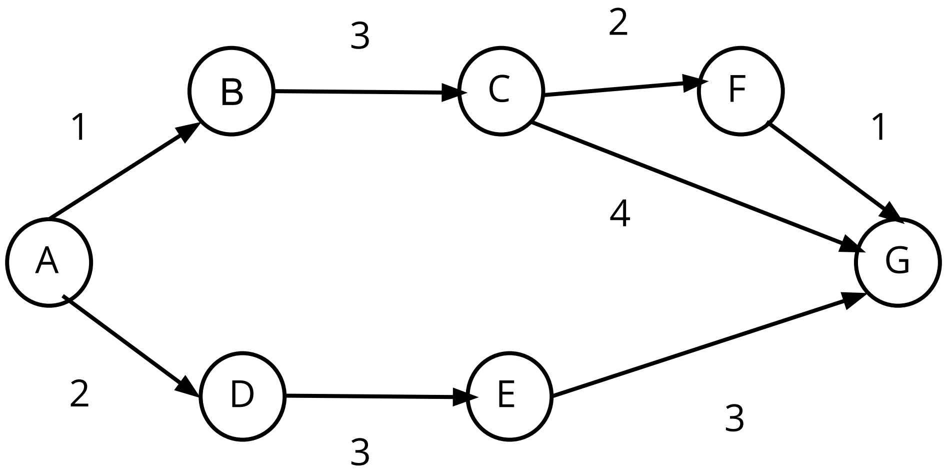
Q2.1: Dijkstra's Algorithm Solution



Pop A.

Priority Queue	Distances
~~A: 0~~	A: 0
B: inf	
C: inf	
D: inf	
E: inf	
F: inf	
G: inf	

Q2.1: Dijkstra’s Algorithm Solution

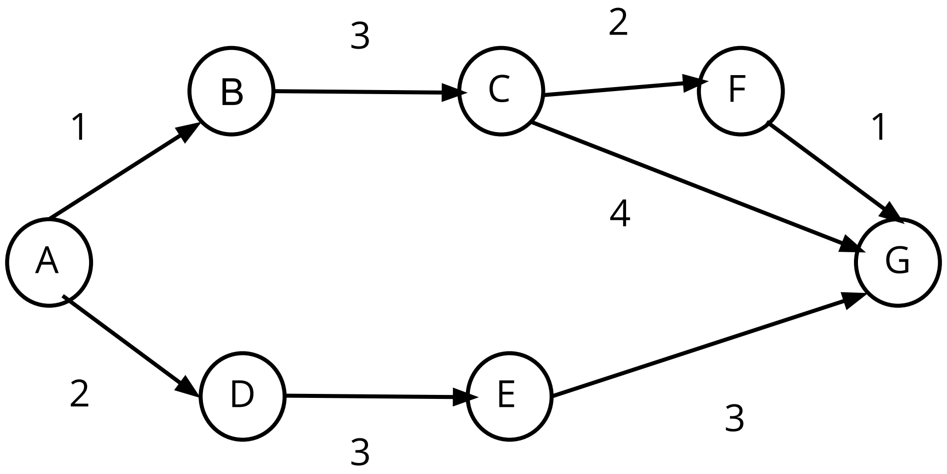


changePriority(B, 1),
changePriority(D, 2)

Priority Queue	Distances
~~A: 0~~	A: 0
B: 1	
D: 2	
C: inf	
E: inf	
F: inf	
G: inf	

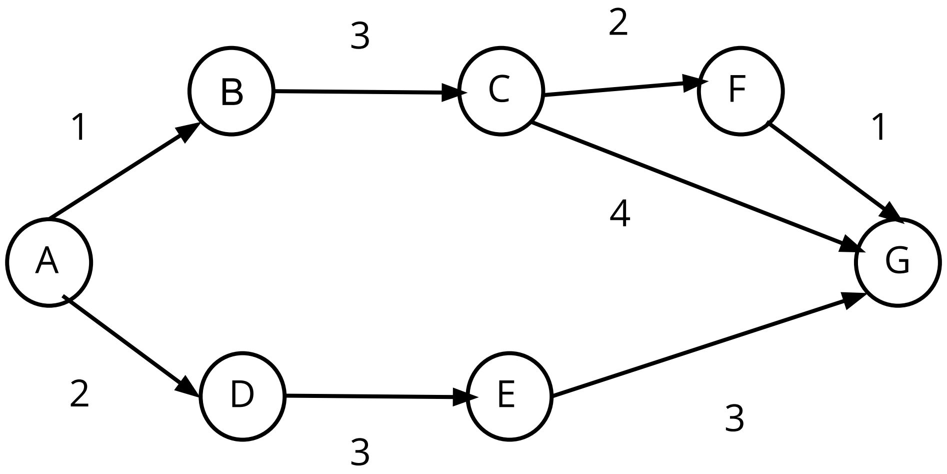
Q2.1: Dijkstra's Algorithm Solution

Pop B.



Priority Queue	**Distances**
~~A: 0~~	A: 0
~~B: 1~~	B: 1
D: 2	
C: inf	
E: inf	
F: inf	
G: inf	

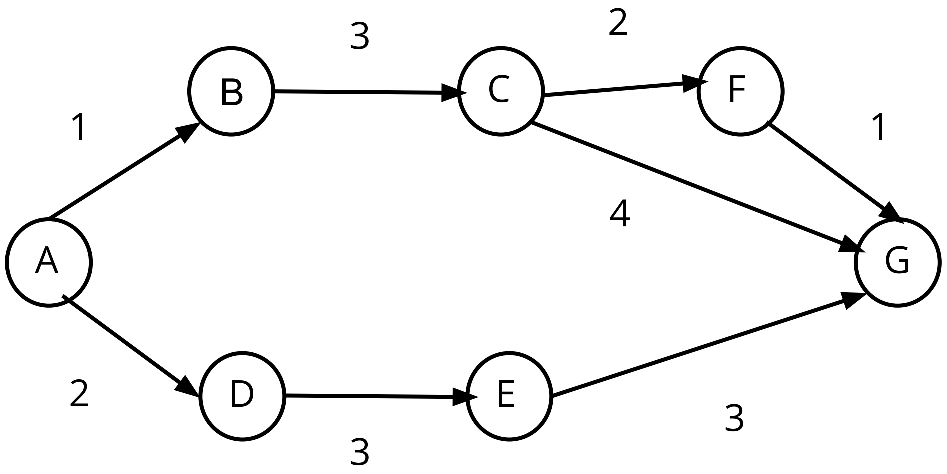
Q2.1: Dijkstra's Algorithm Solution



changePriority(C, 4)

Priority Queue	Distances
~~A: 0~~	A: 0
~~B: 1~~	B: 1
D: 2	
C: 4	
E: inf	
F: inf	
G: inf	

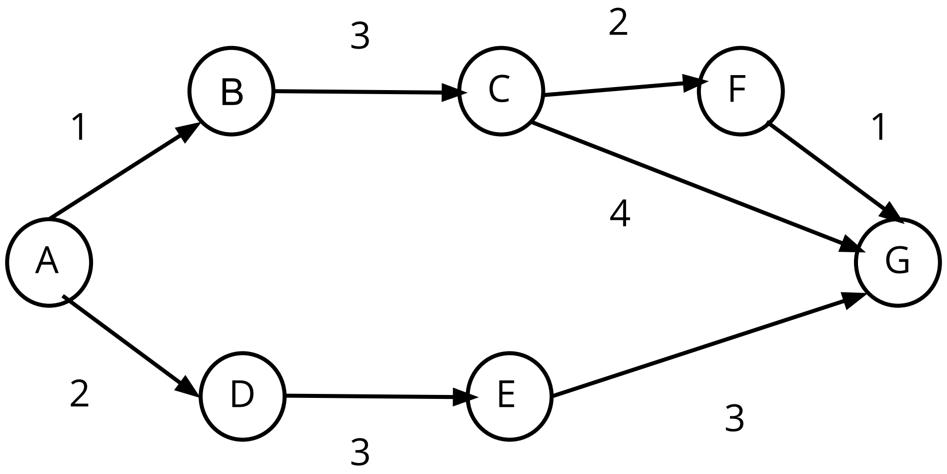
Q2.1: Dijkstra's Algorithm Solution



Pop D.

