CSC 172– Data Structures and Algorithms

Lecture #23

Spring 2018

Please put away all electronic devices

Announcements

- Project 4 is out
  - You may work with a partner
  - (absolute) due date: 05/03/2018

GRAPH ADT

The Graph ADT

The Graph ADT describes a container storing an adjacency relation

- Queries include:
  - The number of vertices
  - The number of edges
  - List the vertices adjacent to a given vertex
  - Are two vertices adjacent?
  - Are two vertices connected?

- Modifications include:
  - Inserting or removing an edge
  - Inserting or removing a vertex (and all edges containing that vertex)

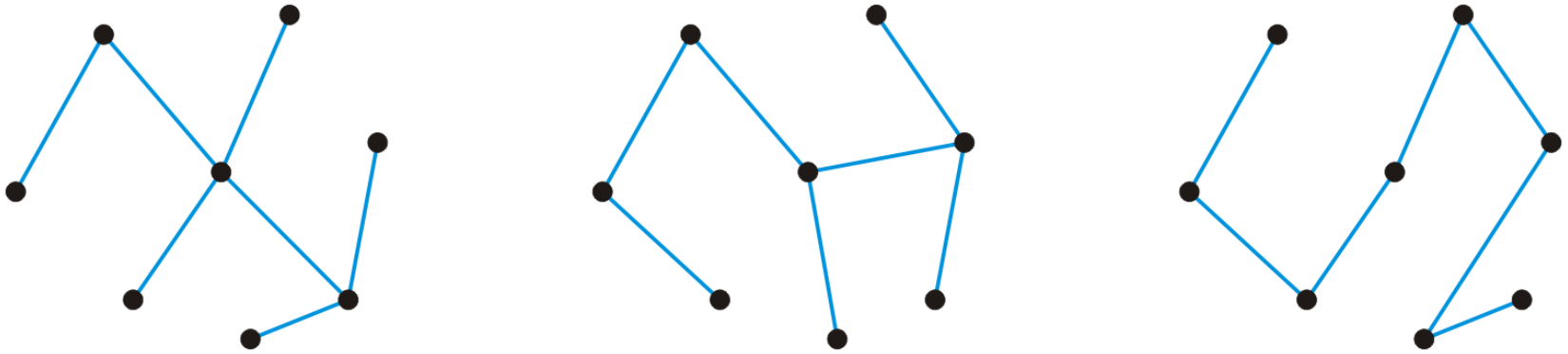
The run-time of these operations will depend on the representation

BACK TO GRAPH BASICS

Trees

A graph is a tree if it is connected and there is a unique path between any two vertices

- Three trees on the same eight vertices



- The graph is *acyclic*, that is, it does not contain any cycles
- Adding one more edge must create a cycle
- Removing any one edge creates two disjoint non-empty sub-graphs

Trees

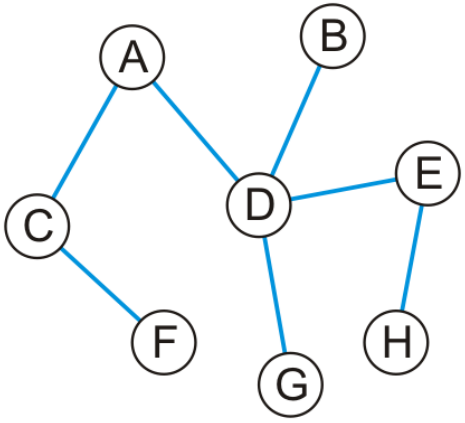
Any tree can be converted into a rooted tree by:

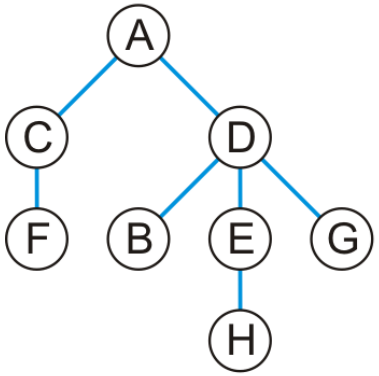
- Choosing any vertex to be the root
- Defining its neighboring vertices as its children

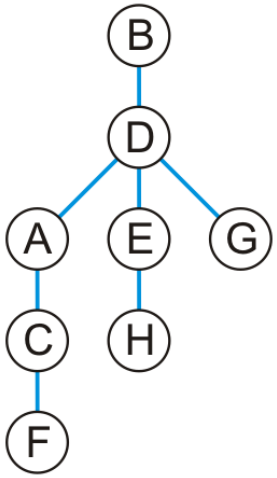
and then recursively defining:

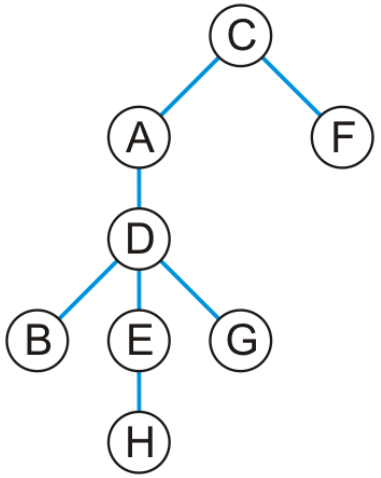
- All neighboring vertices other than that one designated its parent are now defined to be that vertices children

Given this tree, here are three rooted trees (out of many others) associated with it









Forests

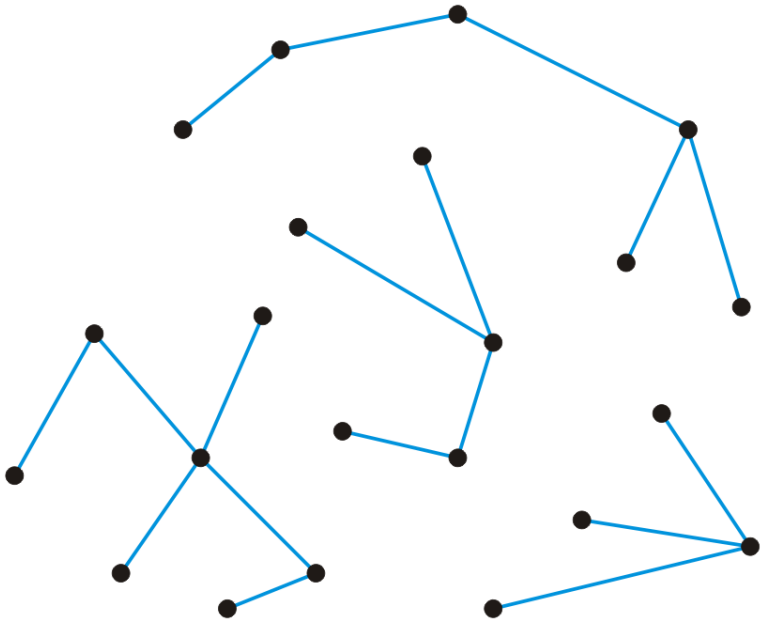
A forest is any graph that has no cycles

Consequences:

- The number of edges is $|E| < |V|$
- The number of trees is $|V| - |E|$
- Removing any one edge adds one more tree to the forest

Here is a forest with 22 vertices and 18 edges

- There are four trees



Directed graphs

In a *directed graph*, the edges on a graph are be associated with a direction

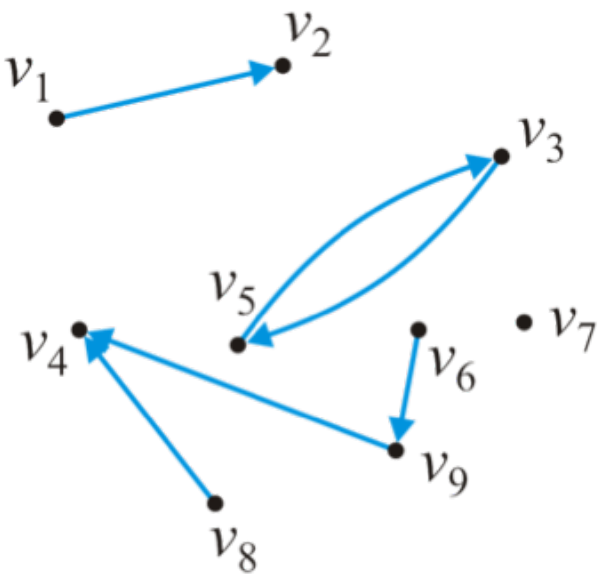
- Edges are ordered pairs (v_j, v_k) denoting a connection from v_j to v_k
- The edge (v_j, v_k) is different from the edge (v_k, v_j)

Directed graphs

Given our graph of nine vertices $V = \{v_1, v_2, \ldots v_9\}$

– These six pairs (v_j , v_k) are *directed edges*

$$E = \{(v_1, v_2), (v_3, v_5), (v_5, v_3), (v_6, v_9), (v_8, v_4), (v_9, v_4)\}$$



