Shortest Paths and MSTs

Exam Prep 09

Announcements

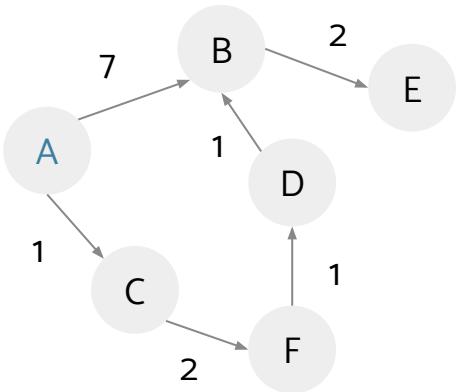
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	3/18 Homework 3 Due			3/21 **Midterm 2**		

Content Review

Dijkstra's Algorithm

We've learned that BFS can help us find paths from the start to other nodes with a minimum number of edges. However, neither BFS or DFS account for finding shortest paths based off edge weight.

Dijkstra's algorithm is a method of finding the shortest path from one node to every other node in the graph. You use a priority queue that sorts vertices based off of their distance to the root node.

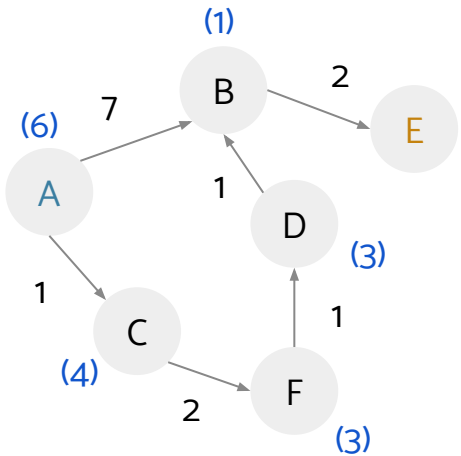


Steps:

1. Pop node from the front of the queue - this is the current node.
2. Add/update distances of all of the neighbors of the current node in the PQ.
3. Re-sort the priority queue (technically the PQ does this itself).
4. Finalize the distance to the current node from the root.
5. Repeat while the PQ is not empty.

A*

A* is a method of finding the shortest path from one node to a specific other node in the graph. It operates similarly to Dijkstra's except for that we use a (given) heuristic to estimate a vertex's distance from the goal.



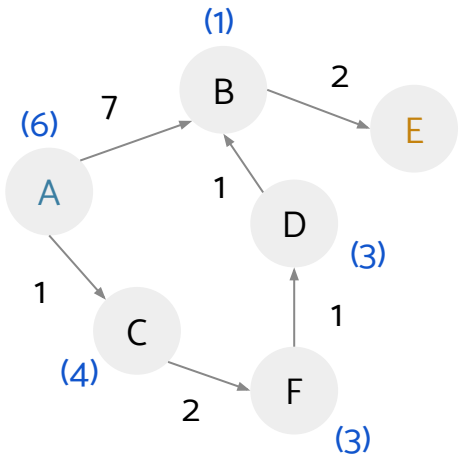
We're guaranteed to get the shortest path if our heuristic is admissible (never overestimates the true distance to the goal) and consistent (estimate always <= the estimated distance from any neighboring vertex to the goal + the cost of reaching that neighbor).

A*

A* is a method of finding the shortest path from one node to a specific other node in the graph. It operates similarly to Dijkstra's except for that we use a (given) heuristic to estimate a vertex's distance from the goal.

Steps:

1. Pop node from the top of the queue - this is the current node.
2. Add/update distances of all of the children of the current node. This distance will be the sum of the distance up to that child node and our guess of how far away the goal node is (our heuristic).
3. Re-sort the priority queue.
4. Check if we've hit the goal node (if so we stop).
5. Repeat while the PQ is not empty.



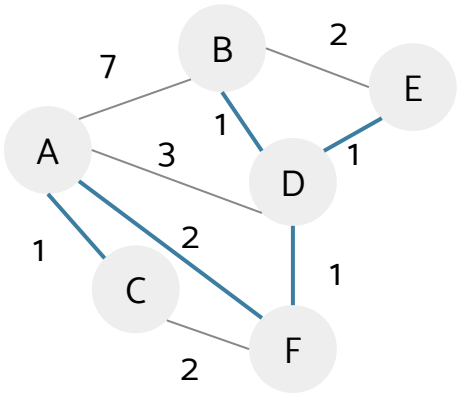
Note: the heuristic may not always be very good and might lead us down a path that isn't the shortest!

Minimum Spanning Trees

Minimum Spanning Trees are set of edges that connect all the nodes in a graph while being of the smallest possible weight.

MSTs may not be unique if there are multiple edges of the same weight.

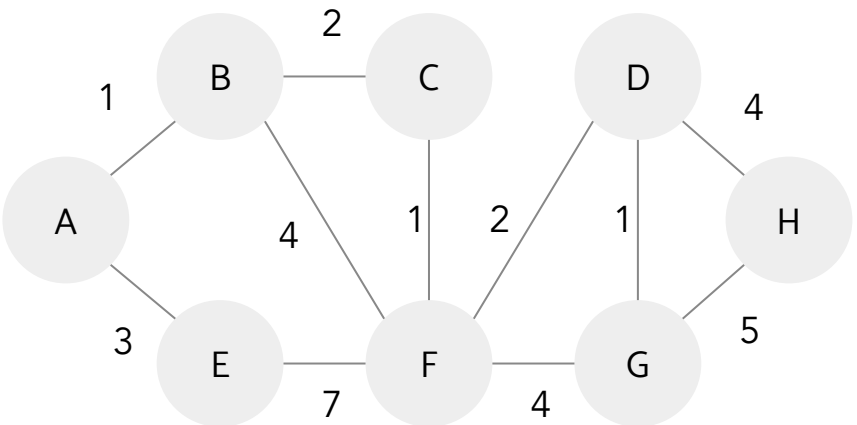
There are two main algorithms for finding MSTs in this class: Prim's and Kruskal's. Both are based on the **cut property**: if we "cut" across any edges and separate the graph into two groups, the minimum weight edge that falls along that cut will be in some MST.



Worksheet

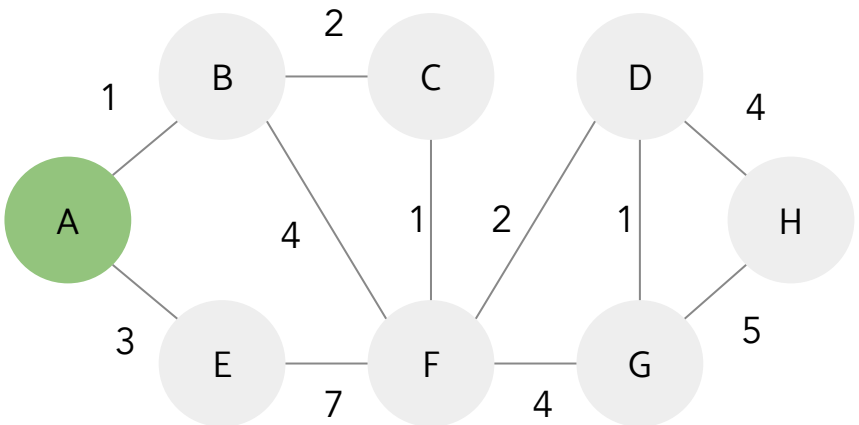
1A Dijkstra's, A*

	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞



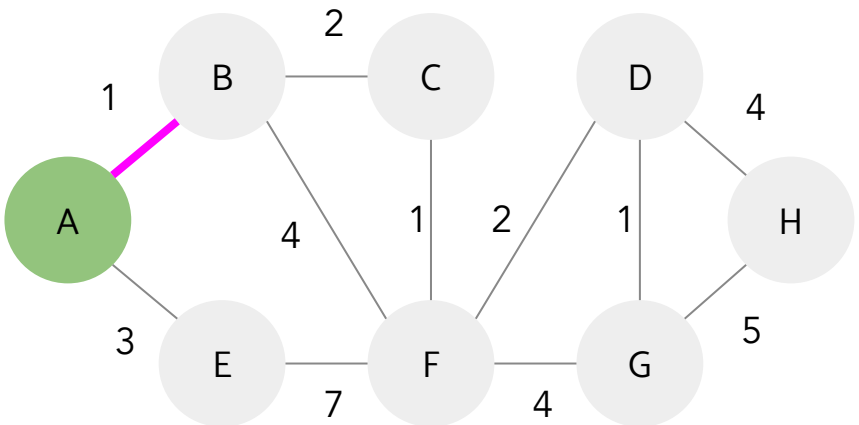
1A Dijkstra's, A*

	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	∞	∞	∞	∞	∞	∞	∞



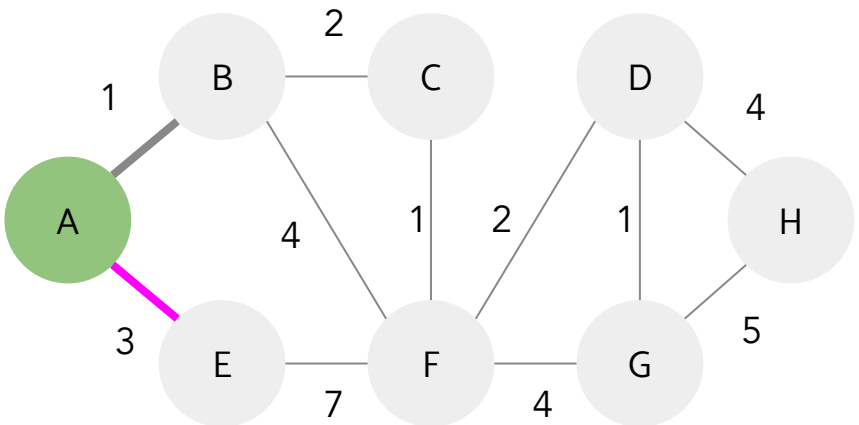
1A Dijkstra's, A*

	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	∞	∞	∞	∞

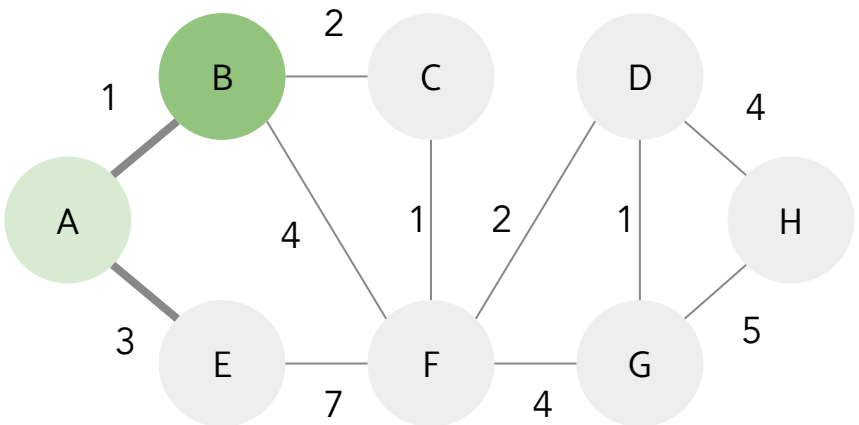


1A Dijkstra's, A*

	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞

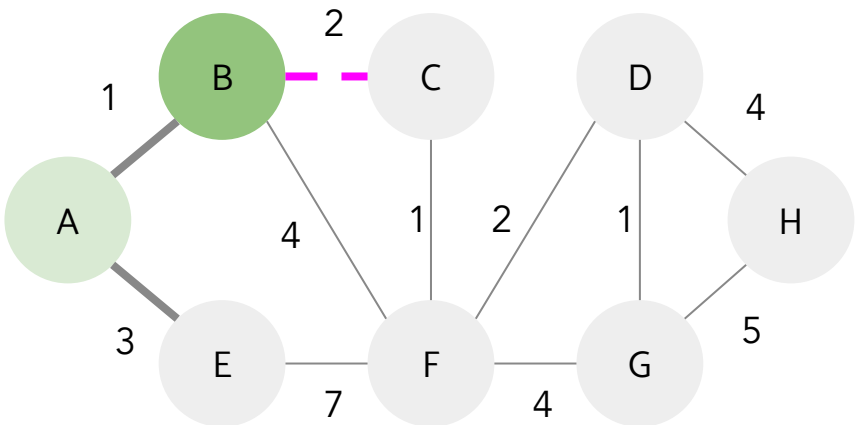


1A Dijkstra's, A*



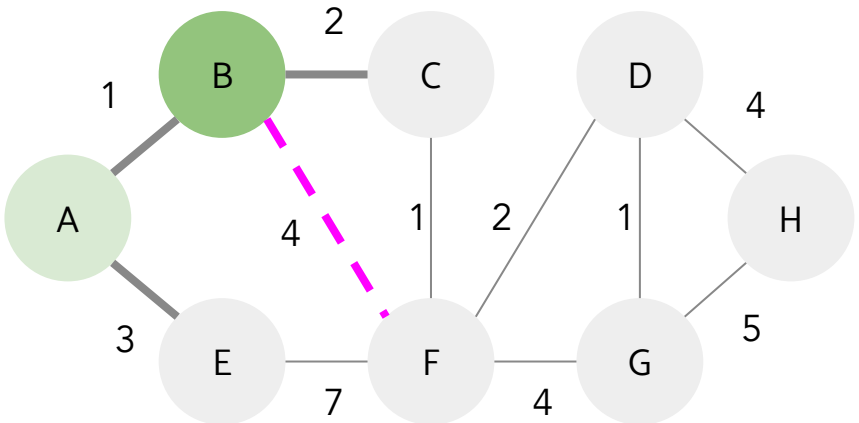
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	∞	∞	3	∞	∞	∞