Priority Queue	**Distances**
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
C: 4	
E: inf	
F: inf	
G: inf	

Q2.1: Dijkstra's Algorithm Solution

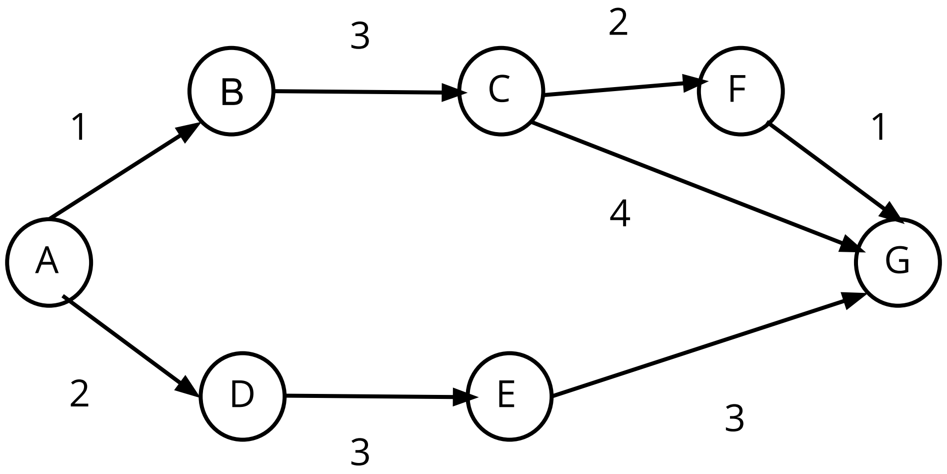


changePriority(E, 5)

Priority Queue	Distances
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
C: 4	
E: 5	
F: inf	
G: inf	

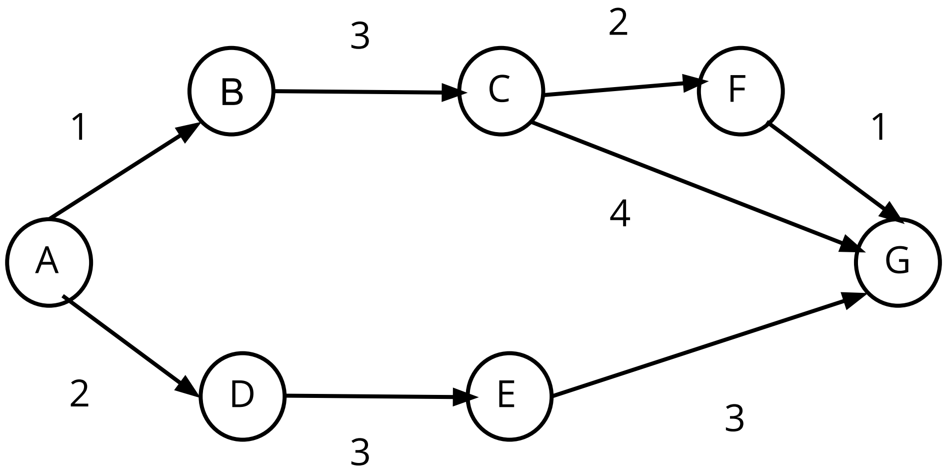
Q2.1: Dijkstra's Algorithm Solution

Pop C.



Priority Queue	Distances
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
E: 5	
F: inf	
G: inf	

Q2.1: Dijkstra's Algorithm Solution

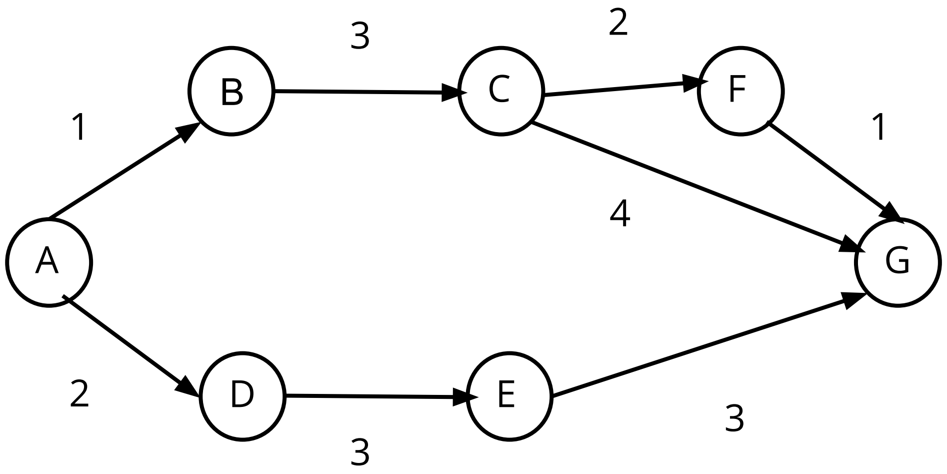


changePriority(F, 6),
changePriority(G, 8)

Priority Queue	**Distances**
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
E: 5	
F: 6	
G: 8	

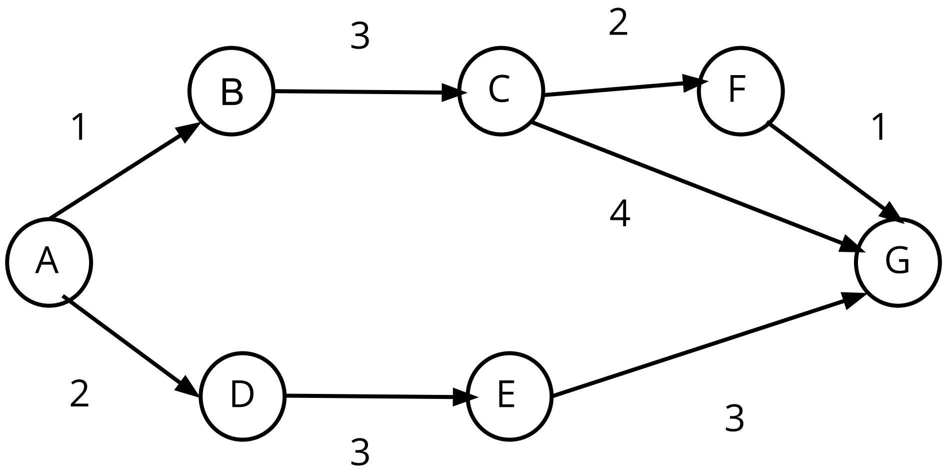
Q2.1: Dijkstra's Algorithm Solution

Pop E.



Priority Queue	**Distances**
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
~~E: 5~~	E: 5
F: 6	
G: 8	

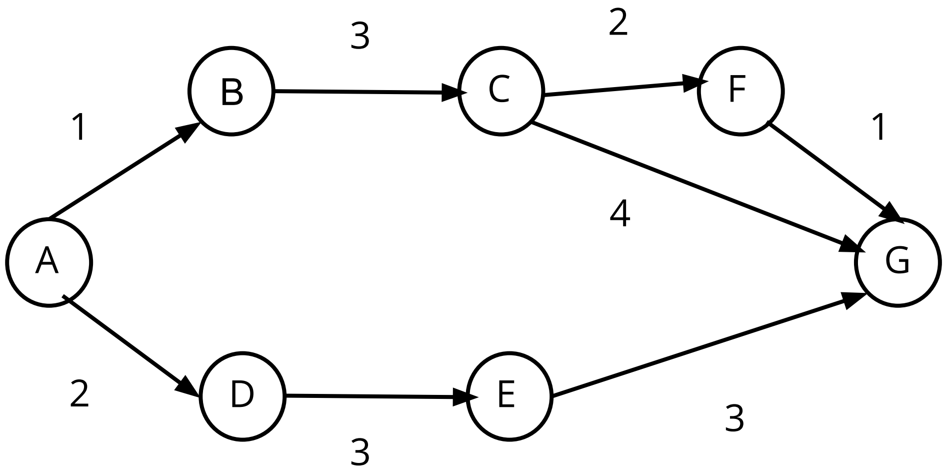
Q2.1: Dijkstra’s Algorithm Solution



Potential distance to G is 8, which is the same. Don’t change priority.