Directed graphs

The maximum number of directed edges in a directed graph is

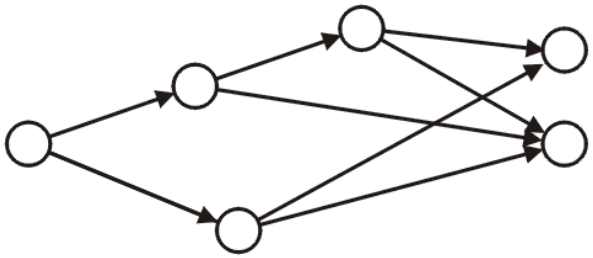
$$|E| \le 2\binom{|V|}{2} = 2\frac{|V|(|V|-1)}{2} = |V|(|V|-1) = \mathrm{O}(|V|^2)$$

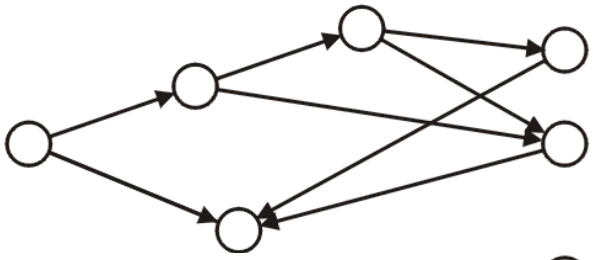
Directed acyclic graphs

A *directed acyclic graph* is a directed graph which has no cycles

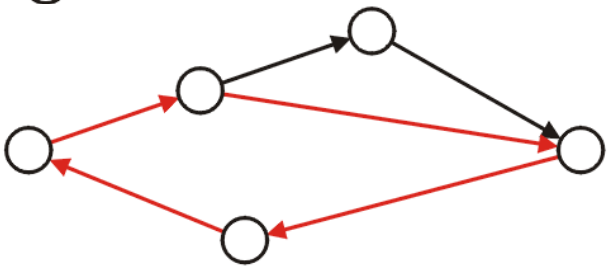
- These are commonly referred to as DAGs
- They are graphical representations of partial orders on a finite number of elements

These two are DAGs:

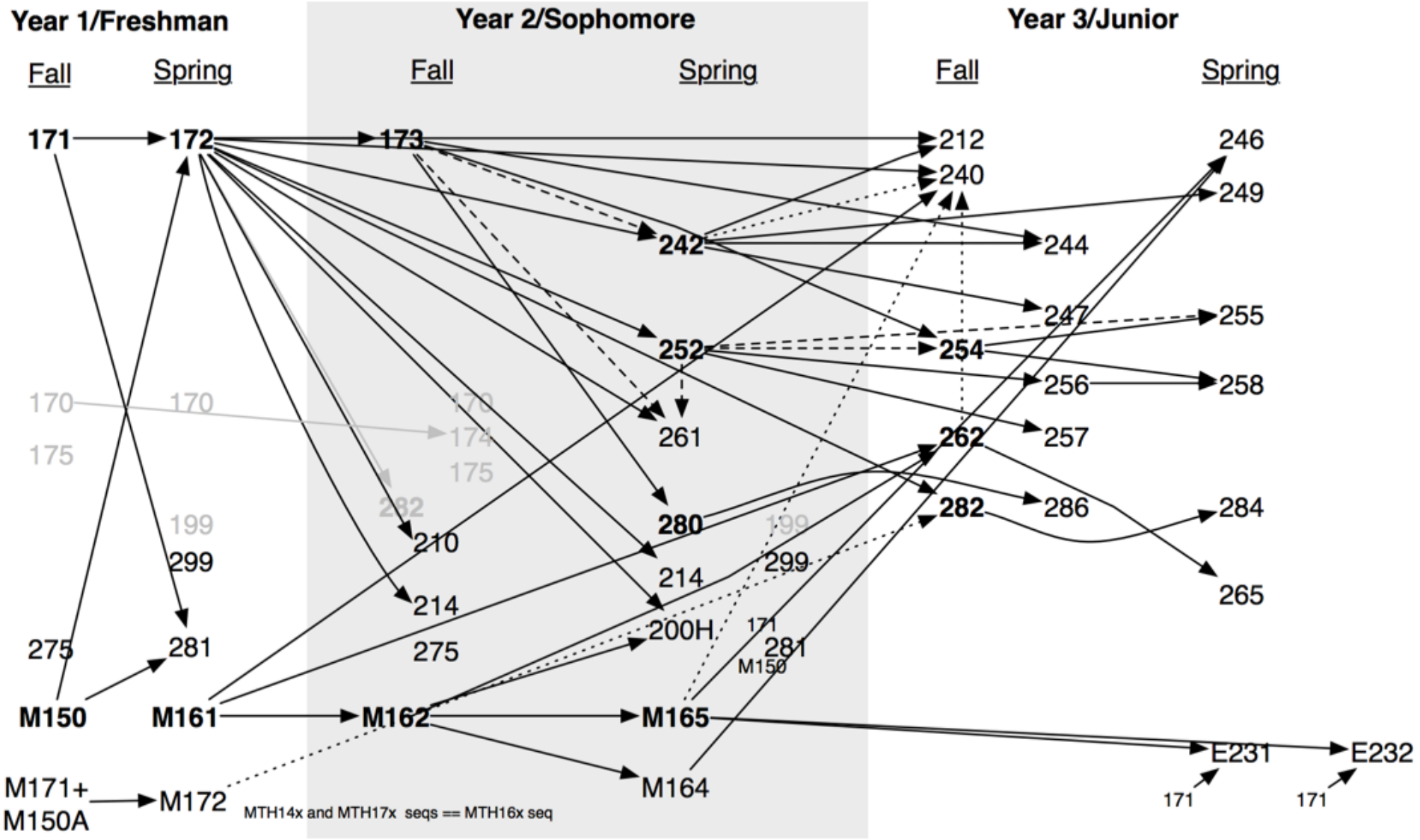




This directed graph is not acyclic:



Another DAG


Year 1/Freshman
Year 2/Sophomore
Year 3/Junior
Fall
Spring
Fall
Spring
Fall
Spring
171
172
173
212
246
240
249
242
244
247
255
252
254
256
258
261
257
262
170
170
170
174
175
175
175
282
286
284
199
280
199
210
299
214
265
299
214
200H
171
275
281
275
281
M150
M150
M161
M162
M165
E231
E232
M164
M171+
M150A
M172
MTH14x and MTH17x seqs == MTH16x seq
171
171

GRAPH TRAVERSALS

Outline

We will look at traversals of graphs

- Breadth-first or depth-first traversals
- Must avoid cycles
- Depth-first traversals can be recursive or iterative

Strategies

Traversals of graphs are also called *searches*

We can use either breadth-first or depth-first traversals

- Breadth-first requires a queue
- Depth-first requires a stack

We each case, we will have to track which vertices have been visited requiring $\Theta(|V|)$ memory

- One option for storing this is a Map (HashMap or TreeMap)

The time complexity cannot be better than and should not be worse than $\Theta(|V| + |E|)$

- Connected graphs simplify this to $\Theta(|E|)$
- Worst case: $\Theta(|V|^2)$

Breadth-first traversal

Consider implementing a breadth-first traversal on a graph:

- Choose any vertex, mark it as visited and push it onto queue
- While the queue is not empty:
  - Pop to top vertex v from the queue
  - For each vertex adjacent to v that has not been visited:
    - Mark it visited, and
    - Push it onto the queue

This continues until the queue is empty

- Note: if there are no unvisited vertices, the graph is connected,

Iterative Breadth-first traversal

An implementation can use a queue

Does this remind you of any other algorithm?

Level order traversal

Breadth-first traversal

The size of the queue is $O(|V|)$

- The size depends both on:
  - The number of edges, and
  - The out-degree of the vertices

Depth-first traversal

Consider implementing a depth-first traversal on a graph:

- Choose any vertex, mark it as visited
- From that vertex:
  - If there is another adjacent vertex not yet visited, go to it
  - Otherwise, go back to the most previous vertex that has not yet had all of its adjacent vertices visited and continue from there
- Continue until no visited vertices have unvisited adjacent vertices

Two implementations:

- Recursive
- Iterative

Recursive depth-first traversal

A recursive implementation uses the call stack for memory:

Iterative depth-first traversal

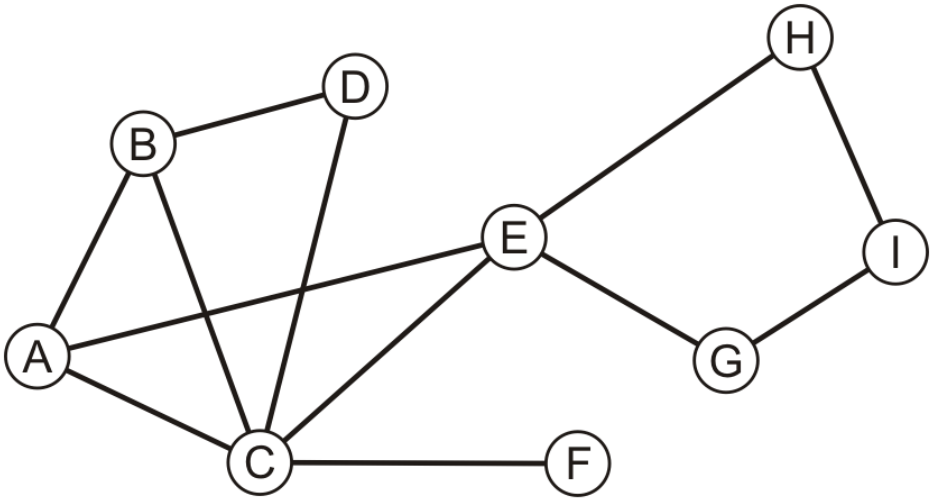
An iterative implementation can use a stack

Does this remind you of any other algorithm?