1A Dijkstra's, A*



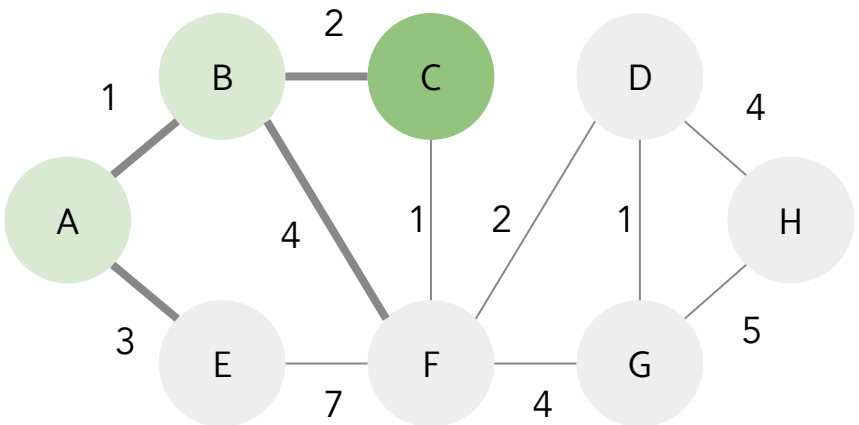
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	∞	∞	∞

1A Dijkstra's, A*



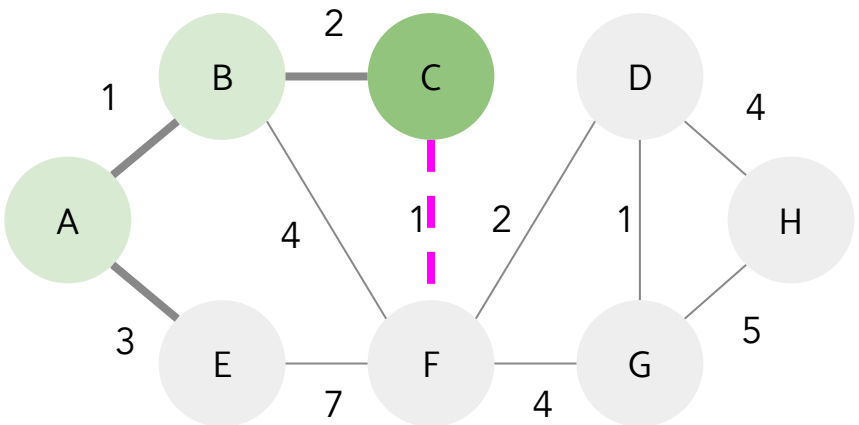
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞

1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	5	∞	∞

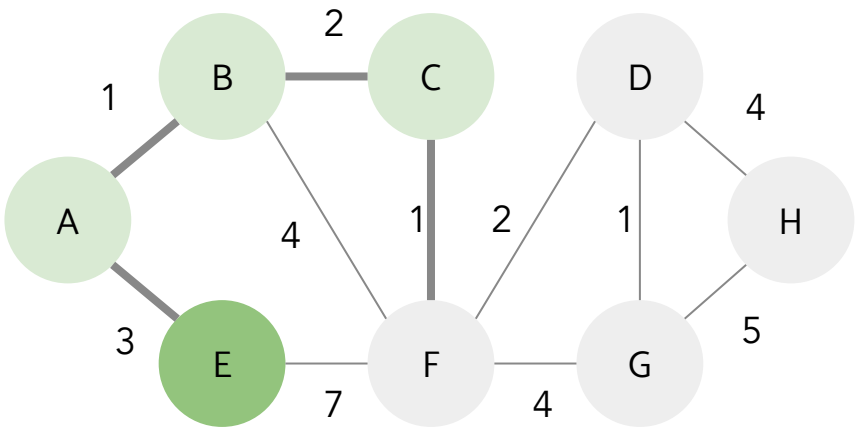
1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞

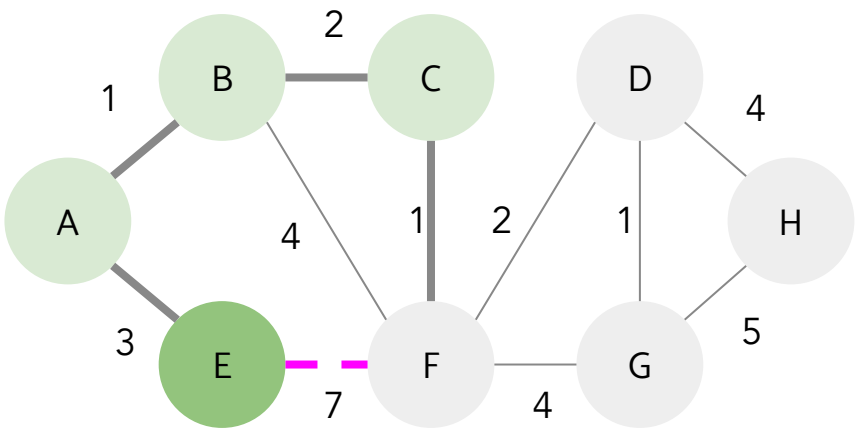
We found a better path to F, so we update distTo[F]

1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞

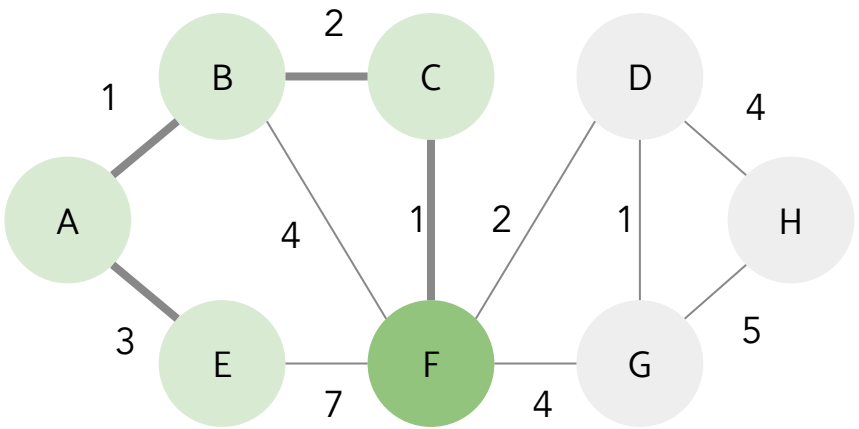
1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞

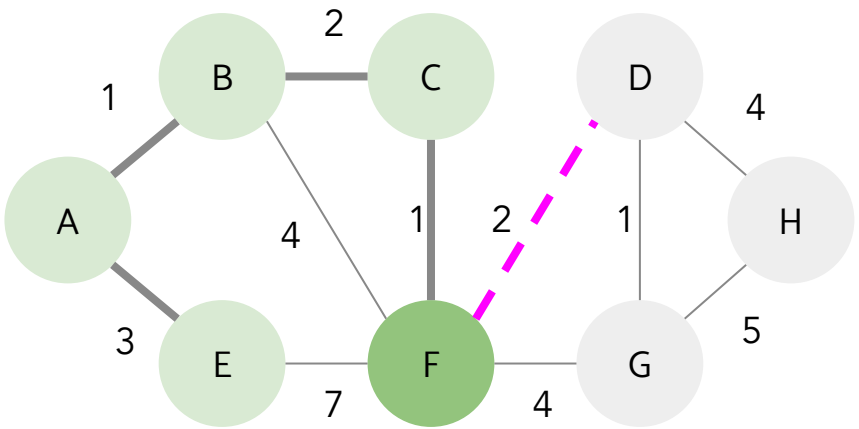
This new path to F is NOT better than our current one

1A Dijkstra's, A*



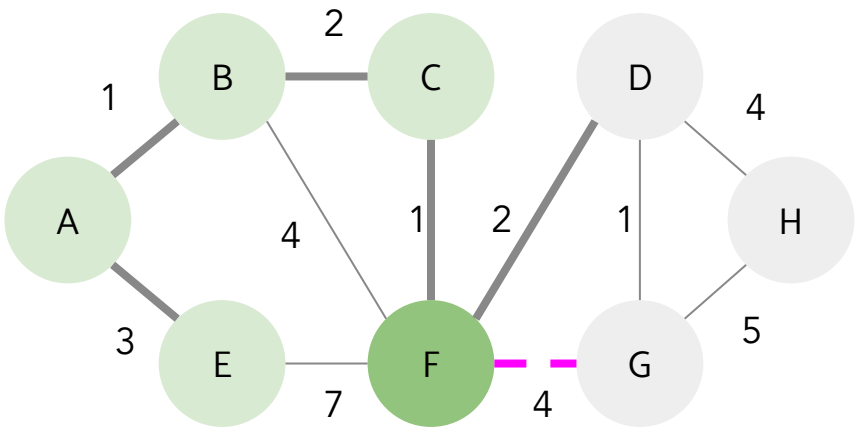
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				∞		✓	∞	∞

1A Dijkstra's, A*



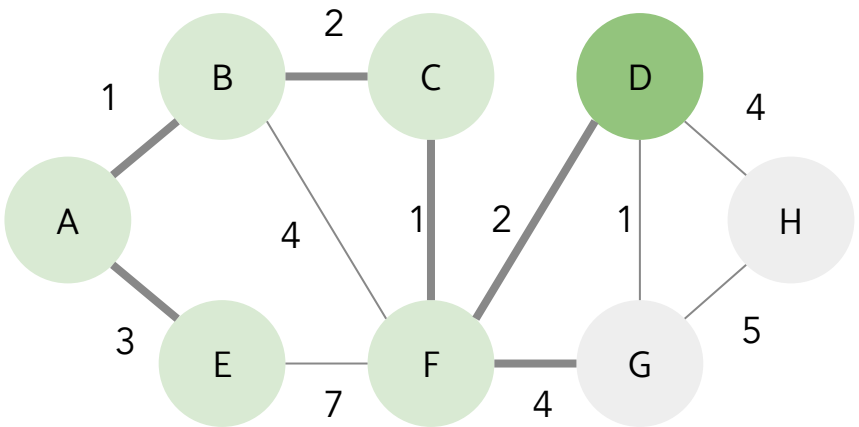
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				6		✓	∞	∞

1A Dijkstra's, A*



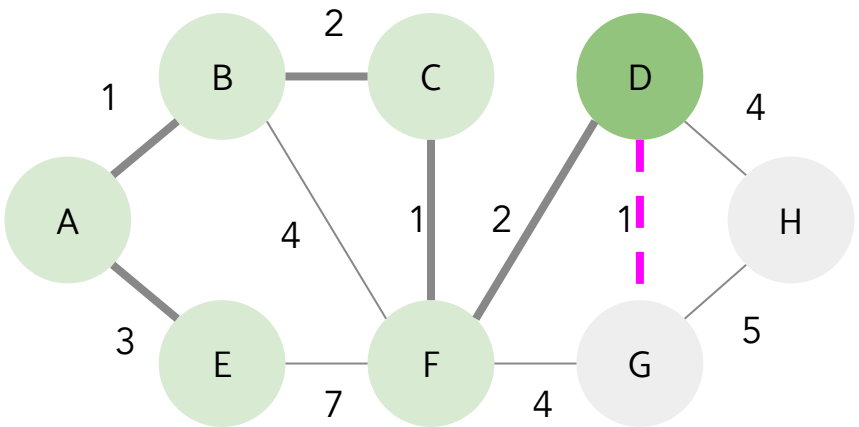
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				6		✓	8	∞