Priority Queue	**Distances**
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
~~E: 5~~	E: 5
F: 6	
G: 8	

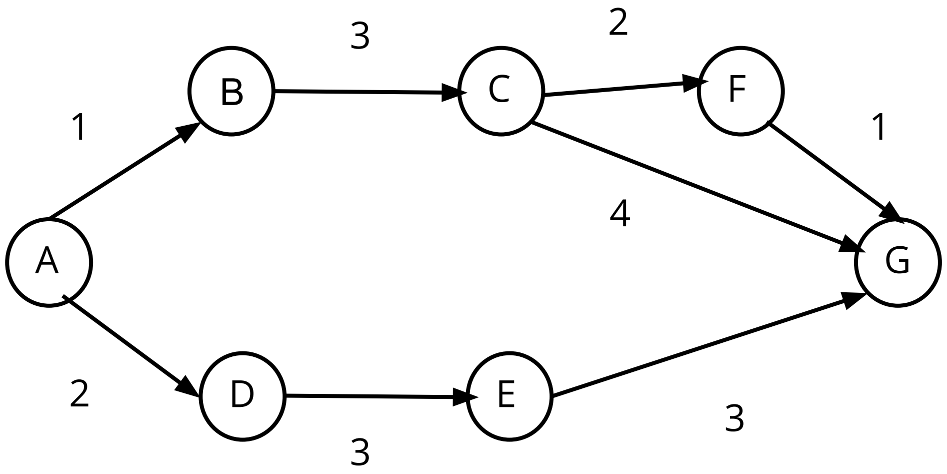
Q2.1: Dijkstra's Algorithm Solution



Pop F.

Priority Queue	Distances
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
~~E: 5~~	E: 5
~~F: 6~~	F: 6
G: 8	

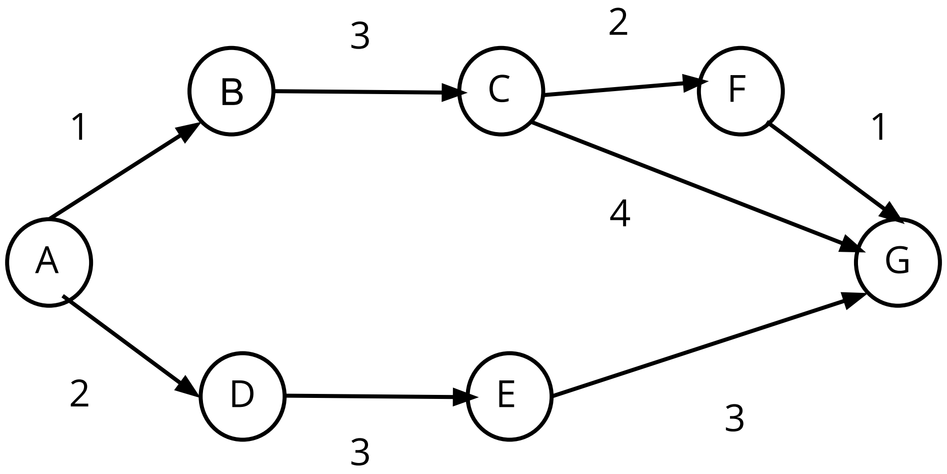
Q2.1: Dijkstra's Algorithm Solution



Potential distance to G is 7, changePriority(G, 7).

Priority Queue	Distances
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
~~E: 5~~	E: 5
~~F: 6~~	F: 6
G: 7	

Q2.1: Dijkstra's Algorithm



Pop G. Now we have the shortest paths from A to every other vertex!

Priority Queue	**Distances**
~~A: 0~~	A: 0
~~B: 1~~	B: 1
~~D: 2~~	D: 2
~~C: 4~~	C: 4
~~E: 5~~	E: 5
~~F: 6~~	F: 6
~~G: 7~~	G: 7

A* Search

- Dijkstra’s isn’t always optimal if we just want to find the shortest path between point A and point B.

- A* uses a heuristic, which can be thought of our sense as to whether we think a point is closer or not.
  - These heuristics do not change as the search runs. You’ll learn more about the quality of a heuristic in 188, but Professor Hug previewed it in lecture.

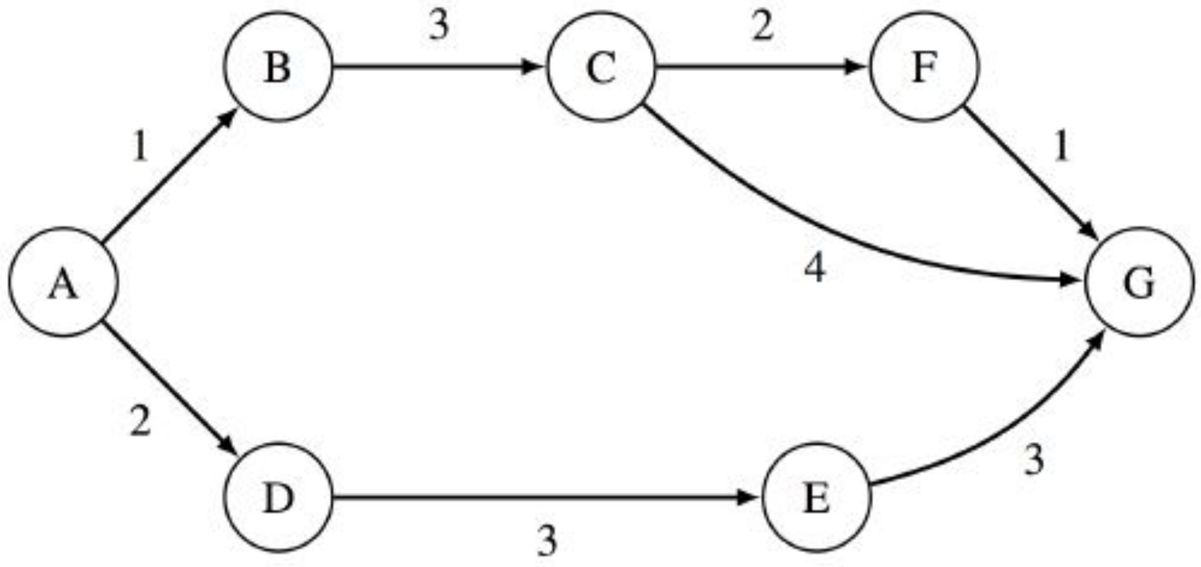
A* Search

- Main Idea
    - Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).
    - Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.
    - Single source, single target

- A* Demo Link (from lecture)

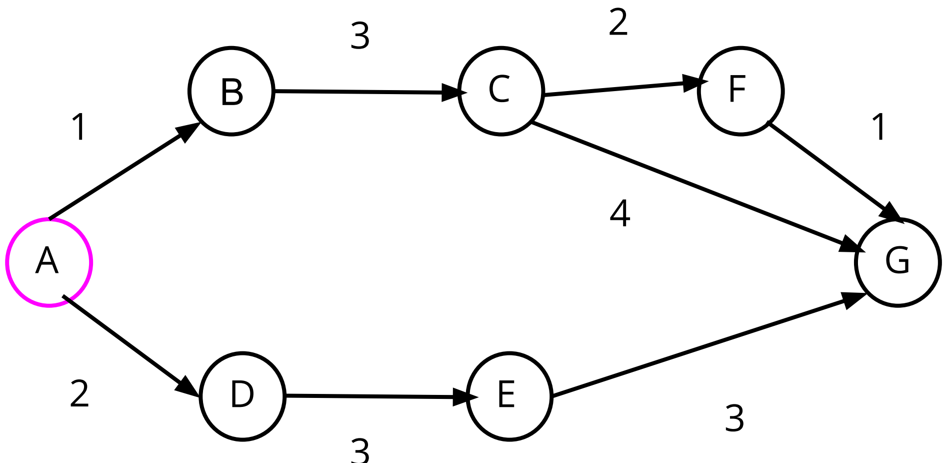
Q2.2: A* Search

What would A* return, starting with A and ending with G?