Reverse Pre-order traversal

Example

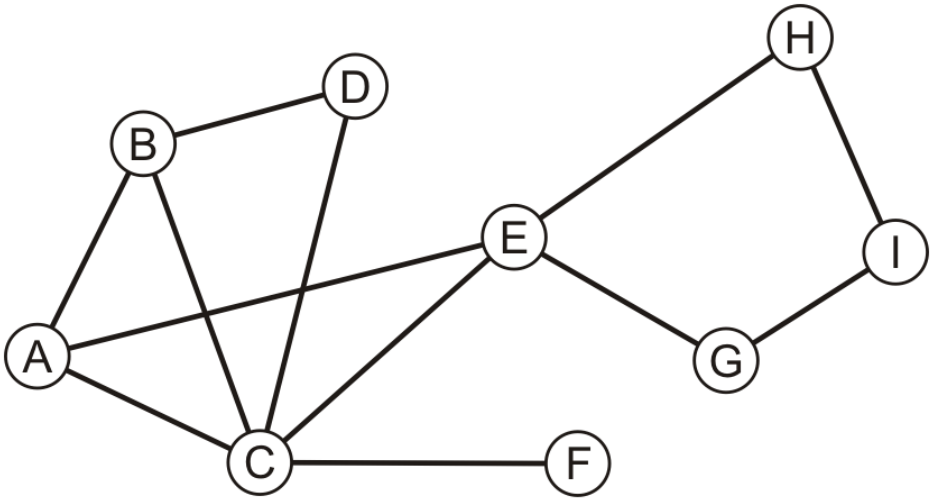
Consider this graph



Example

Performing a breadth-first traversal

– Push the first vertex onto the queue



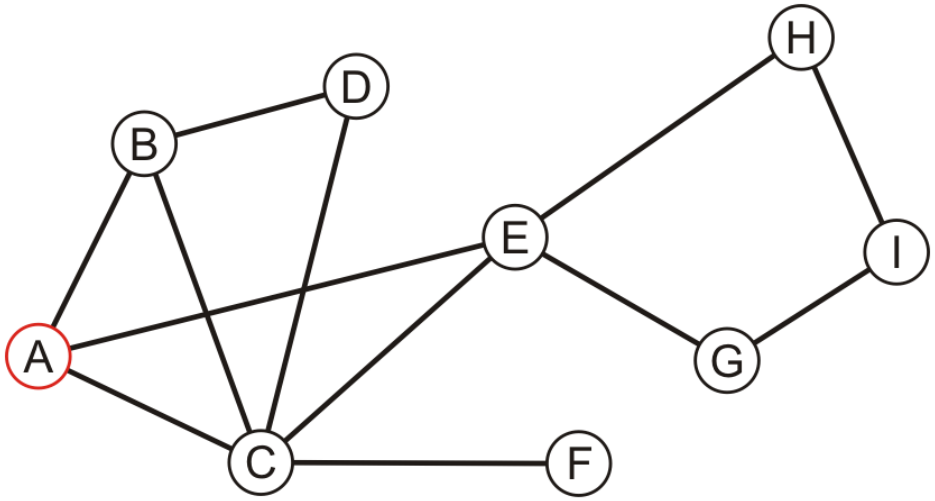
A					

Example

Performing a breadth-first traversal

– Pop A and push B, C and E

A



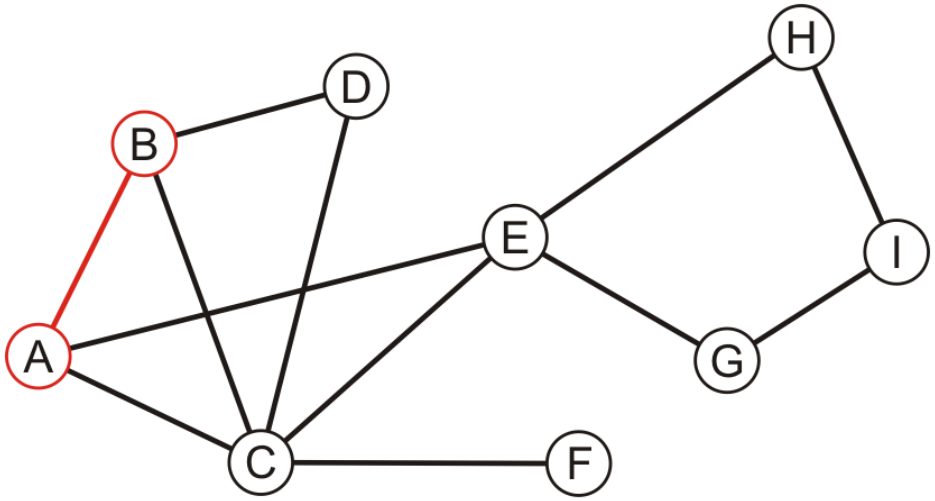
B	C	E			

Example

Performing a breadth-first traversal:

– Pop B and push D

A, B



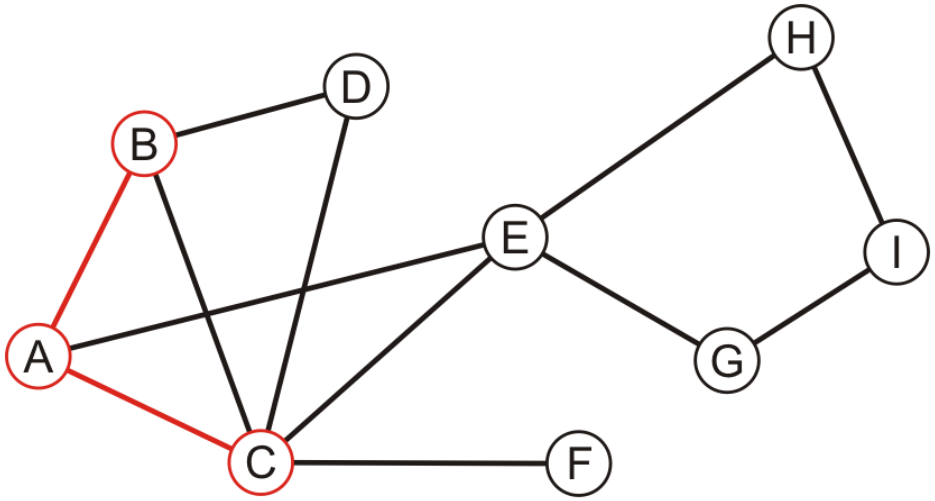
C	E	D			

Example

Performing a breadth-first traversal:

– Pop C and push F

A, B, C



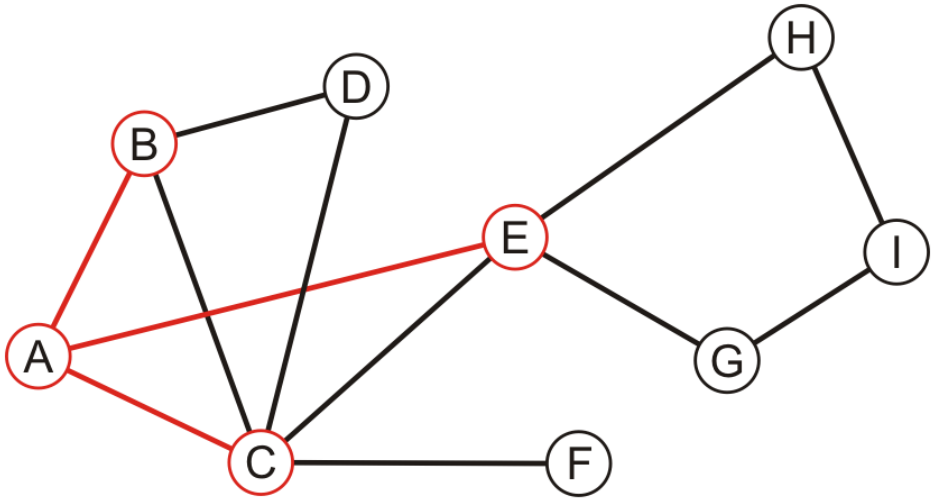
E	D	F			

Example

Performing a breadth-first traversal:

– Pop E and push G and H

A, B, C, E



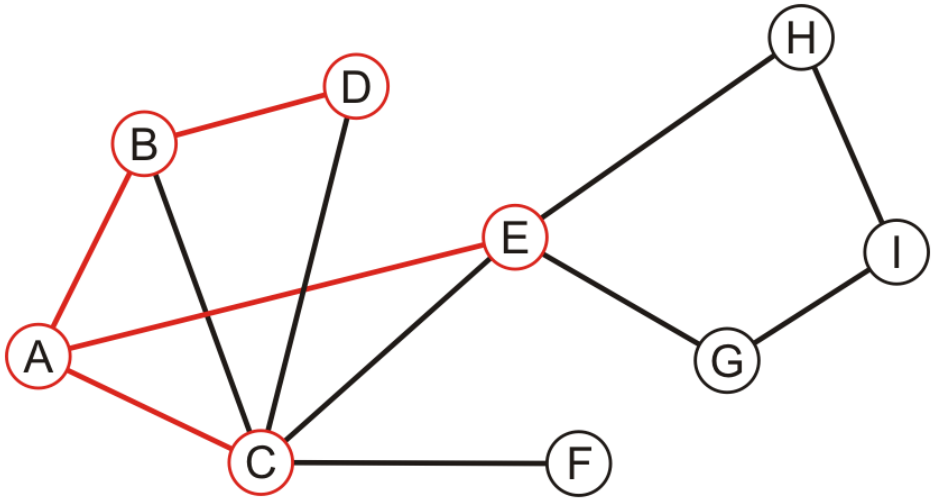
D	F	G	H		

Example

Performing a breadth-first traversal:

– Pop D

A, B, C, E, D



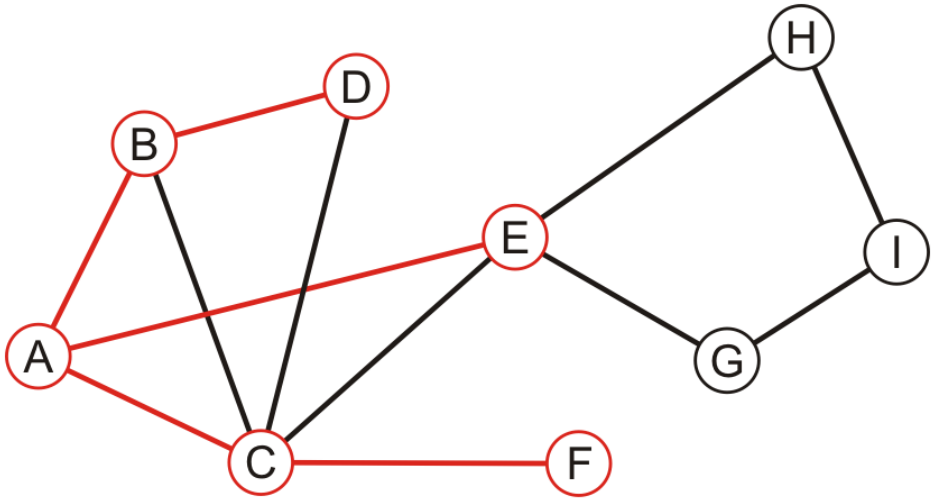
F	G	H			

Example

Performing a breadth-first traversal:

– Pop F

A, B, C, E, D, F



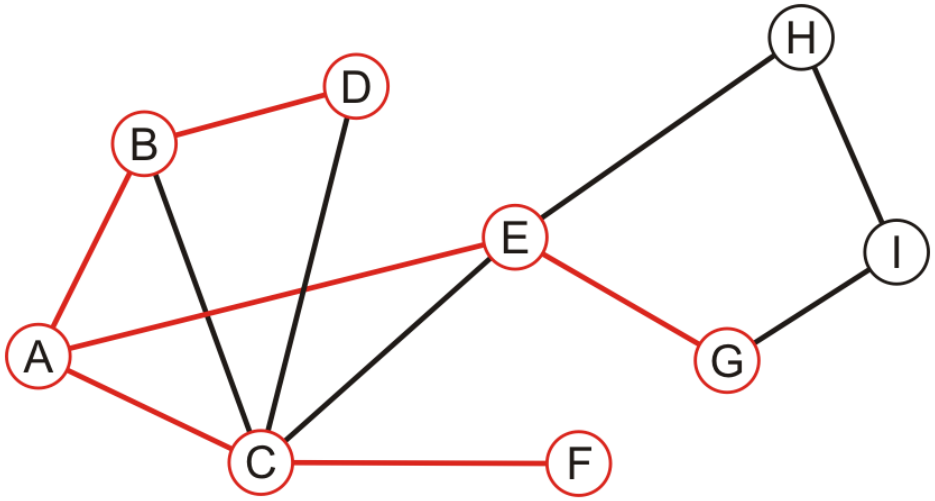
G	H				

Example

Performing a breadth-first traversal:

– Pop G and push I

A, B, C, E, D, F, G



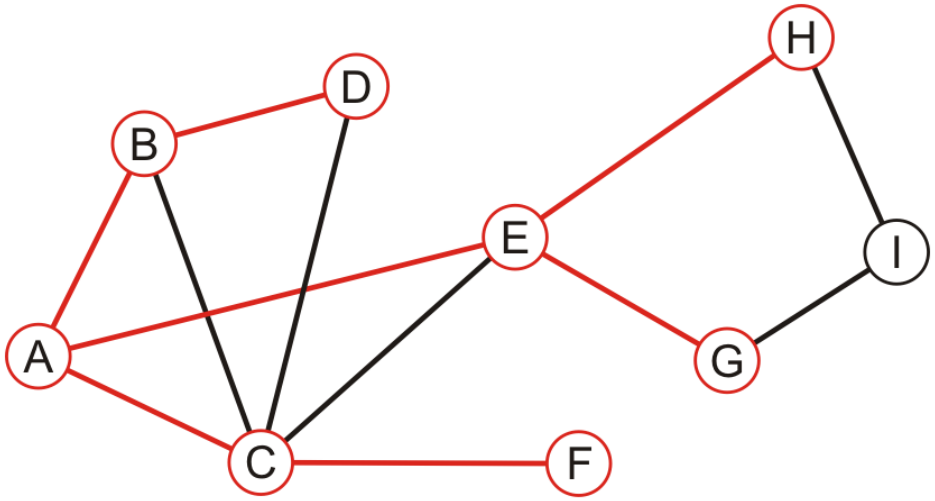
H	I				

Example

Performing a breadth-first traversal:

– Pop H

A, B, C, E, D, F, G, H



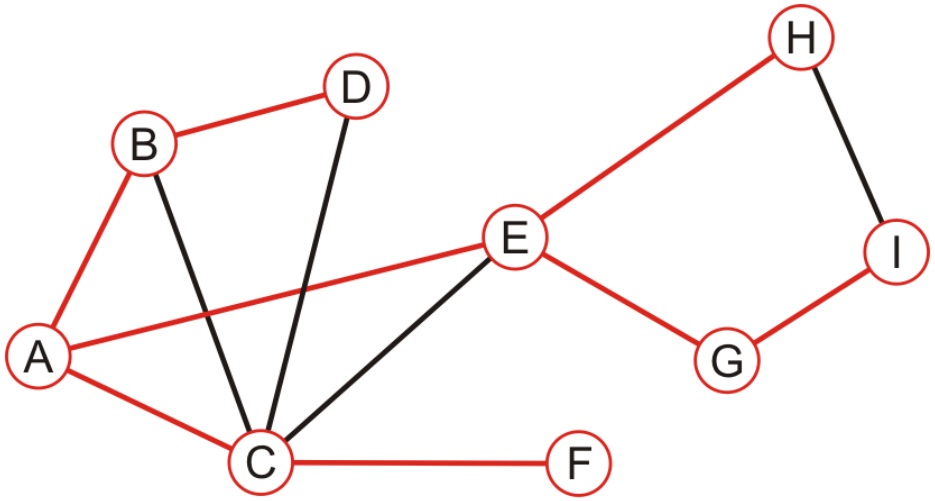
I					

Example

Performing a breadth-first traversal:

– Pop I

A, B, C, E, D, F, G, H, I

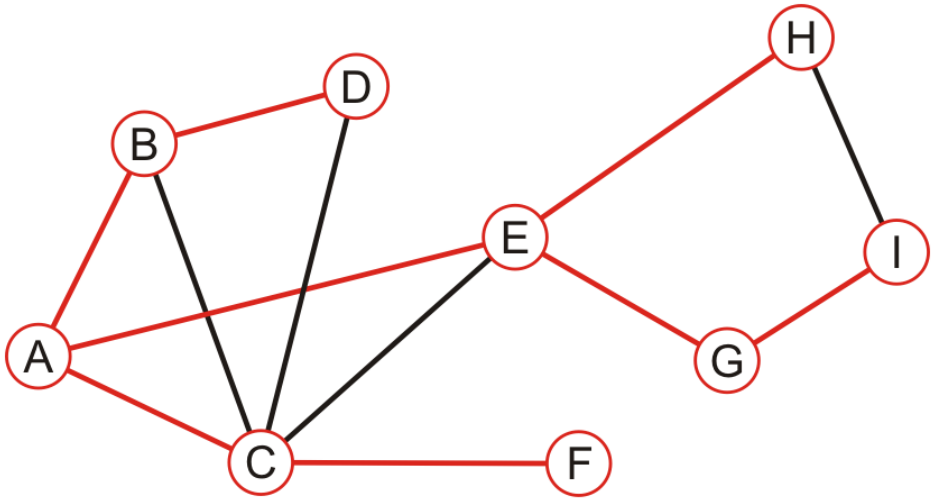


Example

Performing a breadth-first traversal:

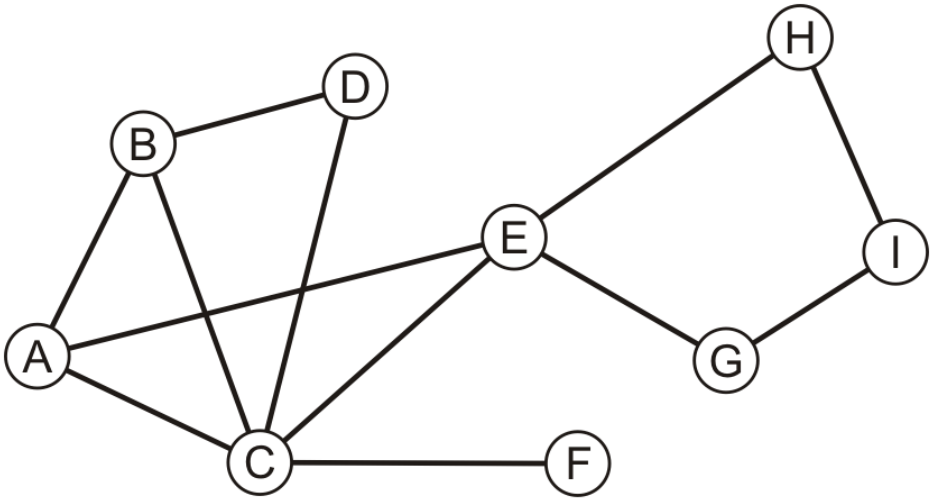
– The queue is empty: we are finished

A, B, C, E, D, F, G, H, I



Example

Perform a recursive depth-first traversal on this same graph

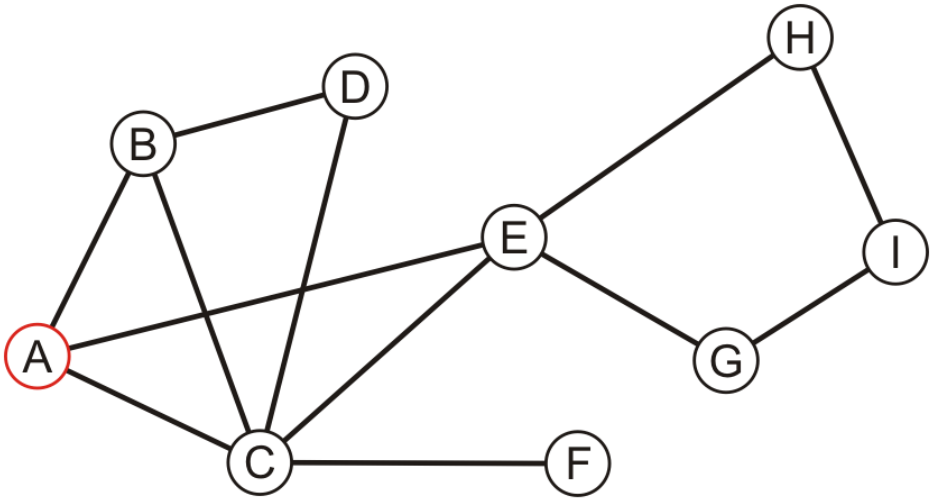


Example

Performing a recursive depth-first traversal:

– Visit the first node

A

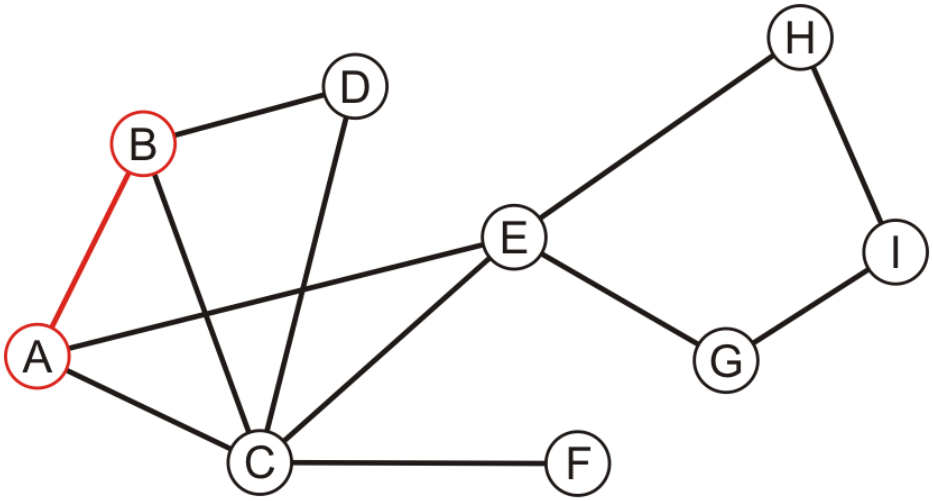


Example

Performing a recursive depth-first traversal:

– A has an unvisited neighbor

A, B

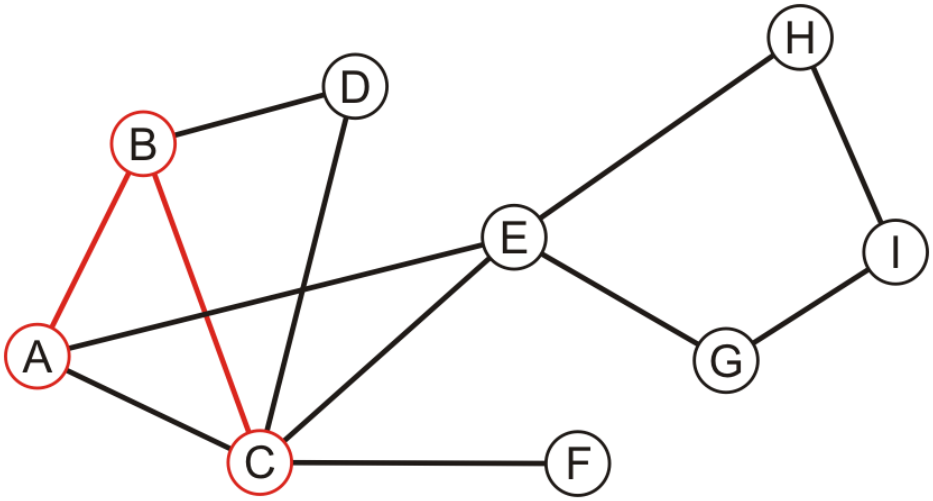


Example

Performing a recursive depth-first traversal:

– B has an unvisited neighbor

A, B, C

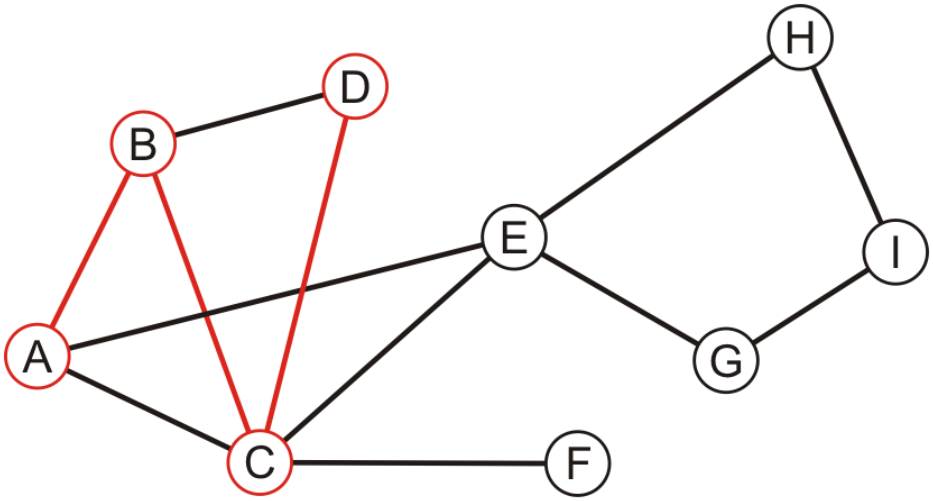


Example

Performing a recursive depth-first traversal:

– C has an unvisited neighbor

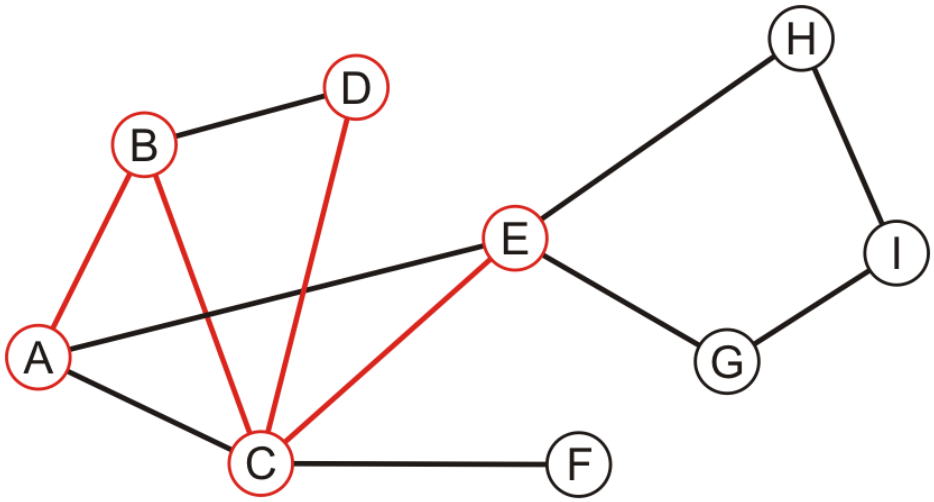
A, B, C, D



Example

Performing a recursive depth-first traversal:

– D has no unvisited neighbors, so we return to C and continue A, B, C, D, E

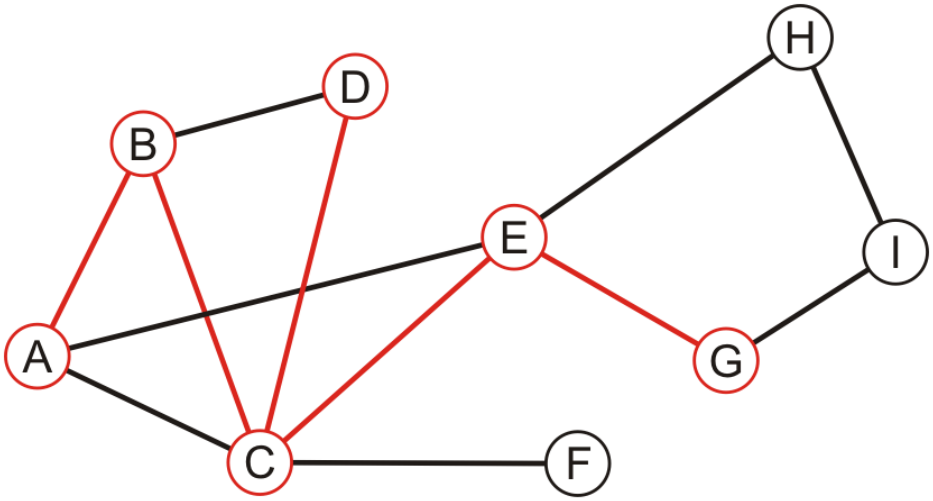


Example

Performing a recursive depth-first traversal:

– E has an unvisited neighbor

A, B, C, D, E, G

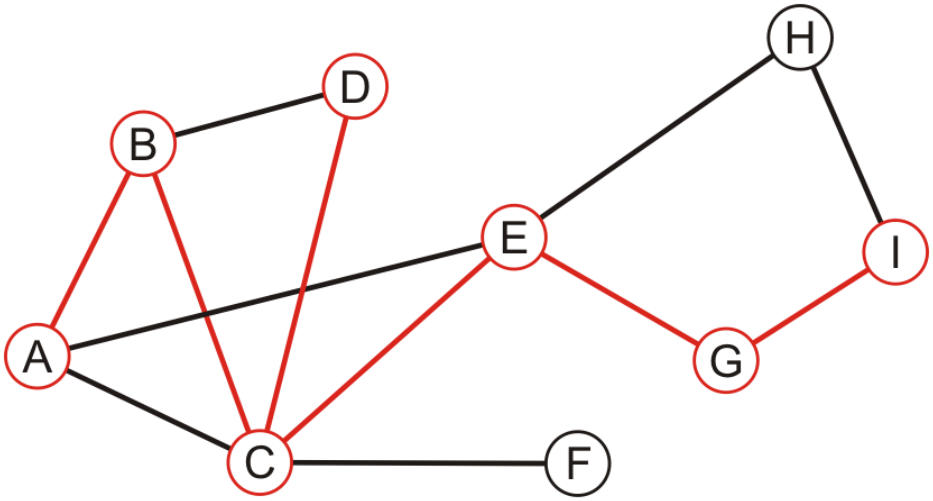


Example

Performing a recursive depth-first traversal:

– F has an unvisited neighbor

A, B, C, D, E, G, I

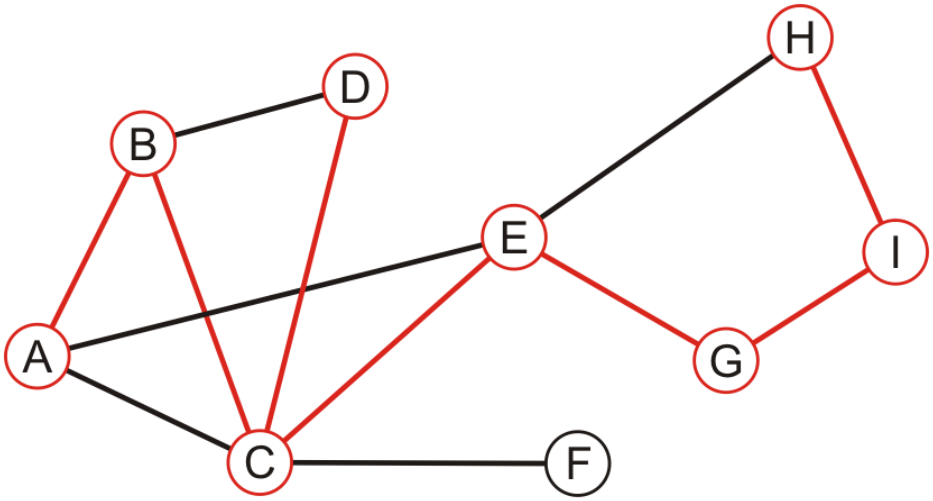


Example

Performing a recursive depth-first traversal:

– H has an unvisited neighbor

A, B, C, D, E, G, I, H

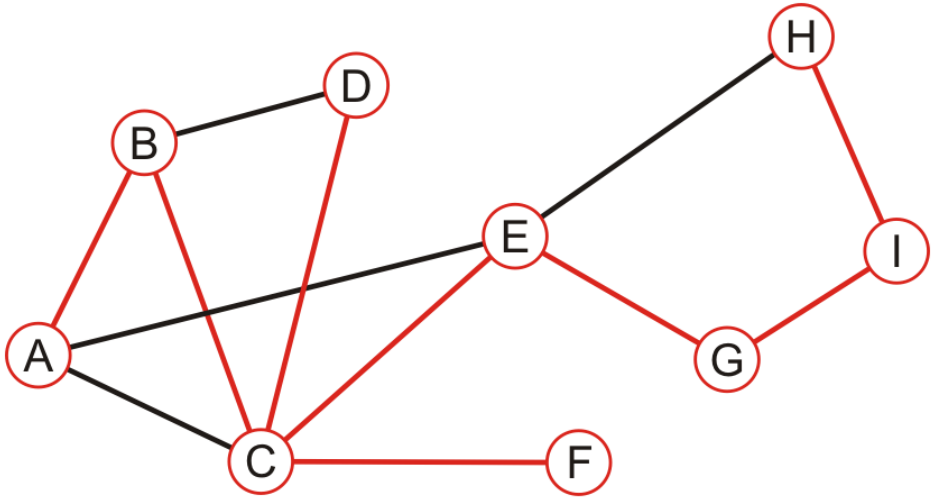


Example

Performing a recursive depth-first traversal:

– We recurse back to C which has an unvisited neighbour

A, B, C, D, E, G, I, H, F

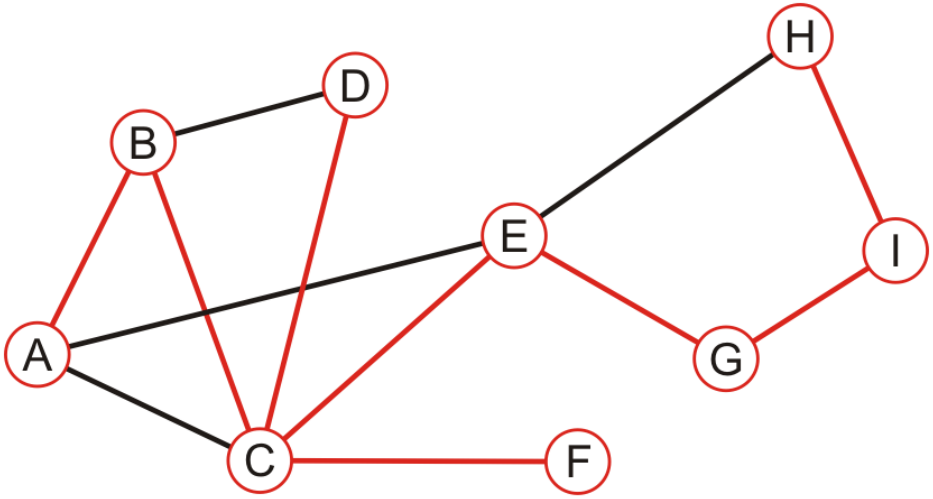


Example

Performing a recursive depth-first traversal:

– We recurse finding that no other nodes have unvisited neighbours

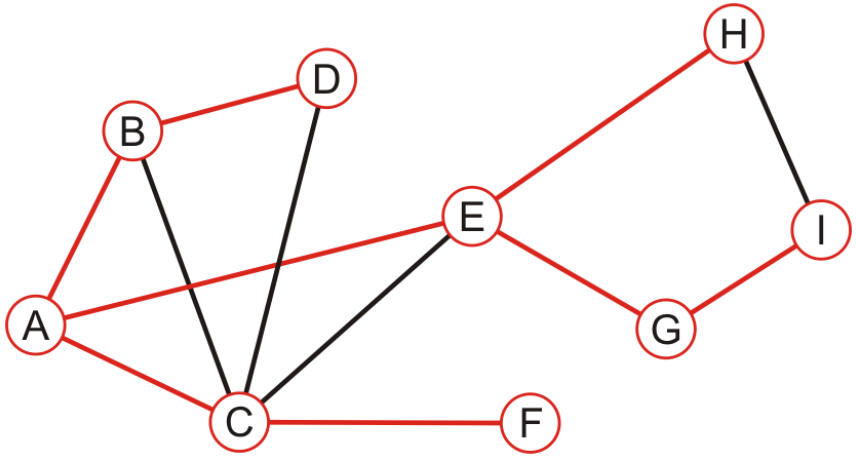
A, B, C, D, E, G, I, H, F



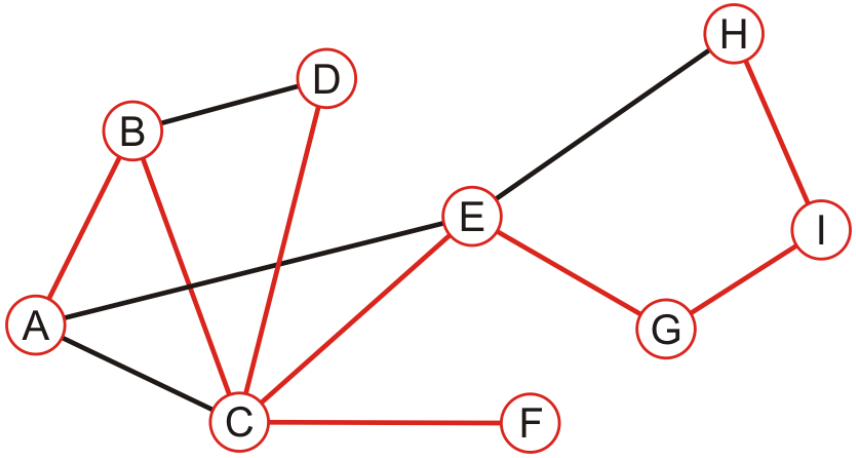
Comparison

– An iterative depth-first traversal may also be different again

A, B, C, E, D, F, G, H, I



A, B, C, D, E, G, I, H, F



The order in which vertices appear can differ greatly. But Given a particular Adjacency list and following either Iterative (using Stack) or Recursive solution, your output would be unique

Applications

Applications of tree traversals include:

- Determining connectiveness and finding connected sub-graphs
- Determining the path length from one vertex to all others

Summary

This topic covered graph traversals

- Considered breadth-first and depth-first traversals
- Depth-first traversals can recursive or iterative
- More overhead than traversals of rooted trees
- Considered an example with both implementations
- They are also called *searches*

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm

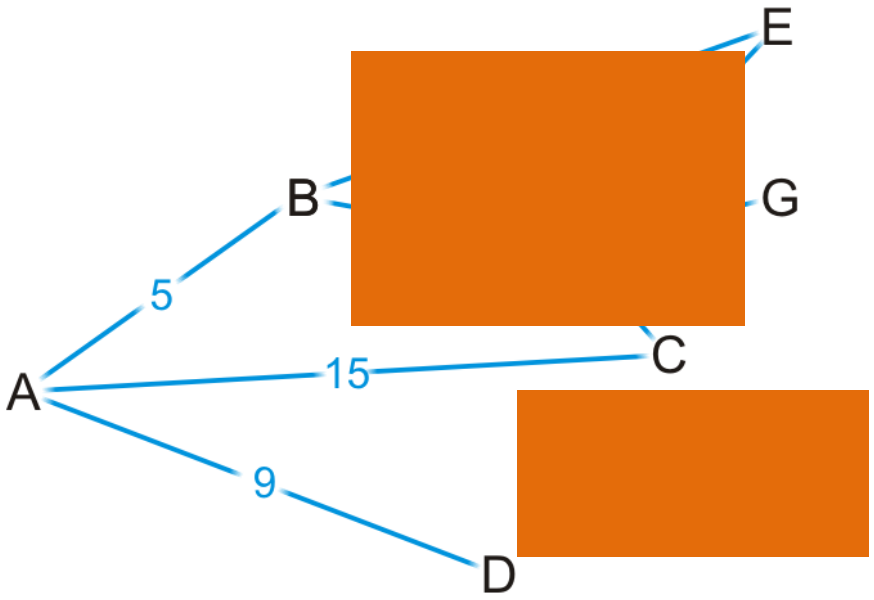
Dijkstra's algorithm solves the single-source shortest path problem

- **Edsger Wybe Dijkstra**
  - formulated and solved the shortest path problem for a demonstration at the official inauguration of the ARMAC computer in 1956
  - Assumption: all the weights are positive

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Strategy

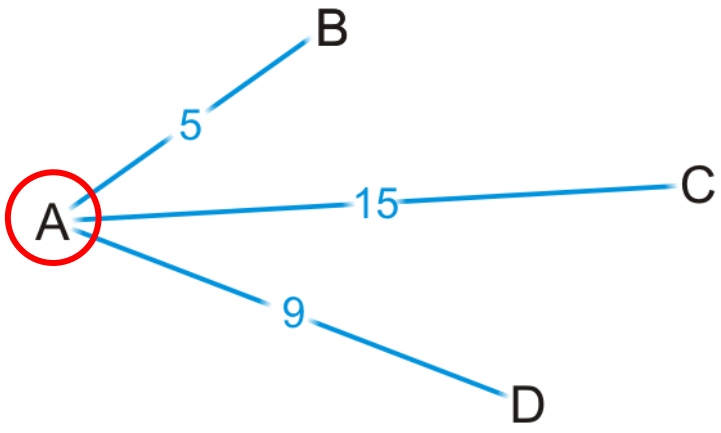
A graph with vertices A,B, C, D, E, F, G are given
Find the shortest path from A to every other vertices



Strategy

Suppose you are at vertex A

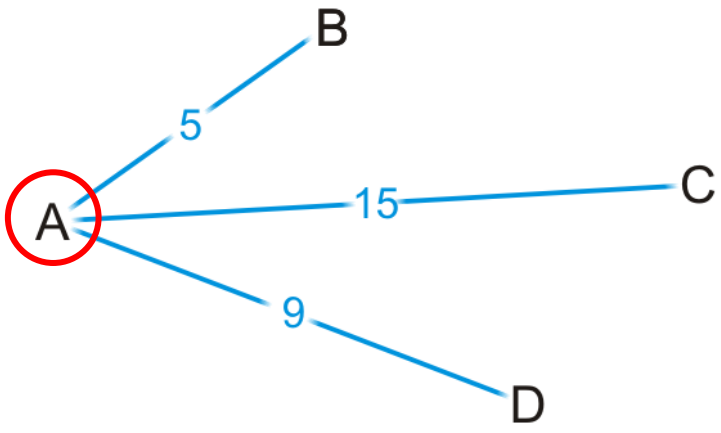
- You are aware of all vertices adjacent to it
- This information is either in an adjacency list or adjacency matrix



Strategy

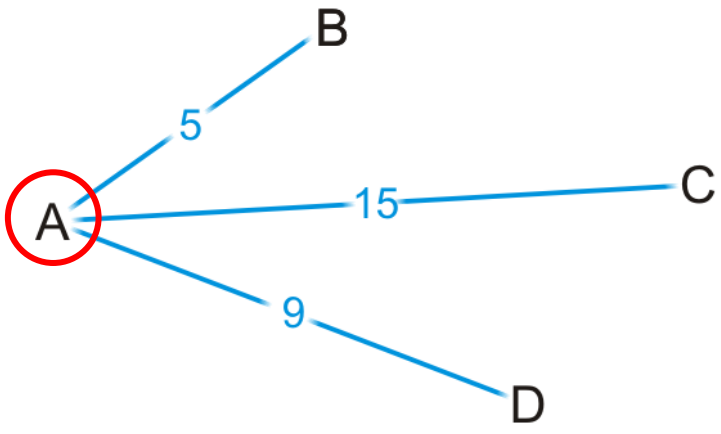
Is 5 the shortest distance to B via the edge (A, B)?