1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				6		✓	8	∞
6				✓			8	∞

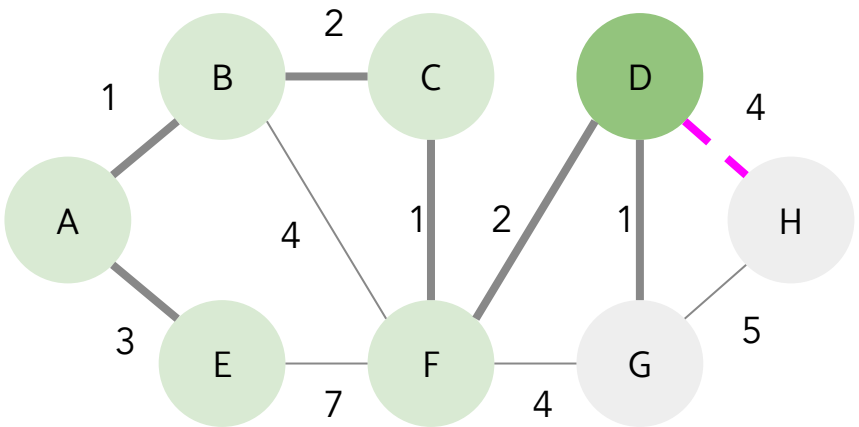
1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				6		✓	8	∞
6				✓			7	∞

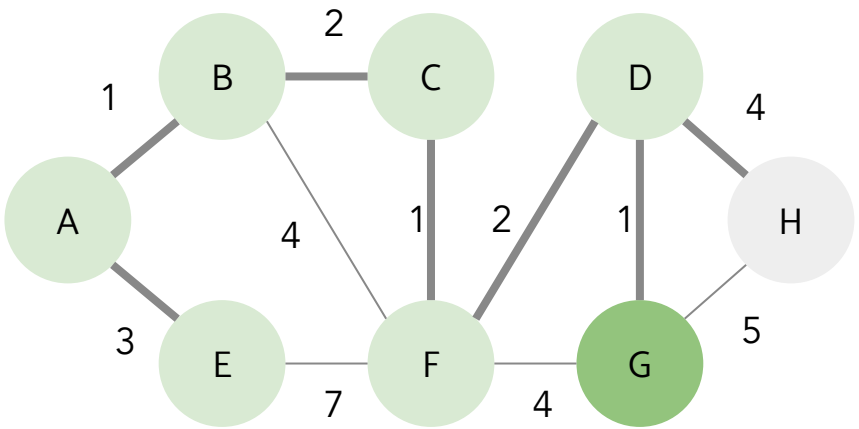
We found a better path to G, so we update distTo[G]

1A Dijkstra's, A*



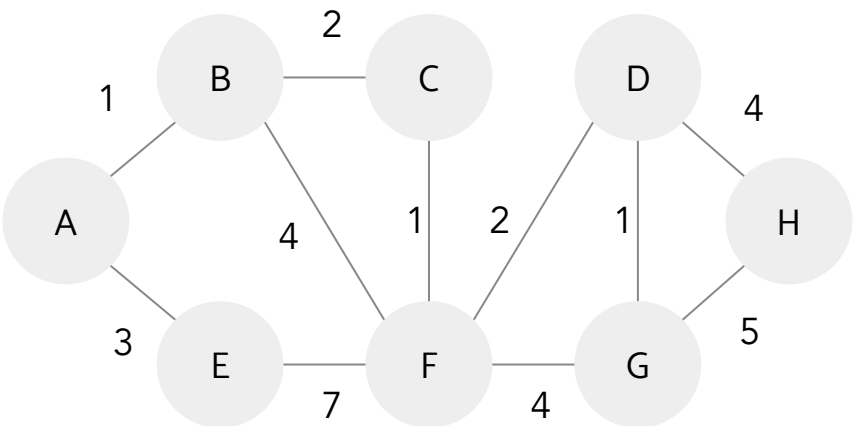
	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				6		✓	8	∞
6				✓			7	10

1A Dijkstra's, A*



	A	B	C	D	E	F	G	H
DistTo	0	∞	∞	∞	∞	∞	∞	∞
1	✓	1	∞	∞	3	∞	∞	∞
2		✓	3	∞	3	5	∞	∞
3			✓	∞	3	4	∞	∞
4				∞	✓	4	∞	∞
5				6		✓	8	∞
6				✓			7	10
7							✓	

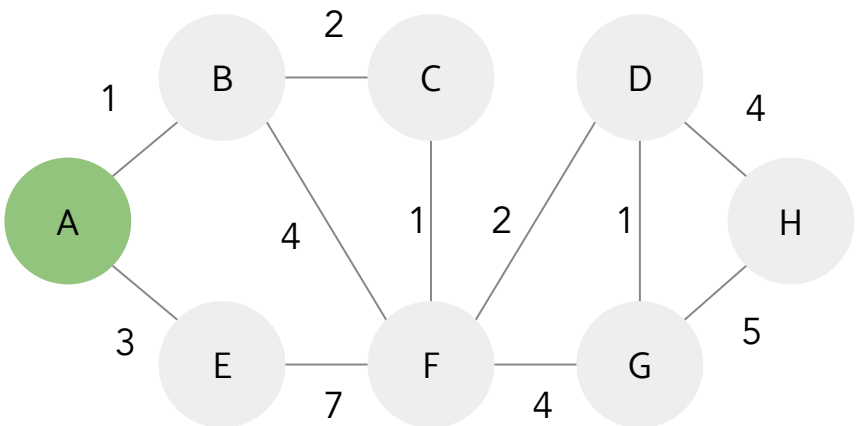
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞

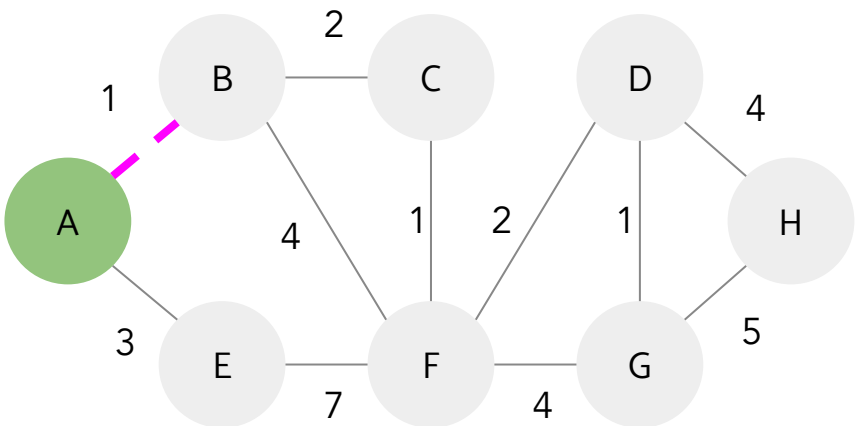
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	∞	∞	∞	∞	∞	∞	∞

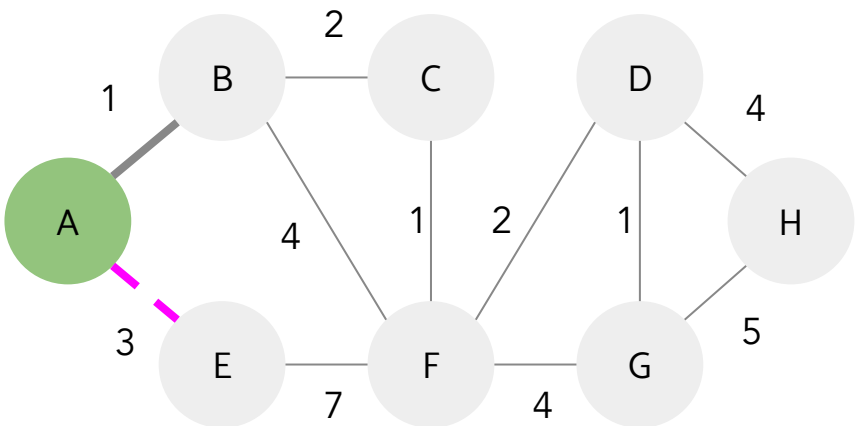
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	∞	∞	∞	∞

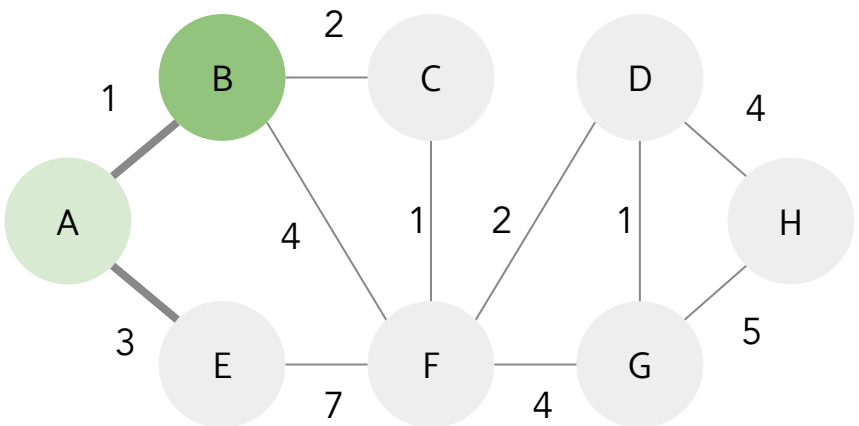
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞

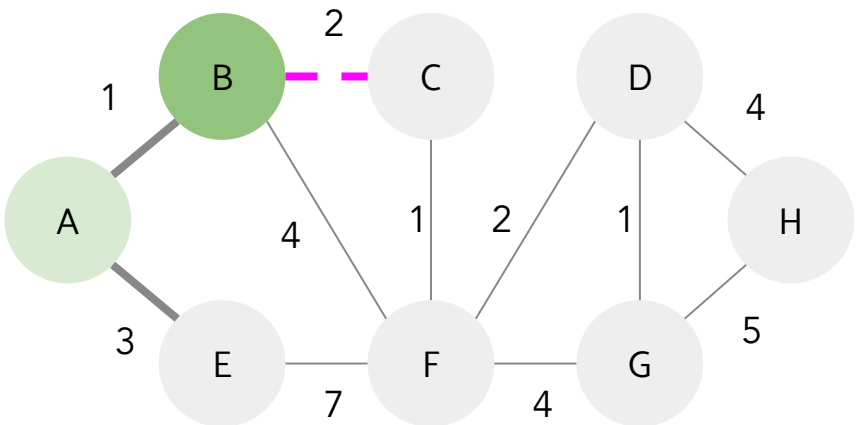
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	∞	∞	3,13	∞	∞	∞

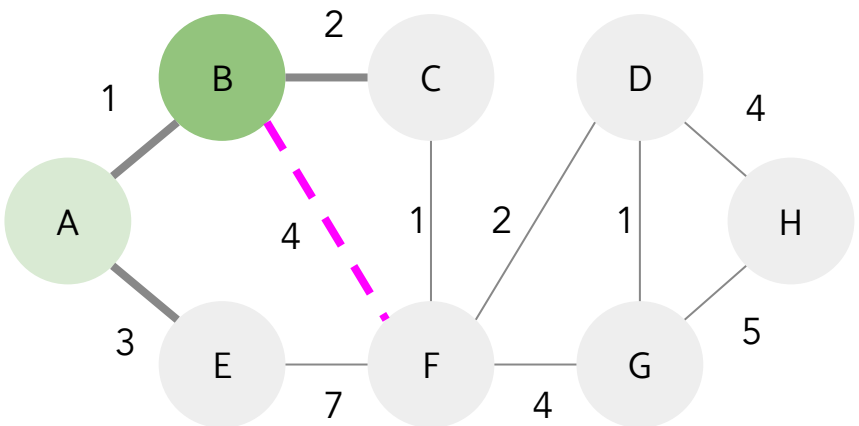
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	∞	∞	∞

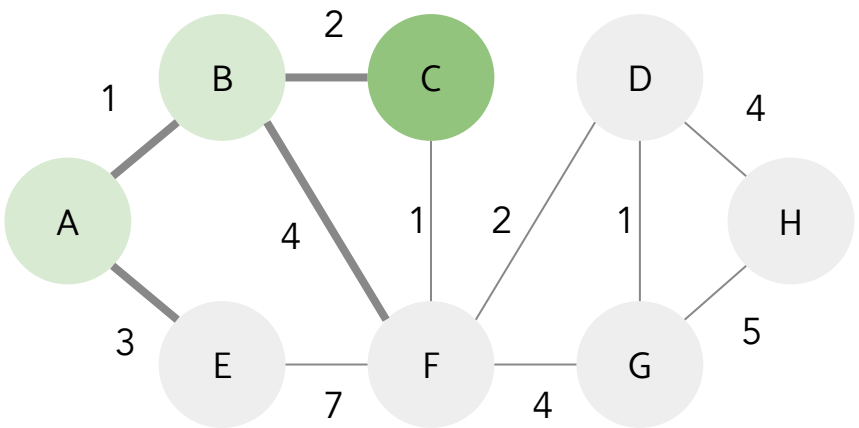
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞

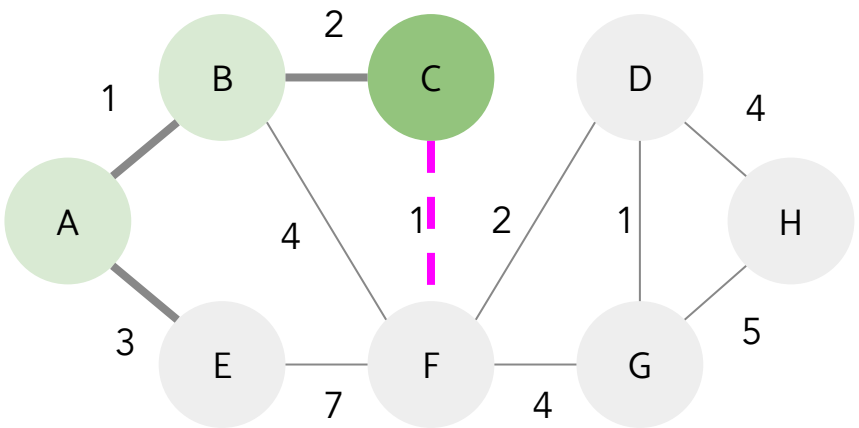
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	5,8	∞	∞

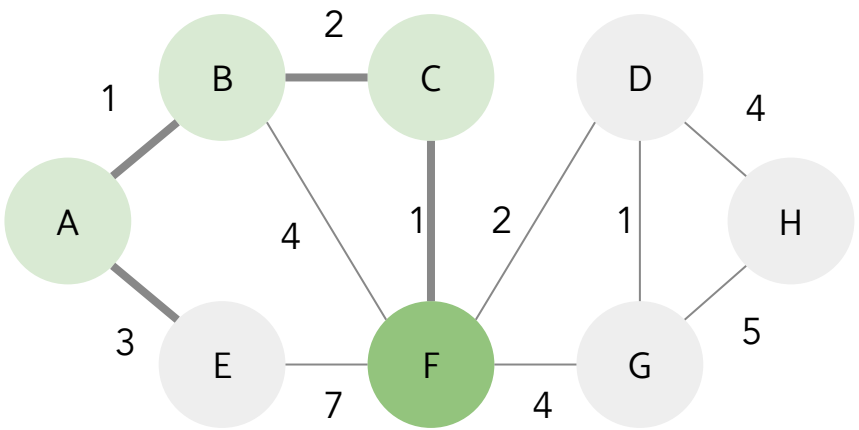
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞

1B Dijkstra's, A*

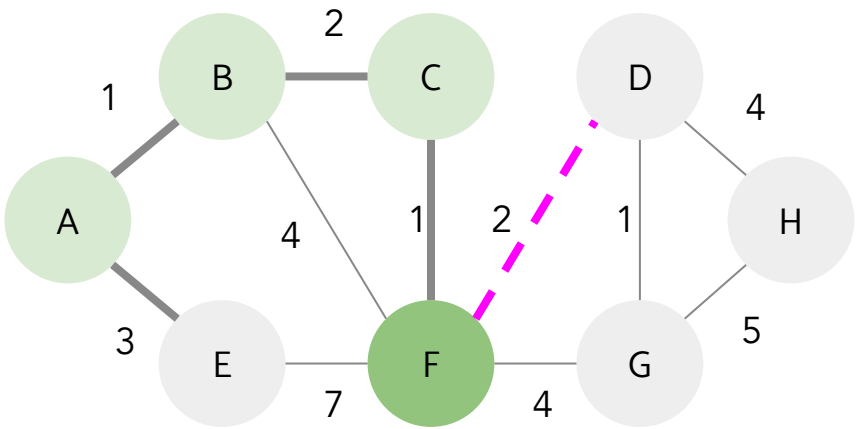


u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				∞	3,13	✓	∞	∞

Even though E has a closer distance, it has lower priority than F because it has a high heuristic!

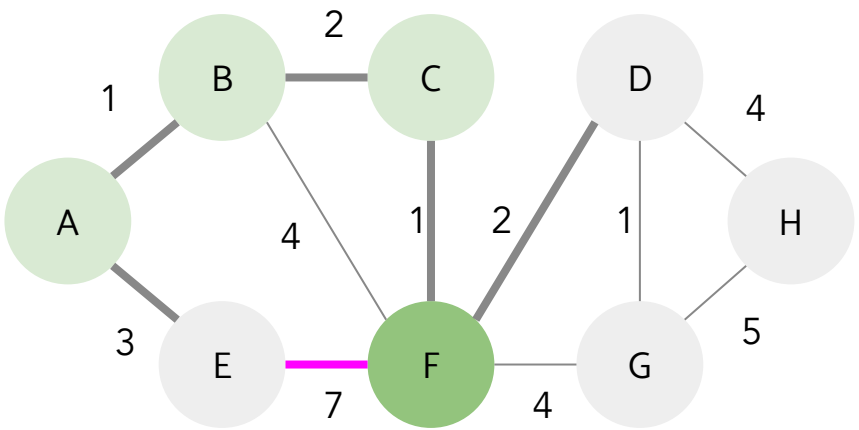
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	∞	∞

1B Dijkstra's, A*

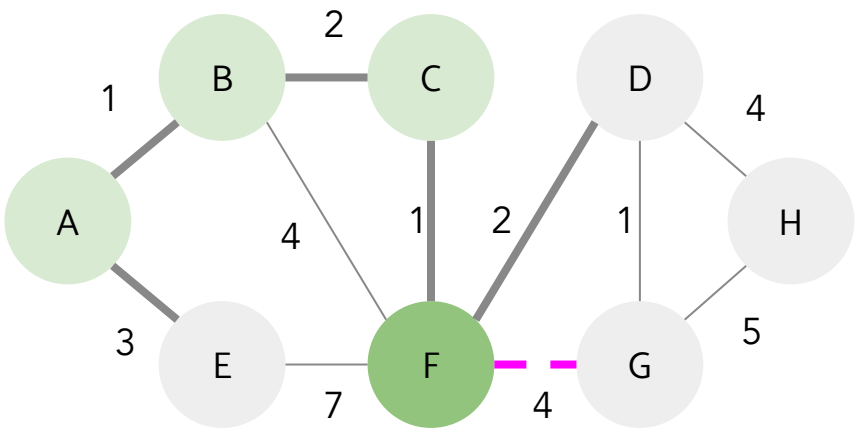


u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	∞	∞

The path to E from F is not better than our current path

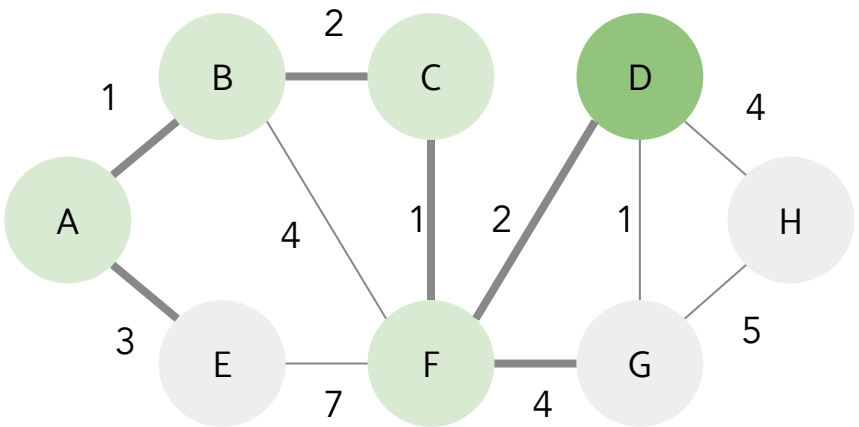
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	8,8	∞

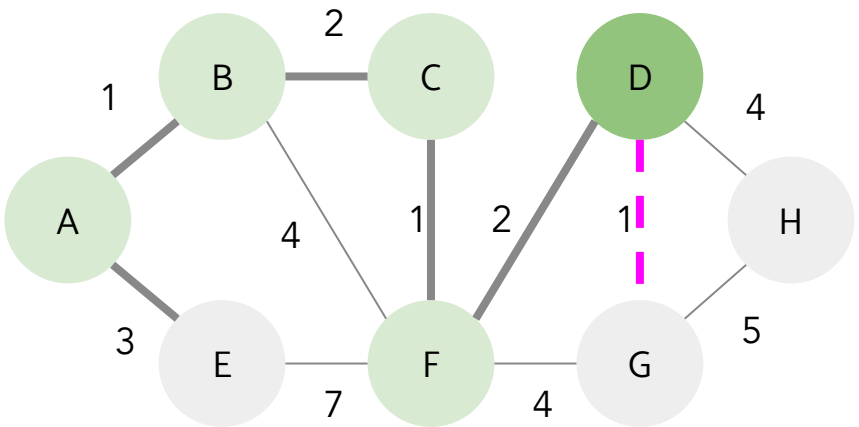
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	8,8	∞
5				✓	3,13		8,8	∞

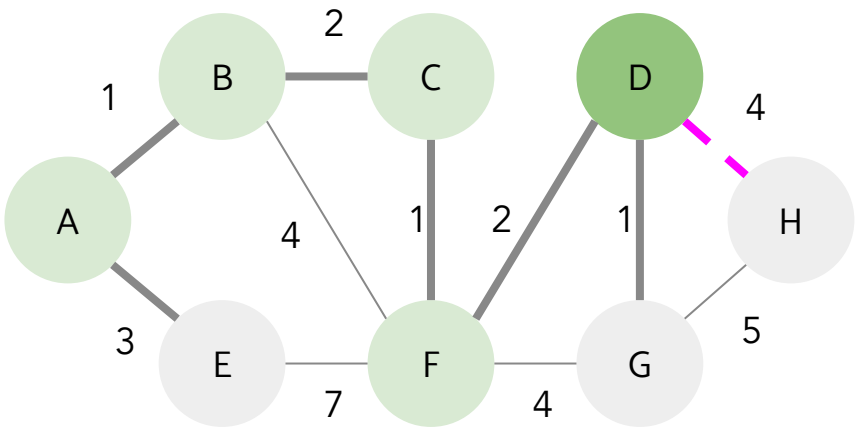
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	8,8	∞
5				✓	3,13		7,7	∞

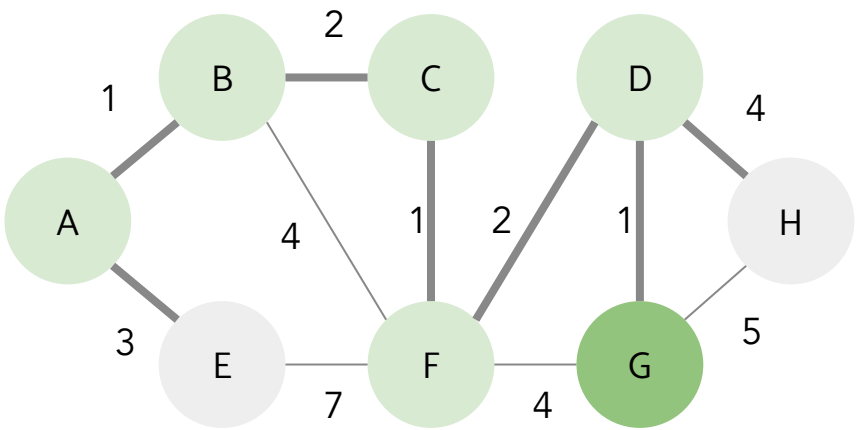
1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	8,8	∞
5				✓	3,13		7,7	10,15

1B Dijkstra's, A*



u	A	B	C	D	E	F	G	H
h(u,G)	9	7	4	1	10	3	0	5

	A	B	C	D	E	F	G	H
Start	0,9	∞	∞	∞	∞	∞	∞	∞
1	✓	1,8	∞	∞	3,13	∞	∞	∞
2		✓	3,7	∞	3,13	5,8	∞	∞
3			✓	∞	3,13	4,7	∞	∞
4				6,7	3,13	✓	8,8	∞
5				✓	3,13		7,7	10,15
6							✓	

2A Conceptual Shortest Paths

For a **weighted, undirected** graph:

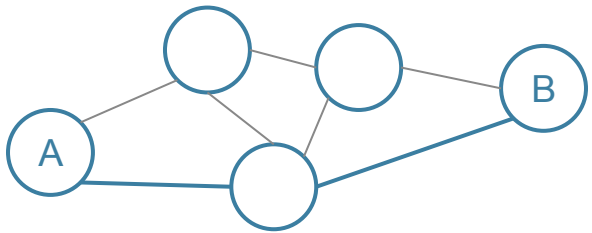
(T/F) If all edge weights are equal and positive, the breadth-first search starting from node A will return the shortest path from a node A to a target node B.