Edge weights	Heuristics
$g(A,B) = 1$	$h(A,G) = 8$
$g(B,C) = 3$	$h(B,G) = 6$
$g(C,F) = 4$	$h(C,G) = 5$
$g(C,G) = 4$	$h(F,G) = 1$
$g(F,G) = 1$	$h(D,G) = 6$
$g(A,D) = 2$	$h(E,G) = 3$
$g(D,E) = 3$	
$g(E,G) = 3$	

Q2.1: A* Search Solutions



Heritage

Heuristics

h(A,G) = 8

h(B,G) = 6

h(C,G) = 5

h(F,G) = 1

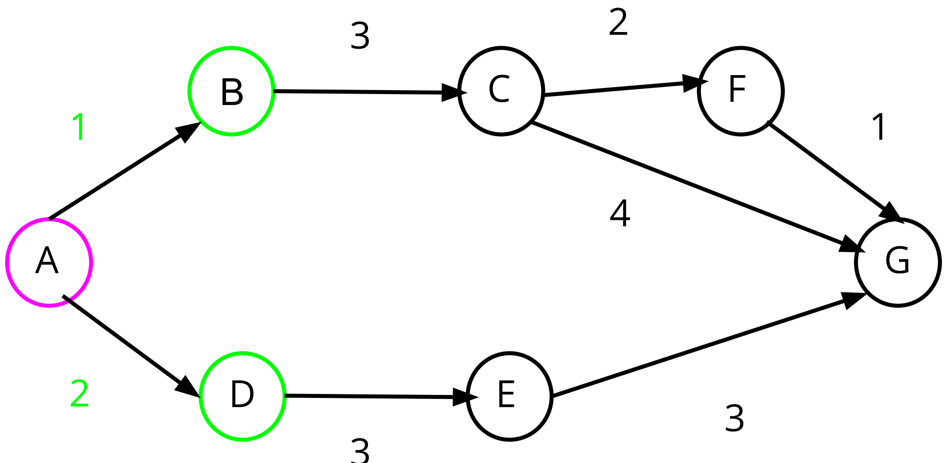
h(D, G) = 6

h(E,G) = 3

Pop A.

Priority Queue	Distances	edgeTo
~~A: 0 + 8~~	A: 0	—
B: inf		
C: inf		
D: inf		
E: inf		
F: inf		
G: inf		

Q2.1: A* Search Solutions



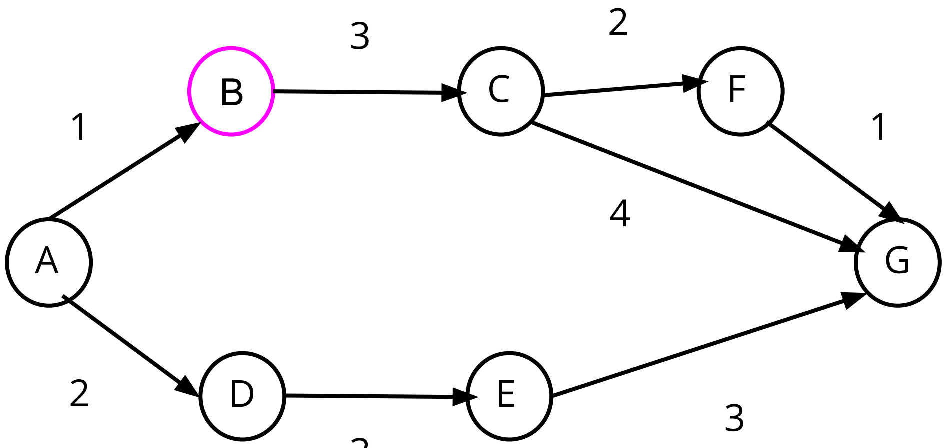
Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

changePriority(B, 7),
changePriority(D, 8)

Priority Queue	**Distances**	**edgeTo**
~~A: 0~~	A: 0	—
B: 1 + 6 = 7	B: 1	A
D: 2 + 6 = 8	D: 2	A
C: inf		
E: inf		
F: inf		
G: inf		

Q2.1: A* Search Solutions

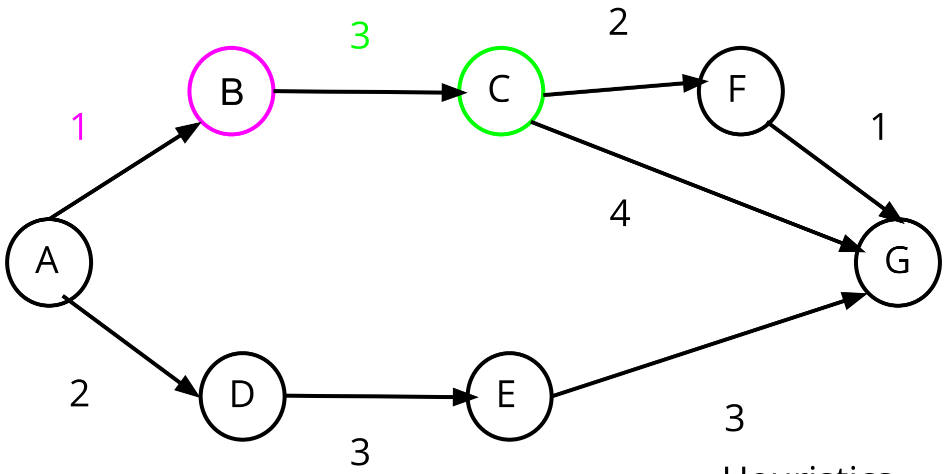
Pop B.



Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

Priority Queue	Distances	edgeTo
~~A: 0 + 8 = 8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
D: 2 + 6 = 8	D: 2	A
C: inf		
E: inf		
F: inf		
G: inf		

Q2.1: A* Search Solutions



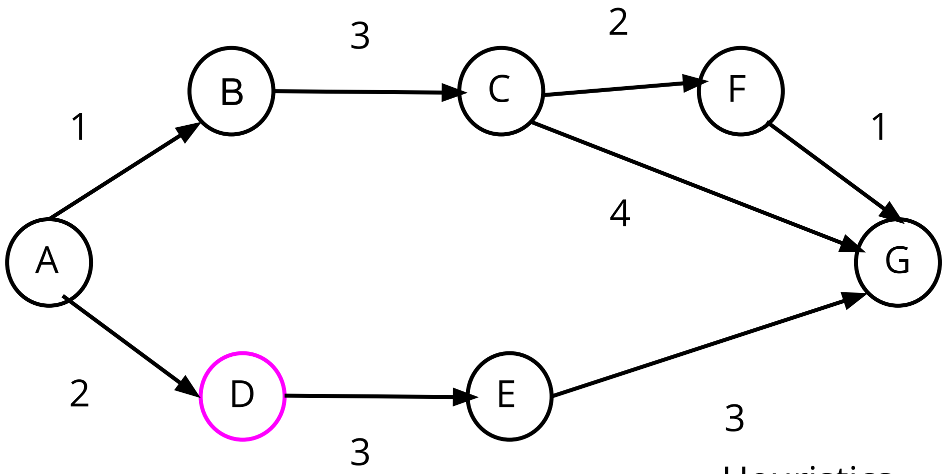
Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

changePriority(C, 9)

Priority Queue	**Distances**	**edgeTo**
~~A: 0 + 8 =8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
D: 2 + 6 = 8	D: 2	A
C: 4 + 5 = 9	C: 4	B
E: inf		
F: inf		
G: inf		

Q2.1: A* Search Solution

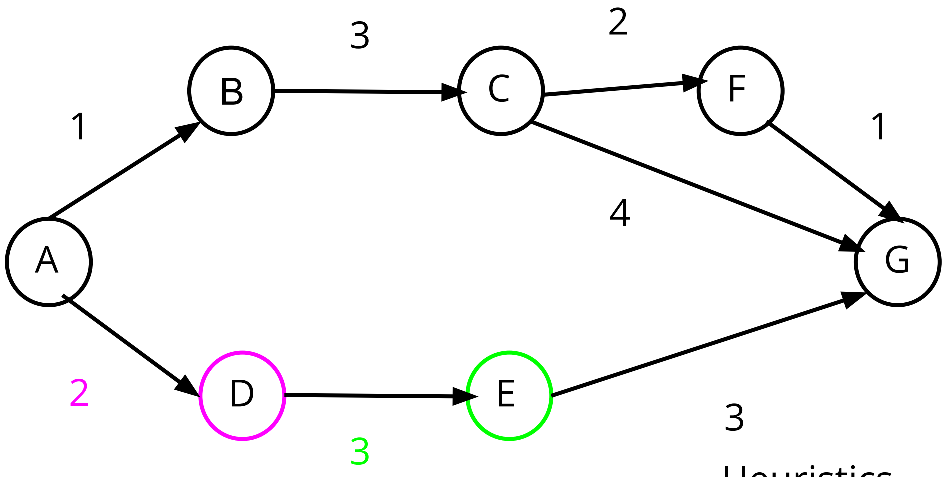
Pop D.



Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

Priority Queue	**Distances**	**edgeTo**
~~A: 0 + 8 = 8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
~~D: 2 + 6 = 8~~	D: 2	A
C: 4 + 5 = 9	C: 4	B
E: inf		
F: inf		
G: inf		

Q2.1: A* Search Solution



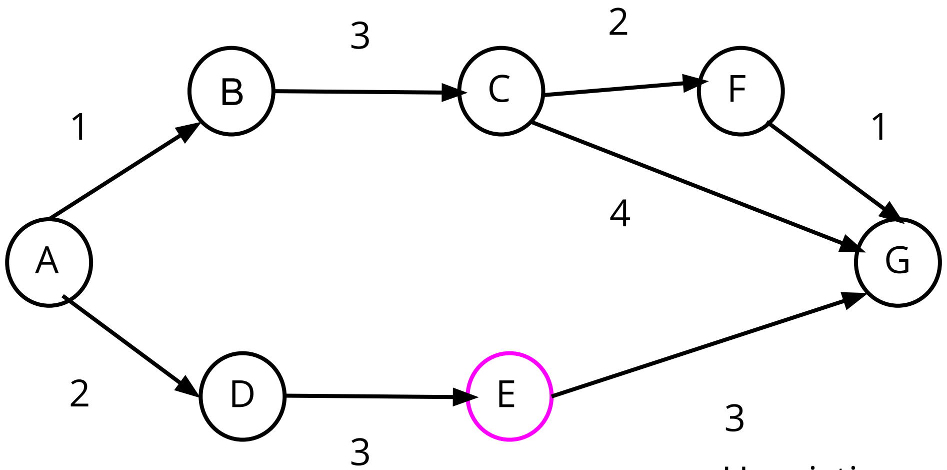
Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

changePriority(E, 8)

Priority Queue	**Distances**	**edgeTo**
~~A: 0 + 8 = 8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
~~D: 2 + 6 = 8~~	D: 2	A
C: 4 + 5 = 9	C: 4	B
E: 5 + 3 = 8	E: 5	D
F: inf		
G: inf		

Q2.1: A* Search Solution

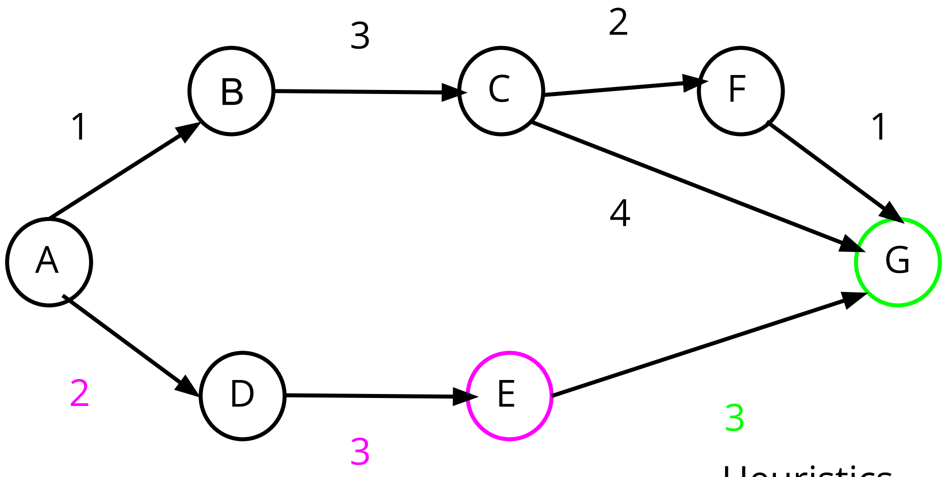
Pop E.



<u>Heuristics</u>
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

Priority Queue	**Distances**	**edgeTo**
~~A: 0 + 8 = 8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
~~D: 2 + 6 = 8~~	D: 2	A
C: 4 + 5 = 9	C: 4	B
~~E: 5 + 3 = 8~~	E: 5	D
F: inf		
G: inf		

Q2.1: A* Search Solution



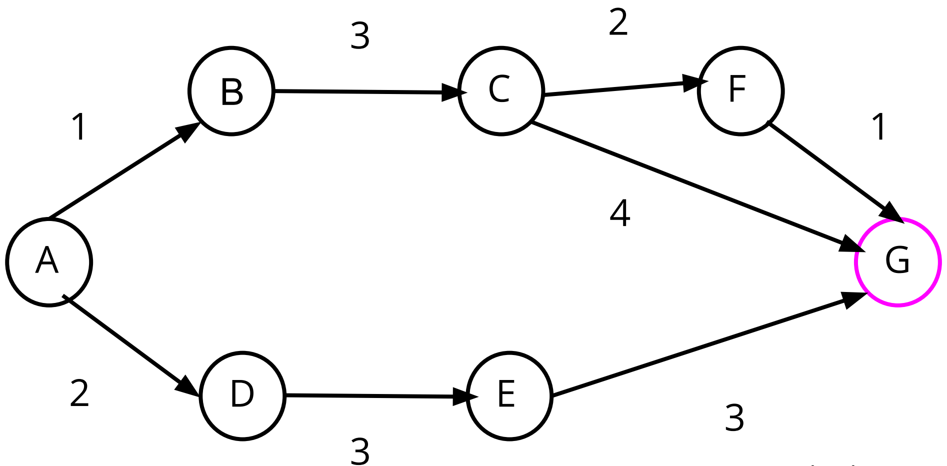
Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

changePriority(G, 8)

Priority Queue	**Distances**	**edgeTo**
~~A: 0 + 8 = 8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
~~D: 2 + 6 = 8~~	D: 2	A
C: 4 + 5 = 9		B
~~E: 5 + 3 = 8~~	E: 5	D
F: inf		
G: 8 + 0 = 8	G: 8	E

Q2.1: A* Search Solution

Pop G.



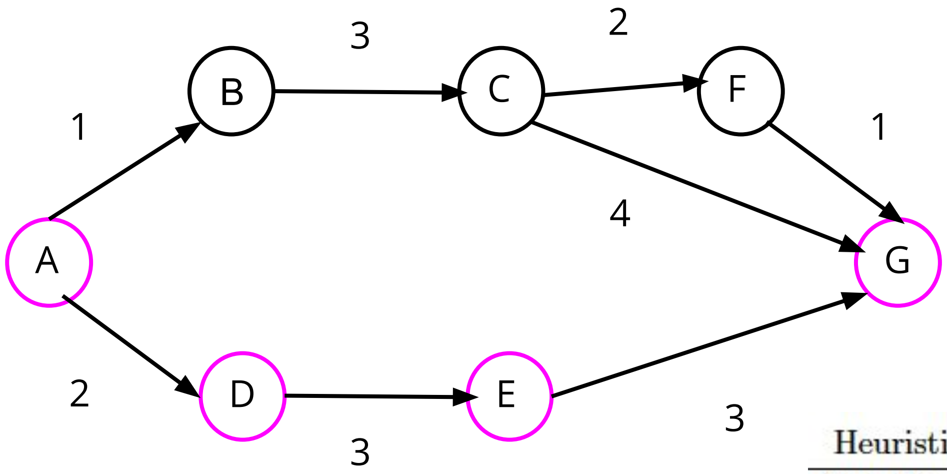
Heuristics
h(A,G) = 8
h(B,G) = 6
h(C,G) = 5
h(F,G) = 1
h(D, G) = 6
h(E,G) = 3

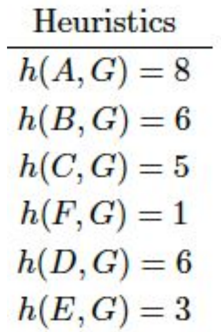
Priority Queue	Distances	edgeTo
~~A: 0 + 8 = 8~~	A: 0	—
~~B: 1 + 6 = 7~~	B: 1	A
~~D: 2 + 6 = 8~~	D: 2	A
C: 4 + 5 = 9		B
~~E: 5 + 3 = 8~~	E: 5	D
F: inf		
~~G: 5 + 3 = 8~~	G: 8	E

Q2.2: A* Search Solution

A* would return A-D-E-G.

We are Done!