2A Conceptual Shortest Paths

For a **weighted, undirected** graph:

(T/F) If all edge weights are equal and positive, the breadth-first search starting from node A will return the shortest path from a node A to a target node B.

True. If all edges are equal in weight, then the shortest path from A to each node is proportional to the number of edges on the path, so breadth first search will return the shortest path.



2B Conceptual Shortest Paths

For a **weighted, undirected** graph:

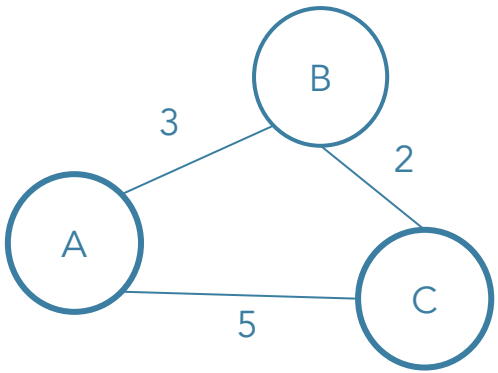
(T/F) If all edges have distinct weights, the shortest path between any two vertices is unique.

2B Conceptual Shortest Paths

For a **weighted, undirected** graph:

(T/F) If all edges have distinct weights, the shortest path between any two vertices is unique.

False. Consider a case of 3 nodes where AB is 3, AC is 5, and BC is 2. Here, the two possible paths from A to C both are of length 5.



2C Conceptual Shortest Paths

For a **weighted, undirected** graph:

(T/F) Adding a constant positive integer k to all edge weights will not change the original shortest path between any two vertices.

2C Conceptual Shortest Paths

For a **weighted, undirected** graph:

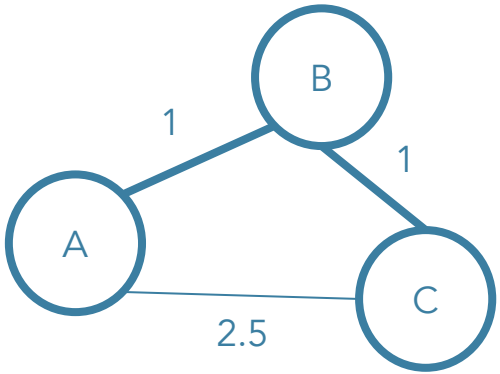
(T/F) Adding a constant positive integer k to all edge weights will not change the original shortest path between any two vertices.

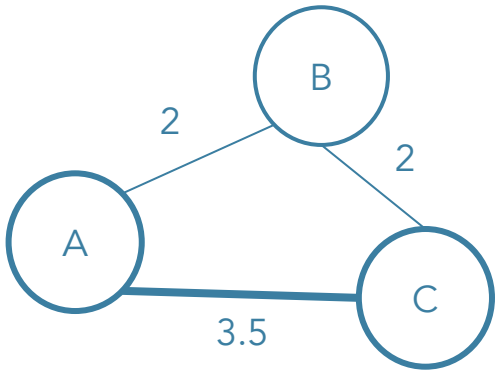
False. Consider a case of 3 nodes A, B, and C where AB is 1, AC is 2.5 and BC is 1. Clearly, the best path from A to C is through B, with weight 2. However, if we add 1 to each edge weight, suddenly the path going through B will have weight 4, while the direct path is only 3.5. In general, paths with greater number of edges end up getting penalized more than paths with fewer edges.

2C Conceptual Shortest Paths

For a **weighted, undirected** graph:

(T/F) Adding a constant positive integer k to all edge weights will not change the original shortest path between any two vertices.





2D Conceptual Shortest Paths

For a **weighted, undirected** graph:

(T/F) Multiplying all edge weights by a constant positive integer k will not change the original shortest path between any two vertices.

2D Conceptual Shortest Paths

For a **weighted, undirected** graph:

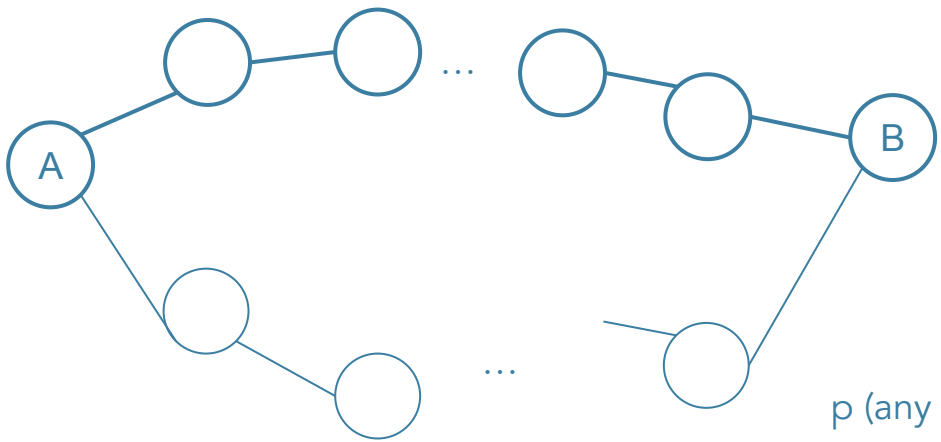
(T/F) Multiplying all edge weights by a constant positive integer k will not change the original shortest path between any two vertices.

True. Suppose we have arbitrary nodes u and v. Let's say the shortest path from u to v, before the multiplication by k, was of total weight w. This implies that every other path from u to v was of total weight greater than w. After multiplying each edge weight by k, the total weight of the shortest path becomes $w * k$ and the total weight of every other path becomes some number greater than $w * k$. Therefore, the original shortest path doesn't change.

2D Conceptual Shortest Paths

For a **weighted, undirected** graph:

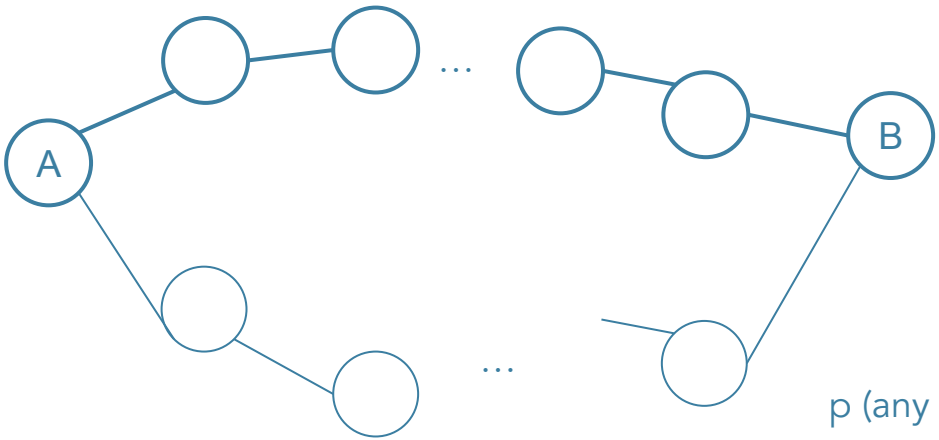
(T/F) Multiplying all edge weights by a constant positive integer k will not change the original shortest path between any two vertices.



2D Conceptual Shortest Paths

For a **weighted, undirected** graph:

(T/F) Multiplying all edge weights by a constant positive integer k will not change the original shortest path between any two vertices.



3A Shortest Paths Algorithm Design

Given a weighted directed graph with V vertices and E edges, two disjoint sets of vertices M and F, determine the shortest path between any vertex in M and any vertex in F.

Case 1: $|M| = 1$, $|F| > 1$.

3A Shortest Paths Algorithm Design

Given a weighted directed graph with V vertices and E edges, two disjoint sets of vertices M and F, determine the shortest path between any vertex in M and any vertex in F.

Case 1: |M| = 1, |F| > 1.

Run Dijkstra's, starting from the only city m in M. This will generate the shortest distance to all cities in the graph. Then, iterate over every city f in F, and take the minimum distance from m to f.

3B Shortest Paths Algorithm Design

Given a weighted directed graph with V vertices and E edges, two disjoint sets of vertices M and F, determine the shortest path between any vertex in M and any vertex in F.

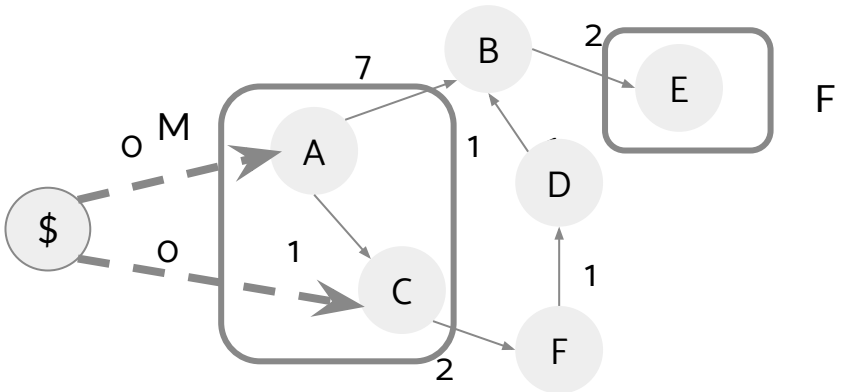
Case 2: |M| > 1, |F| = 1.

3B Shortest Paths Algorithm Design

Given a weighted directed graph with V vertices and E edges, two disjoint sets of vertices M and F, determine the shortest path between any vertex in M and any vertex in F.

Case 2: |M| > 1, |F| = 1.

Create a dummy vertex and add an edge with weight 0 from dummy to all vertices in M. Run Dijkstra's starting from the dummy. Take the distance from dummy to the node in F.



3C Shortest Paths Algorithm Design

Given a weighted directed graph with V vertices and E edges, two disjoint sets of vertices M and F, determine the shortest path between any vertex in M and any vertex in F.

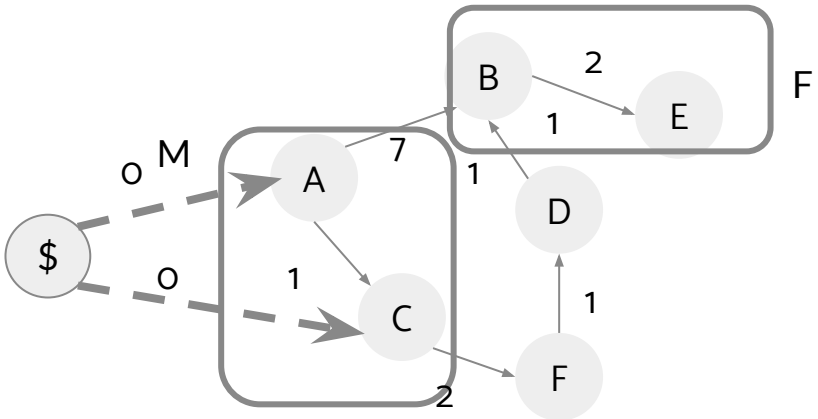
Case 3: |M| > 1, |F| > 1.

3C Shortest Paths Algorithm Design

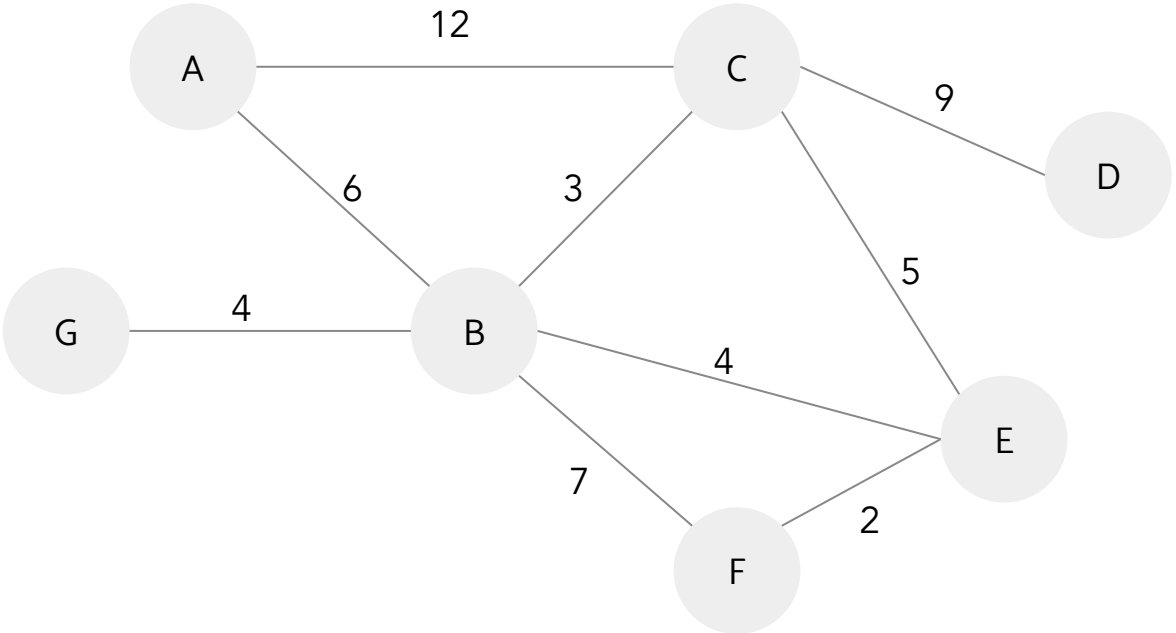
Given a weighted directed graph with V vertices and E edges, two disjoint sets of vertices M and F, determine the shortest path between any vertex in M and any vertex in F.

Case 3: |M| > 1, |F| > 1.

Same algorithm in 3C, but take the minimum distance from dummy to all nodes in F.

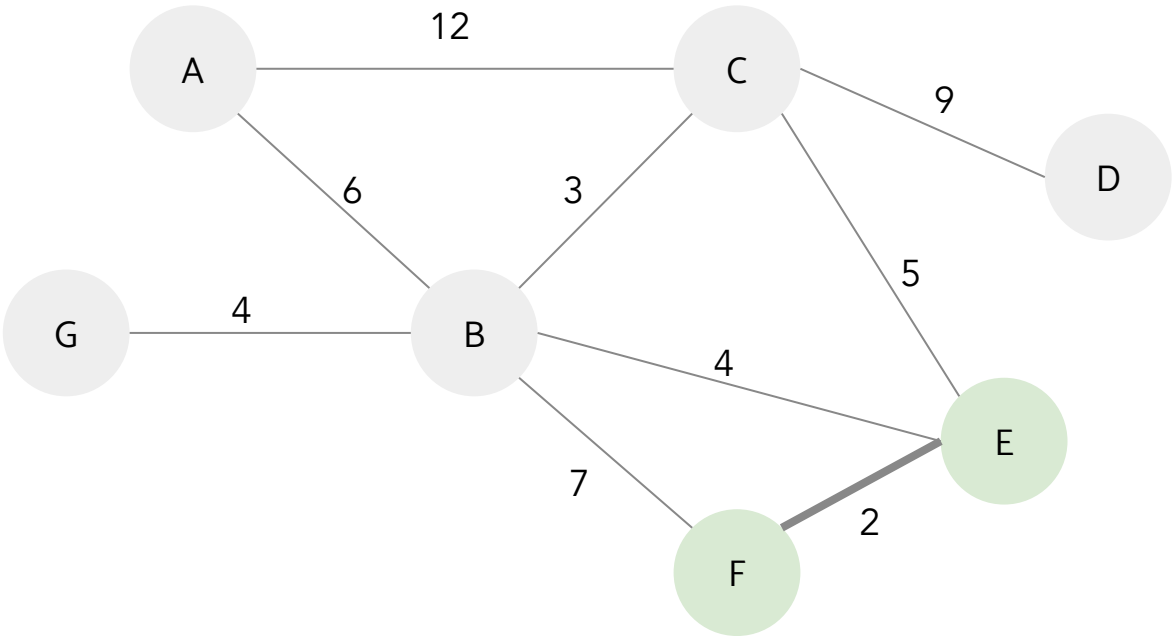


4A Introduction to MSTs - Kruskal's



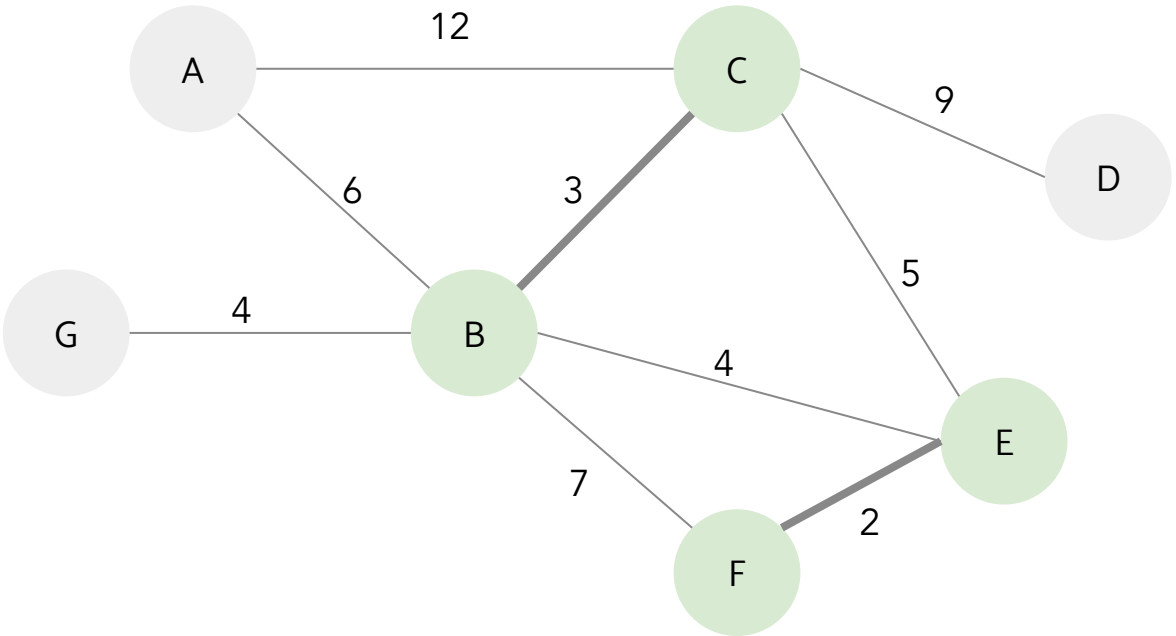
Connected Nodes:

4A Introduction to MSTs - Kruskal's



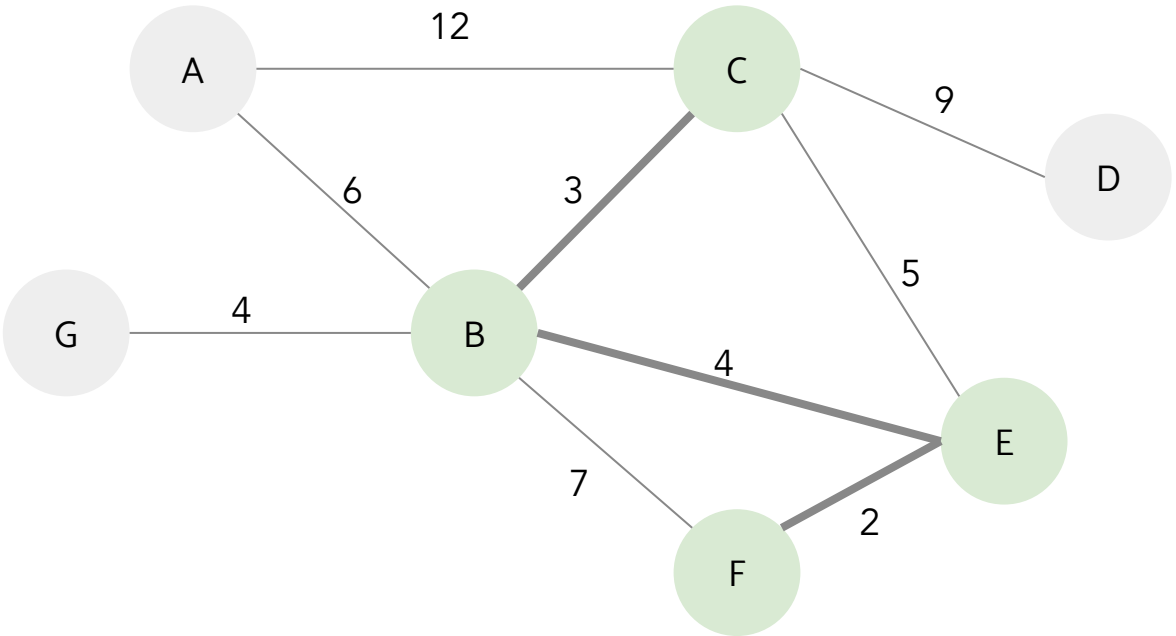
Connected Nodes:
F
E

4A Introduction to MSTs - Kruskal's



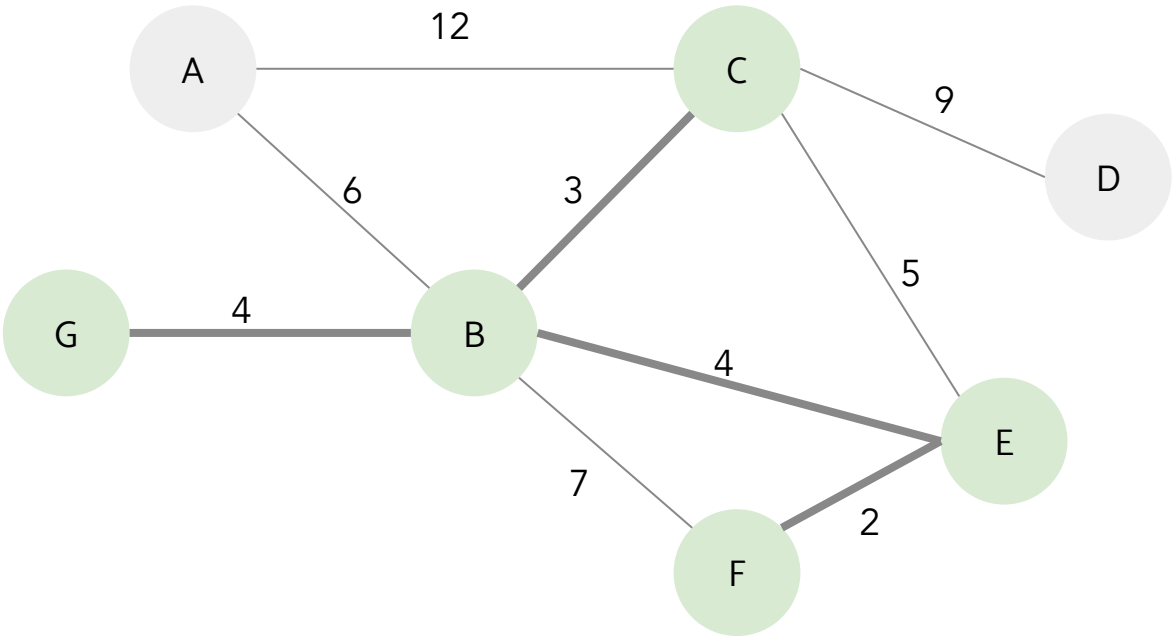
Connected Nodes:
F
E
B
C

4A Introduction to MSTs - Kruskal's



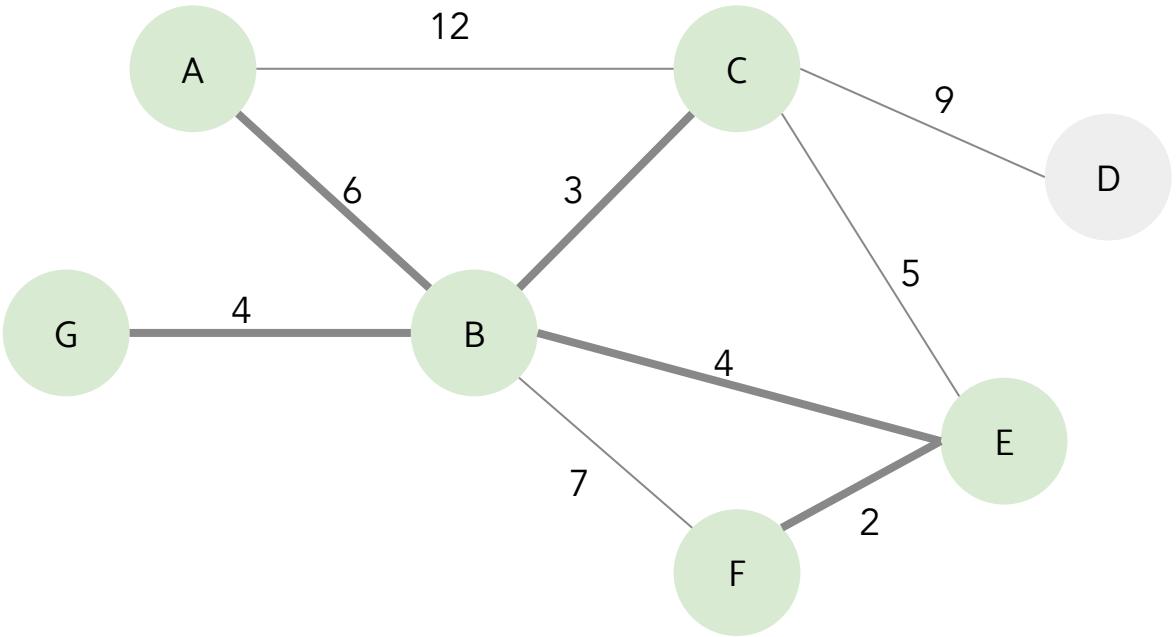
Connected Nodes:
F
E
B
C

4A Introduction to MSTs - Kruskal's



Connected Nodes:
F
E
B
C
G

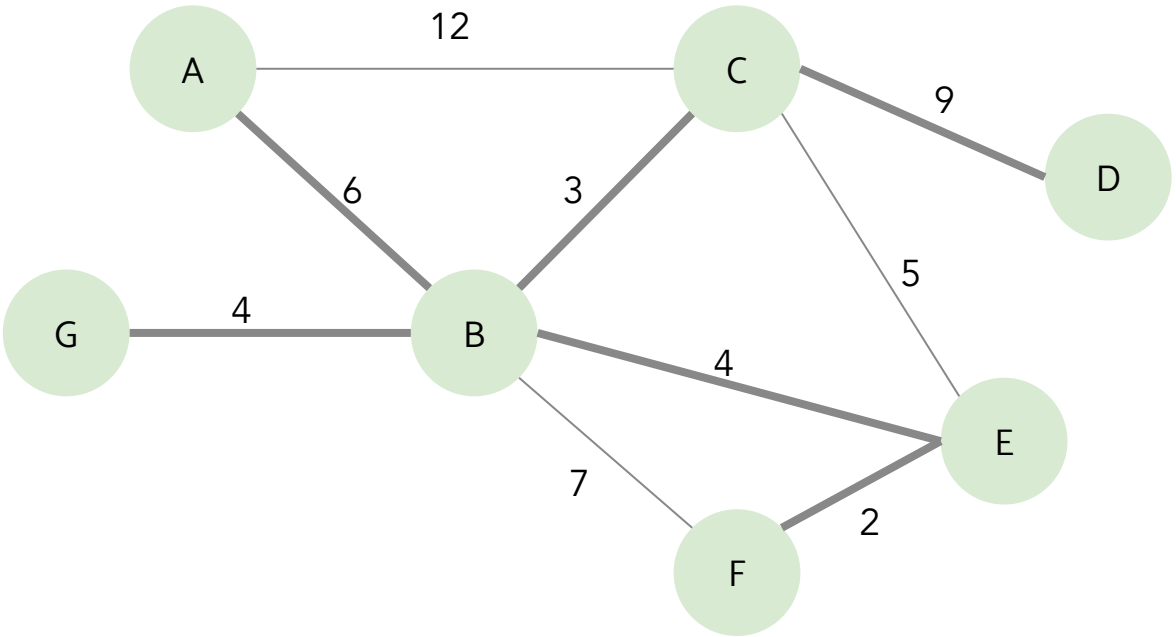
4A Introduction to MSTs - Kruskal's



Connected Nodes:
F
E
B
C
G
A

We don't take edge CE; it creates a cycle!

4A Introduction to MSTs - Kruskal's

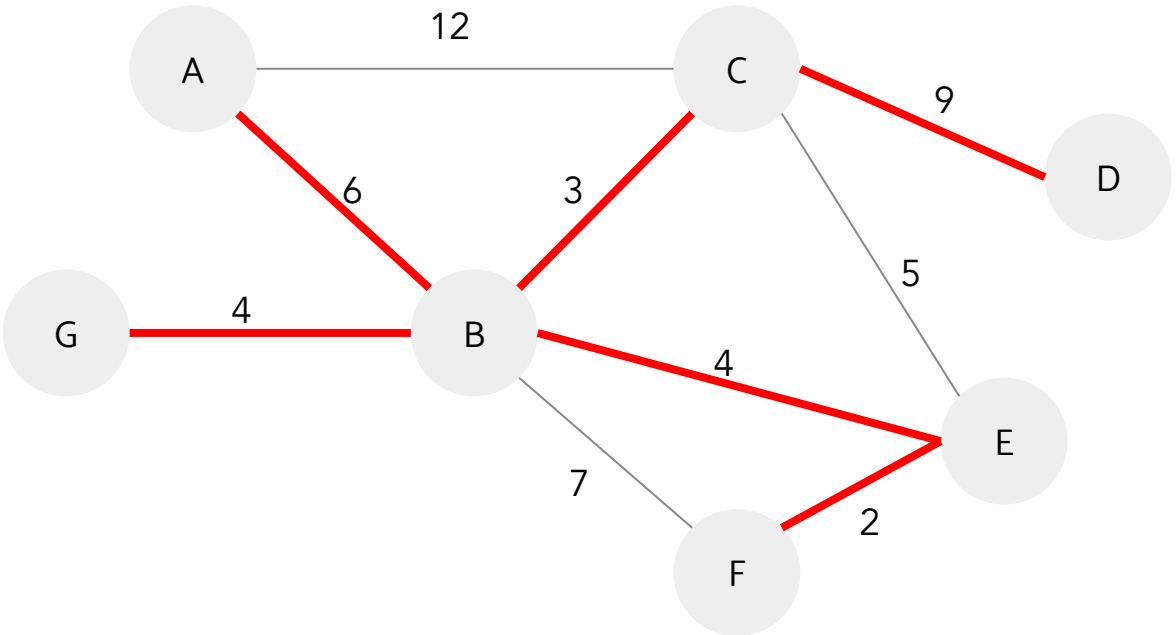


Connected Nodes:
F
E
B
C
G
A
D

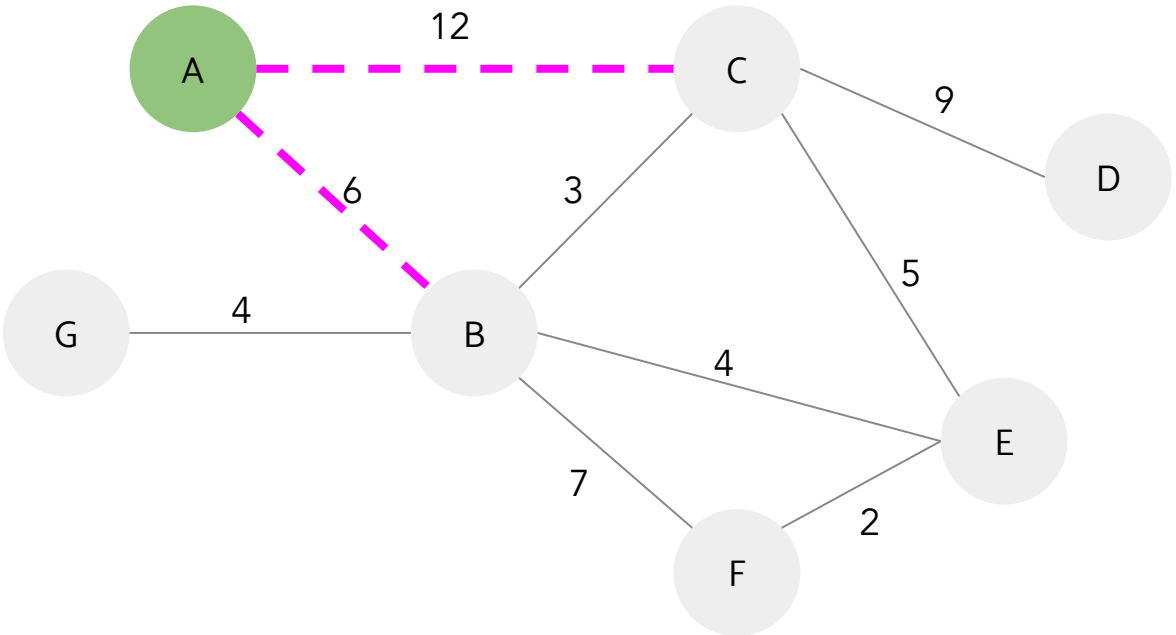
We don't take edge BF; it creates a cycle!