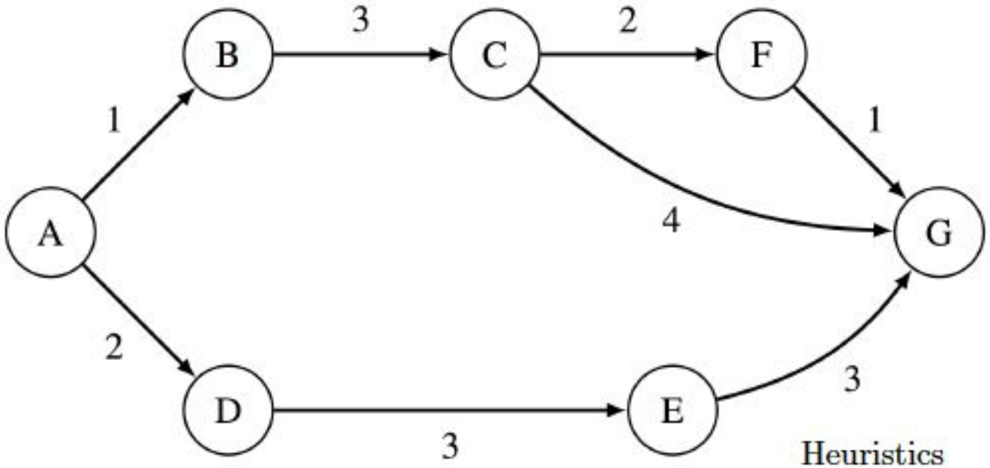




Heuristics
$h(A,G) = 8$
$h(B,G) = 6$
$h(C,G) = 5$
$h(F,G) = 1$
$h(D,G) = 6$
$h(E,G) = 3$

Priority Queue	**Distances**	**edgeTo**
~~A: 0 + 8 = 8~~	**A: 0**	—
~~B: 1 + 6 = 7~~	B: 1	A
~~D: 2 + 6 = 8~~	**D: 2**	**A**
C: 4 + 5 = 9		B
~~E: 5 + 3 = 8~~	**E: 5**	**D**
F: inf		
~~G: 5 + 3 = 8~~	**G: 8**	**E**

Q2.2: A* Search Solution



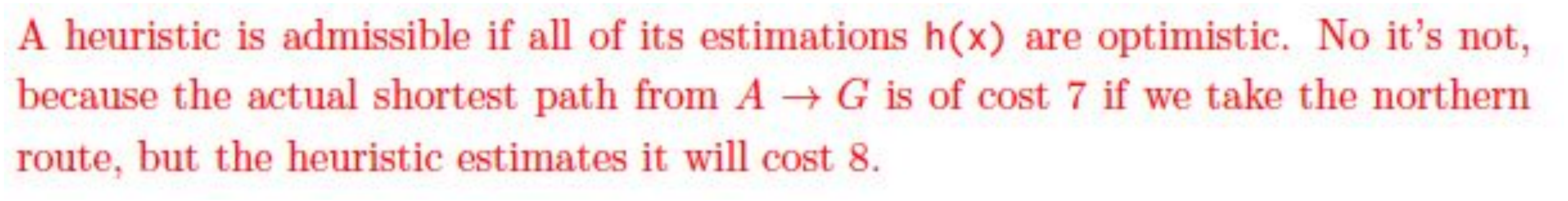
Heuristics
$h(A,G) = 8$
$h(B,G) = 6$
$h(C,G) = 5$
$h(F,G) = 1$
$h(D,G) = 6$
$h(E,G) = 3$

A* runs in a very similar fashion to Dijkstra's. The only difference is the priority in the priority queue. For A*, whenever computing the priority (for the purposes of the priority queue) of a particular node n, always add h(n) to whatever you would use with Dijkstra's.

A* would return A-D-E-G.

Q2.3 Solution

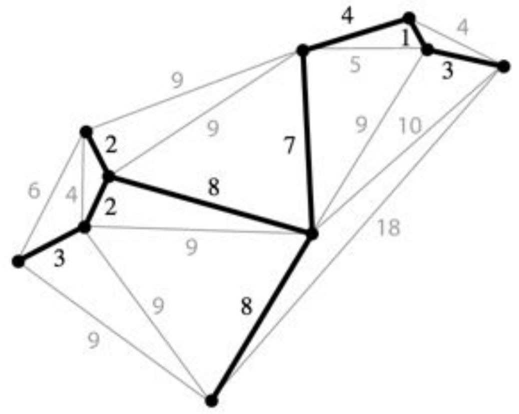
Is the heuristic admissible? Why or why not?



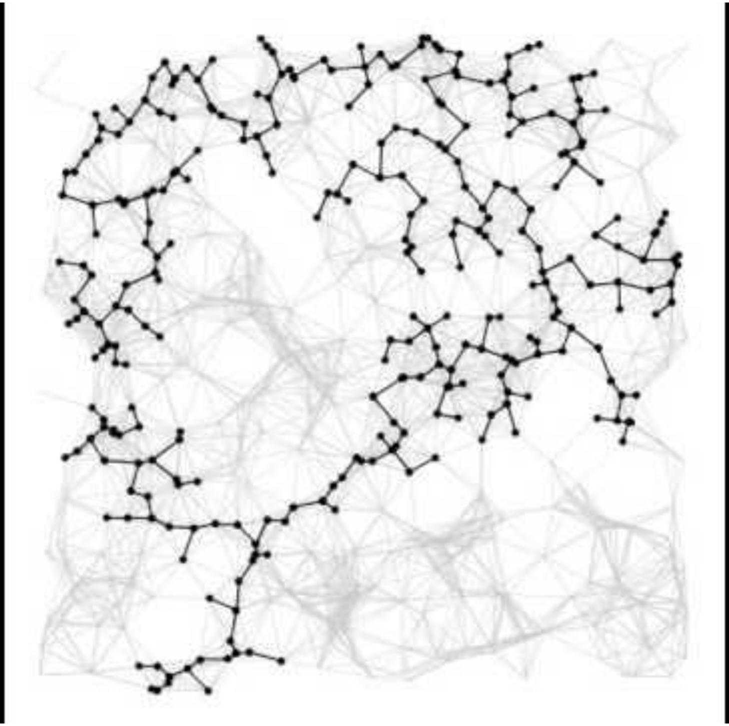
A heuristic is admissible if all of its estimations h(x) are optimistic. No it's not, because the actual shortest path from $A \rightarrow G$ is of cost 7 if we take the northern route, but the heuristic estimates it will cost 8.

Minimum Spanning Trees (MSTs)

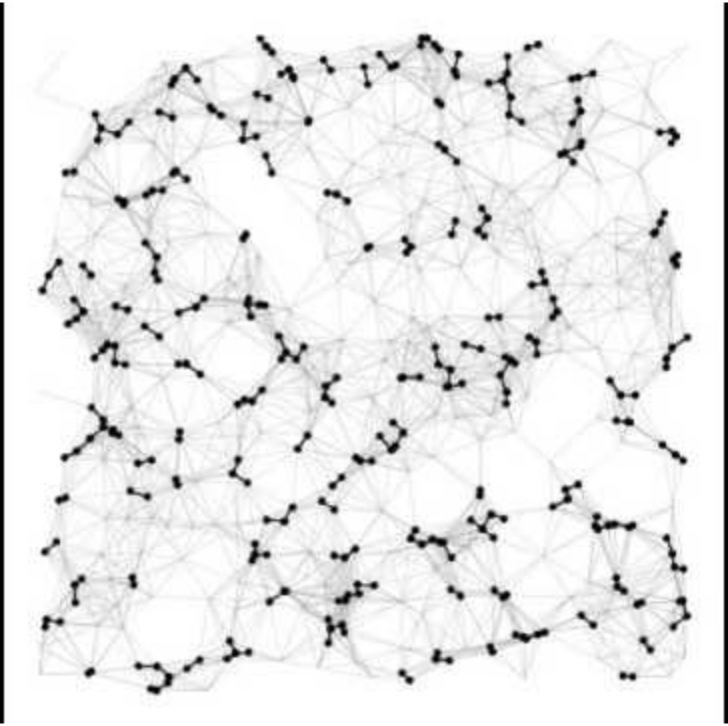
- Given an undirected graph G, an MST is the subgraph of G that is a tree of minimum total weight that includes all of its vertices (this is the spanning part)
  - Note that a tree is has to be acyclic (no cycles), and connected
- The uniqueness of an MST is only guaranteed if the graph has distinct edges
- Spanning tree is outlined in black here



Prim’s vs. Kruskal’s



Prim’s



Kruskal’s

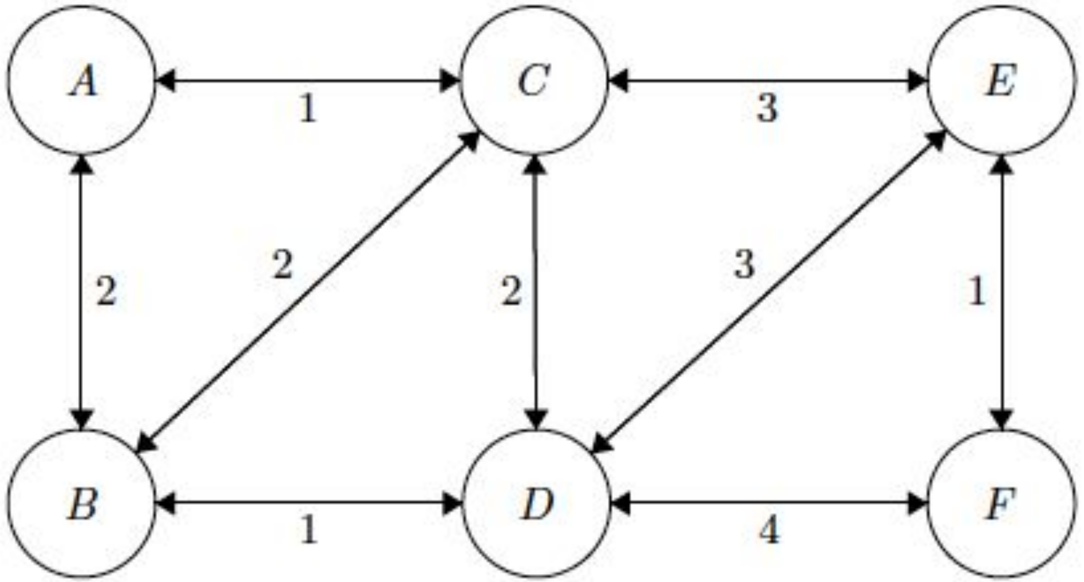
Prim’s vs. Kruskal’s

- They are both ways of finding an MST, but they don’t always output the same MST

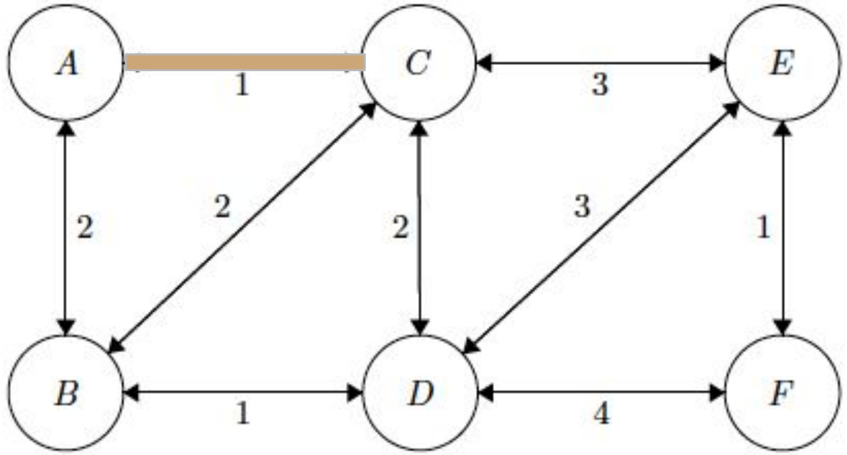
- Prim's attaches a new edge to a single growing tree at each step (another way to think of it is that it engulfs nodes)
- On the other hand, Kruskal’s processes edges in the order of their edge weights at each step, and takes for the MST each edge that does not form a cycle with edges previously added.

Q3.1

Perform Prim's algorithm to find the minimum spanning tree. Pick *A* as the initial node. Whenever there is more than one node with the same cost, process them in alphabetical order.

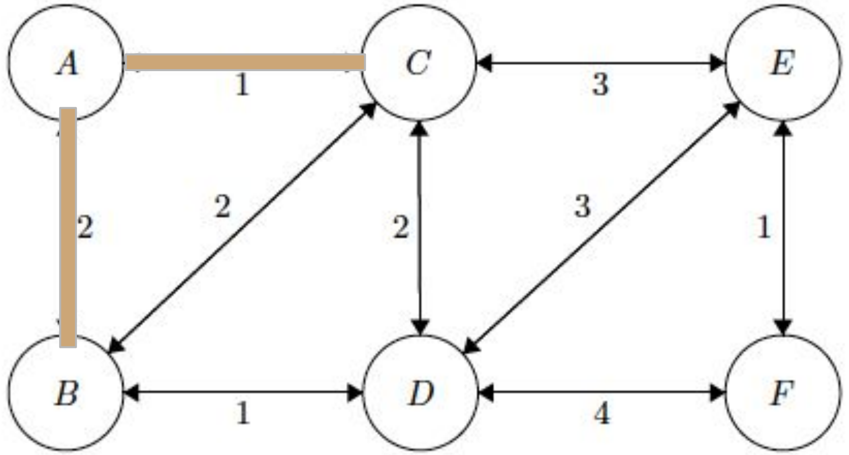


Q3.1 Solution



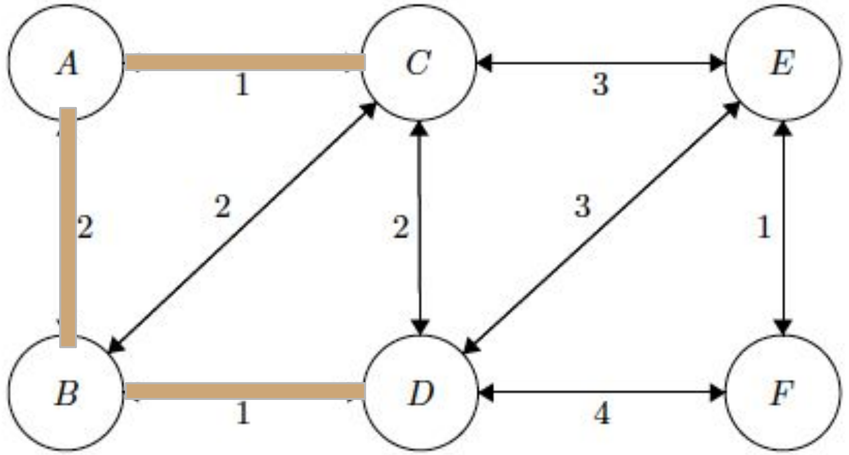
- Begin by engulfing A.
- Our current tree is A. Edges going out from our current tree are AC and AB.
- The cheapest edge from that tree is AC, so we engulf C.

Q3.1 Solution



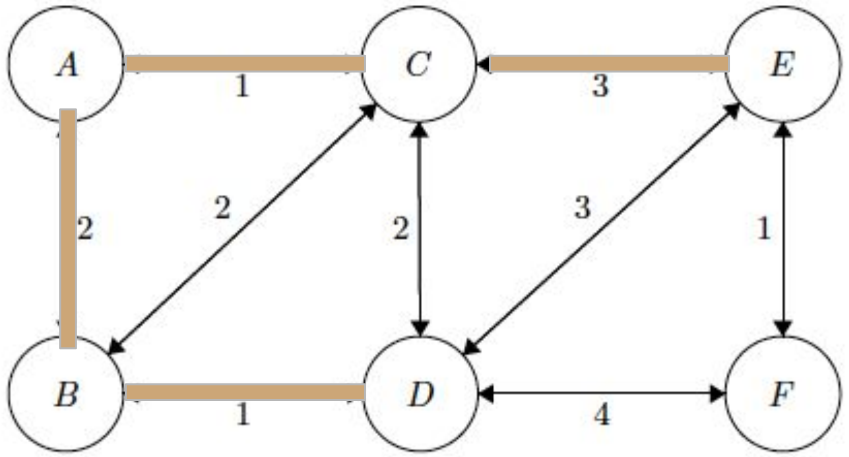
- Now our current tree is AC.
- Edges going out of AC are AB, BC, CD, and CE
- The cheapest edge is AB, so we engulf B.

Q3.1 Solution



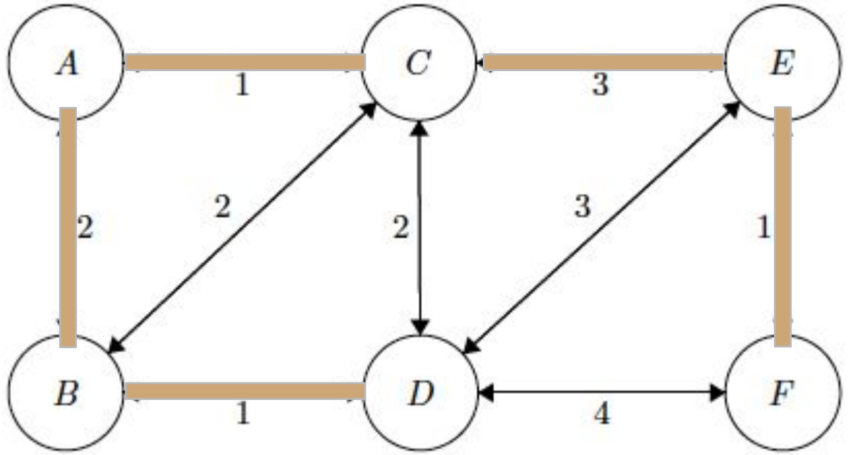
- Now our current tree is ABC.
- Edges going out of ABC are BD, CD and CE.
- The cheapest is BD, so we engulf D.