4A Introduction to MSTs - Kruskal's

Final Result

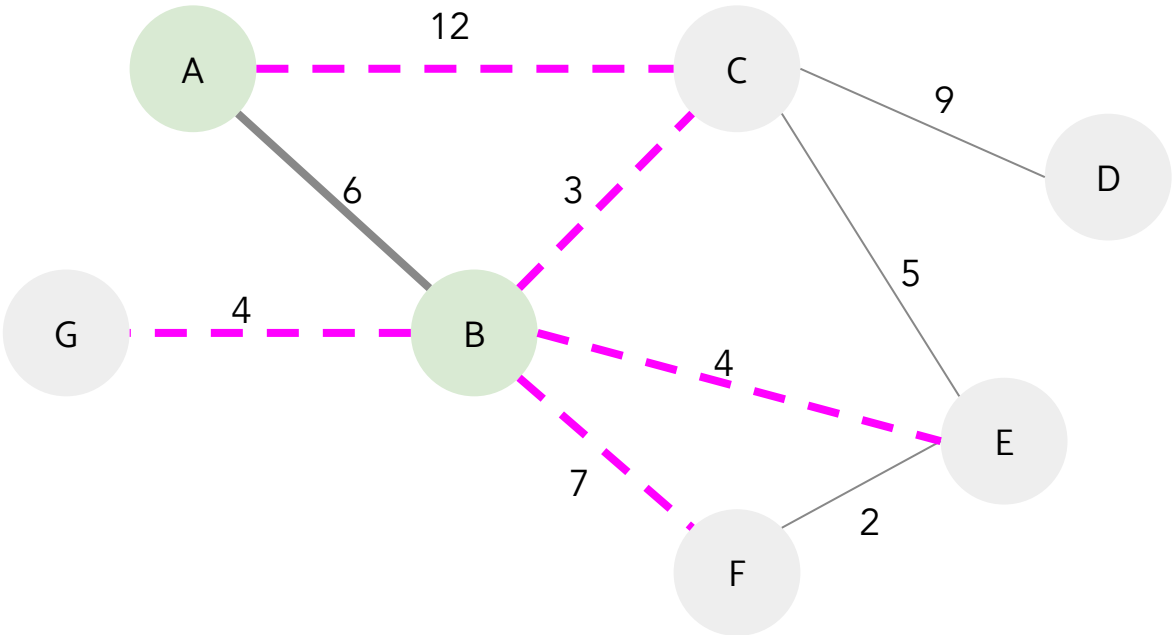


4A Introduction to MSTs - Prim's



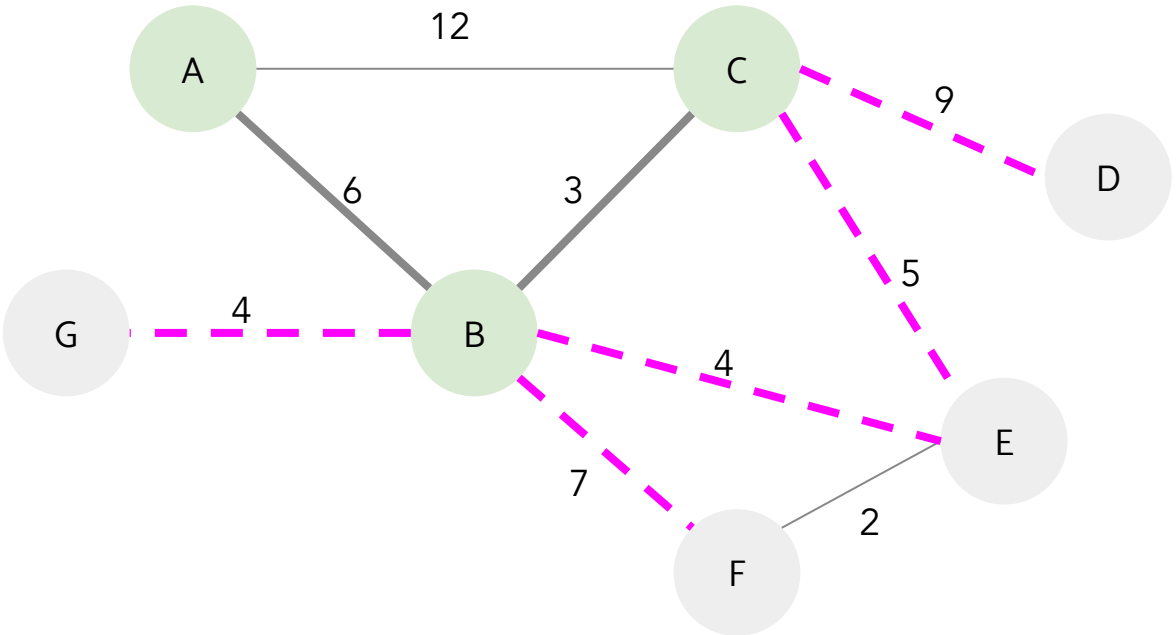
Connected Nodes:
A

4A Introduction to MSTs - Prim's



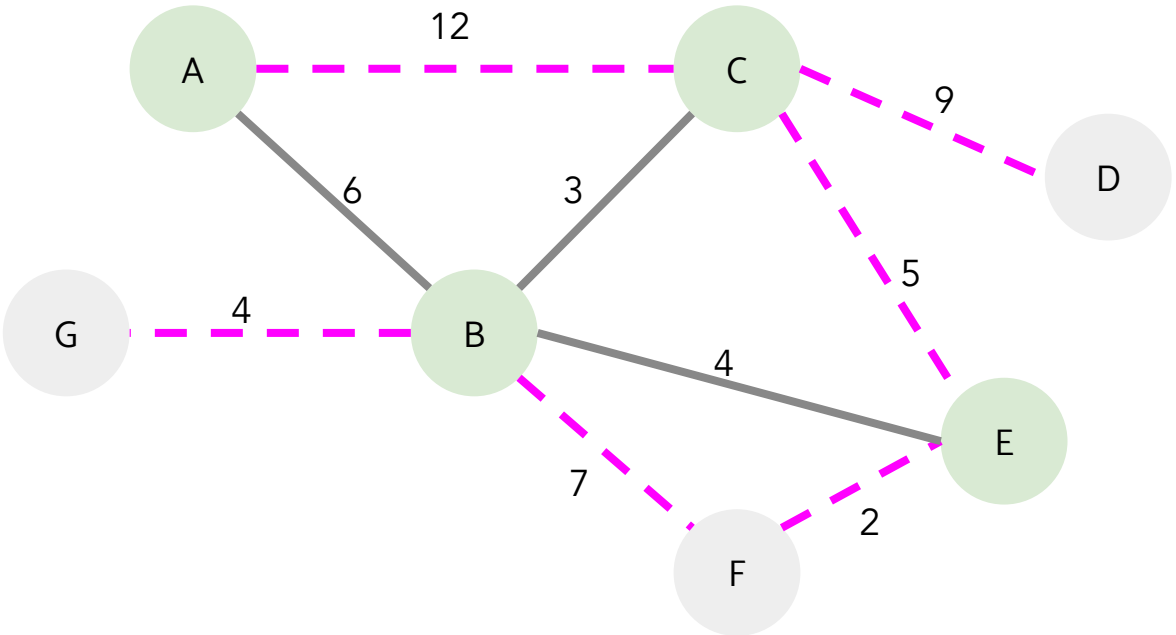
Connected Nodes:
A
B

4A Introduction to MSTs - Prim's



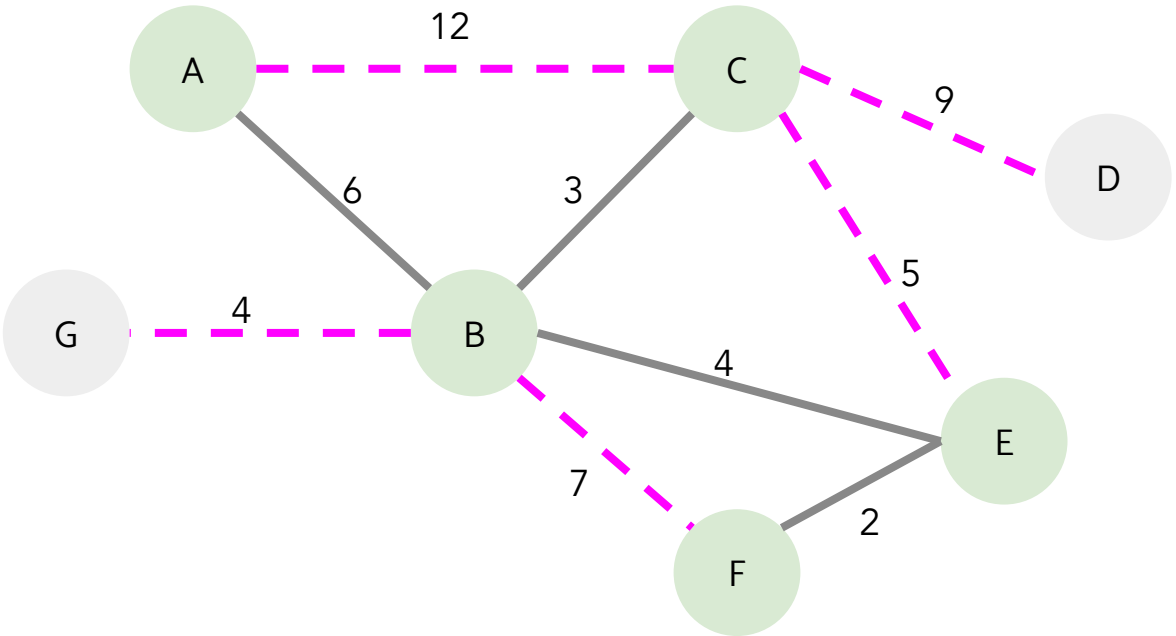
Connected Nodes:
A
B
C

4A Introduction to MSTs - Prim's



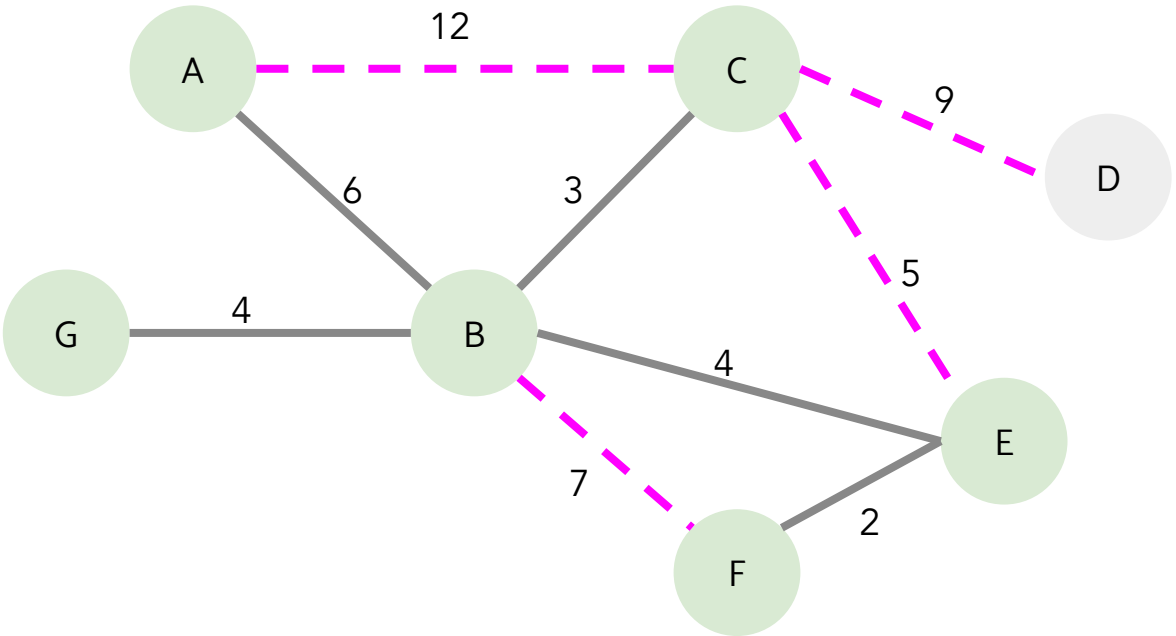
Connected Nodes:
A
B
C
E

4A Introduction to MSTs - Prim's



Connected Nodes:
A
B
C
E
F

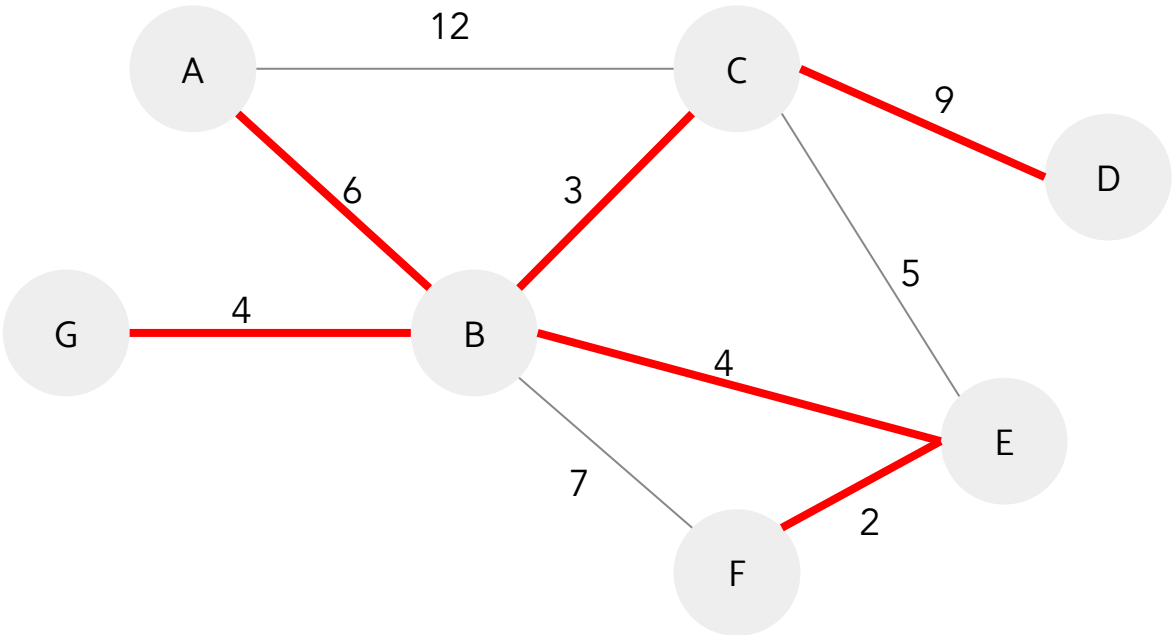
4A Introduction to MSTs - Prim's



Connected Nodes:
A
B
C
E
F
G

4A Introduction to MSTs - Prim's

Final Result



4B Introduction to MSTs

True/False: Adding 1 to the smallest edge of a graph G with unique edge weights must change the total weight of its MST

4B Introduction to MSTs

True/False: Adding 1 to the smallest edge of a graph G with unique edge weights must change the total weight of its MST

True, either this smallest edge (now with weight +1) is included, or this smallest edge is not included and some larger edge takes its place since there was no other edge of equal weight. Either way, the total weight increases.

4C Introduction to MSTs

True/False: If all the weights in a graph is unique, there is only one possible MST.

4C Introduction to MSTs

True/False: If all the weights in an MST are unique, there is only one possible MST.

True. The cut property states that the minimum weight edge in a cut must be in the MST. Since all weights are unique, the minimum weight edge is always unique, so there is only one possible MST.

4D Introduction to MSTs

True/False: The shortest path from vertex *u* to vertex *v* in a graph G is the same as the shortest path from *u* to *v* using only edges in T, where T is the MST of G.

4D Introduction to MSTs

True/False: The shortest path from vertex *u* to vertex *v* in a graph G is the same as the shortest path from *u* to *v* using only edges in T, where T is the MST of G.

False, consider vertices C and E in the graph from 4A