Q3.1 Solution



- Now our current tree is ABCD.
- Edges going out of ABCD are CE, DE, and DF.
- The cheapest and lexicographically smallest edge is CE, so we engulf E.

Q3.1 Solution

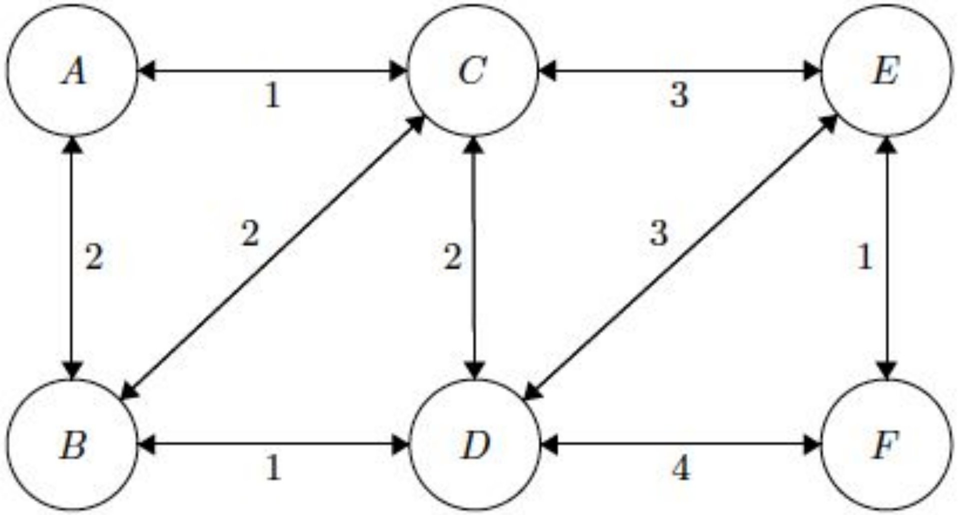


- Finally, we take EF.
- Our MST is outlined in brown.

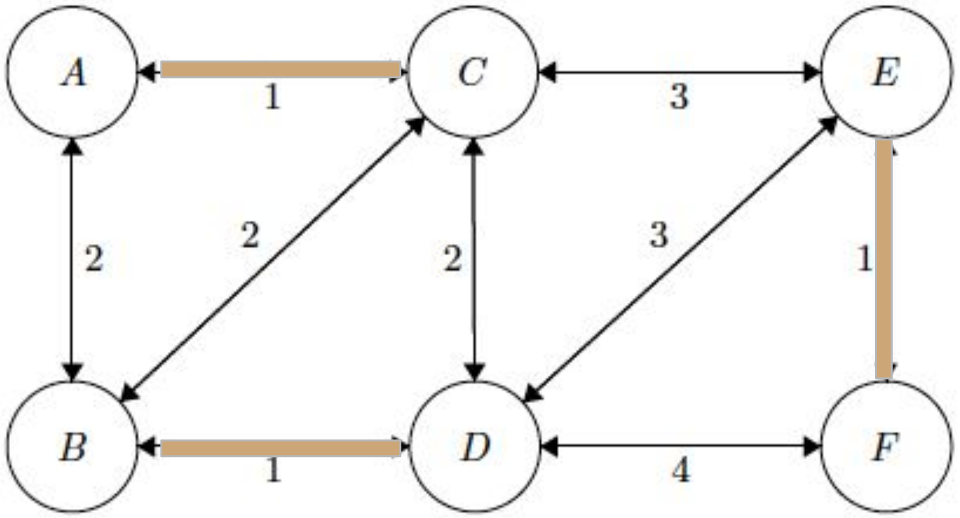
Q3.2

Use Kruskal's algorithm to find a minimum spanning tree. When deciding between equiweighted edges, alphabetically sort the edge, and then pick in lexicographic order.

For instance, edges are always written as AB or AC, never BA or CA. If deciding between AB and AC, pick AB first.

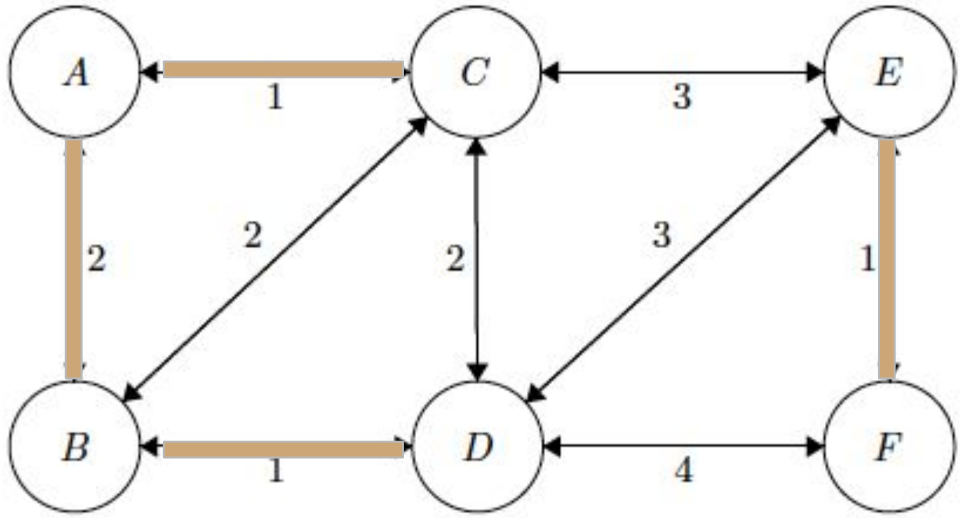


Q3.2



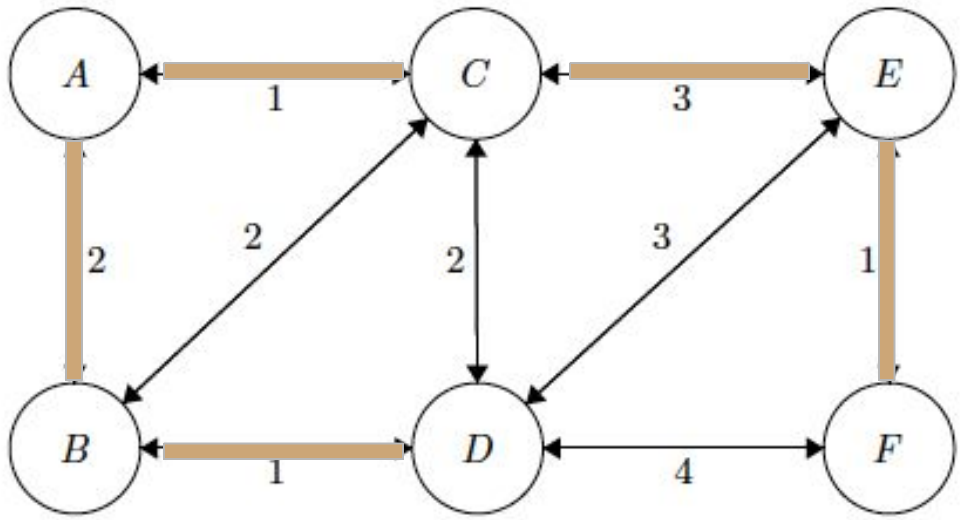
- Kruskal's considers edges of weight 1 first (AC, BD, EF)
- It picks AC, then BD, then EF since picking all 3 does not create a cycle

Q3.2



- Kruskal's considers edges of weight 2 next (AB, BC, CD)
- It adds AB b/c it doesn't create a cycle.
- It skips BC b/c it creates a cycle
- It skips CD b/c it creates a cycle

Q3.2



- Kruskal’s now considers edges of weight 3 (CE, DE)
- It adds CE b/c it doesn’t create a cycle.
- We stop b/c we now have a spanning tree
- The full MST is outlined in brown