– Why or why not?



Strategy

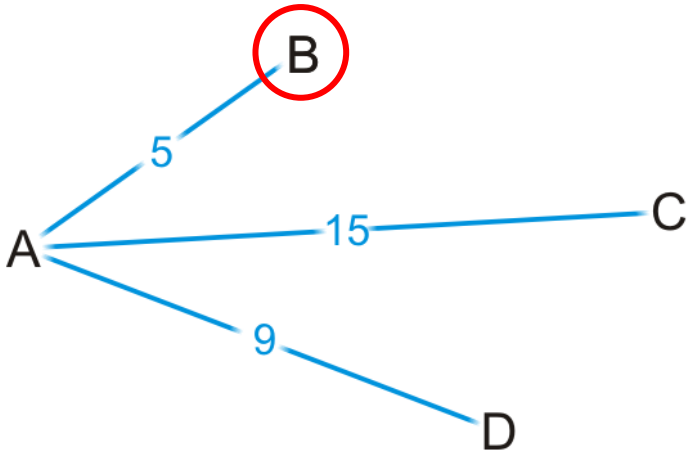
Are you guaranteed that the shortest path to C is (A, C), or that (A, D) is the shortest path to vertex D?



Strategy

We accept that (A, B) is the shortest path to vertex B from A

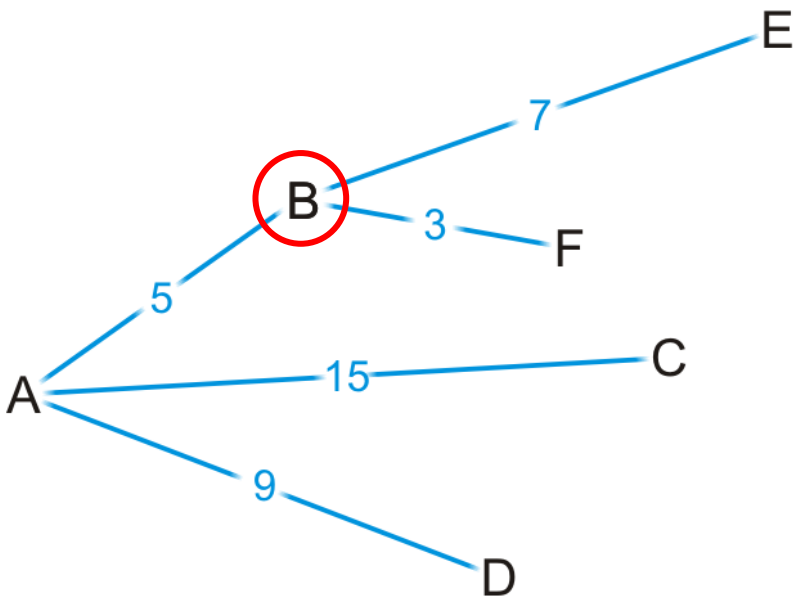
– Let's see where we can go from B



Strategy

By some simple arithmetic, we can determine that

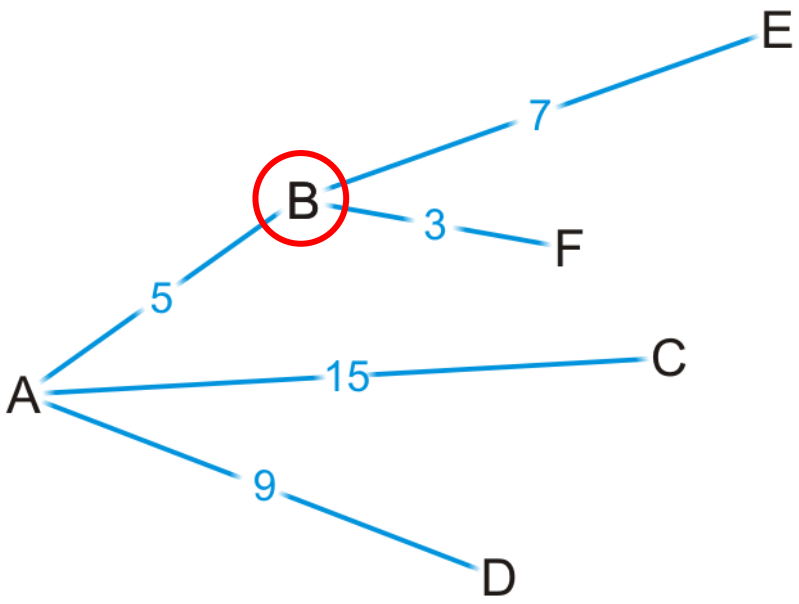
- There is a path (A, B, E) of length 5 + 7 = 12
- There is a path (A, B, F) of length 5 + 3 = 8



Strategy

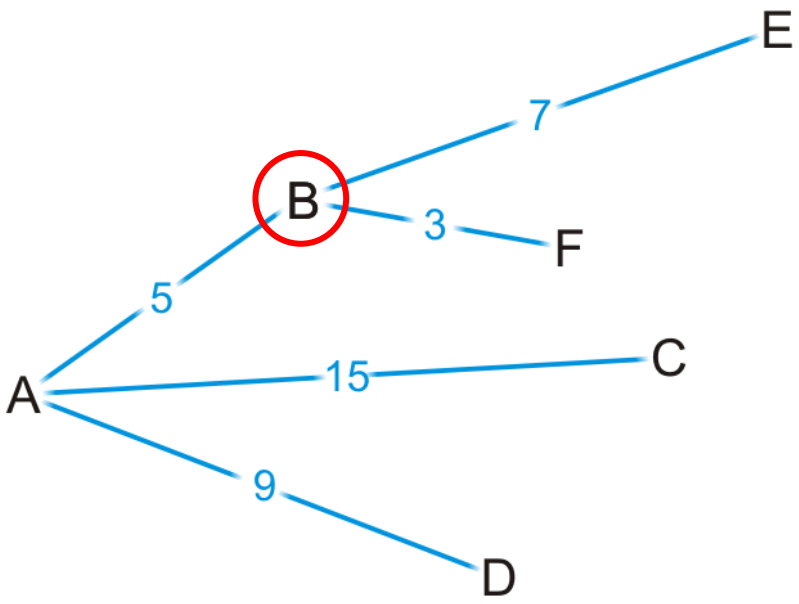
Is (A, B, F) is the shortest path from vertex A to F?

– Why or why not?



Strategy

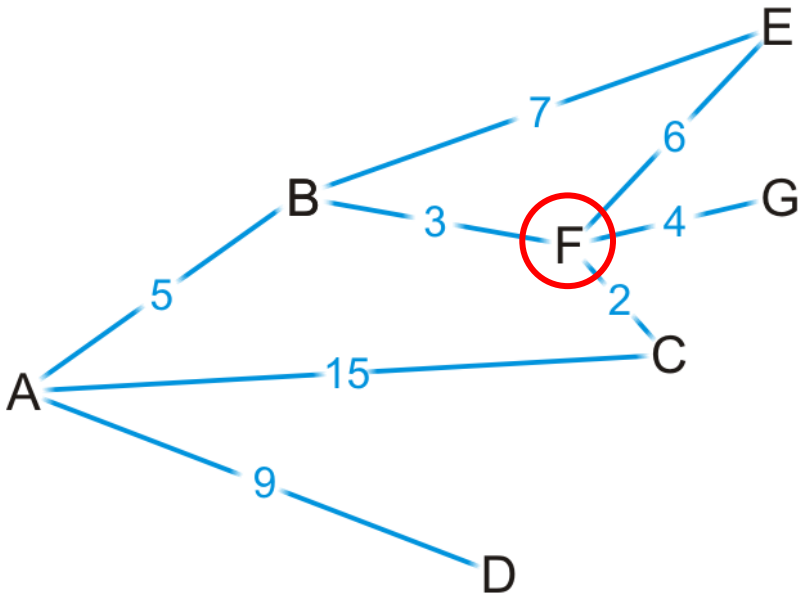
Are we guaranteed that any other path we are currently aware of is also going to be the shortest path?



Strategy

Okay, let's visit vertex F

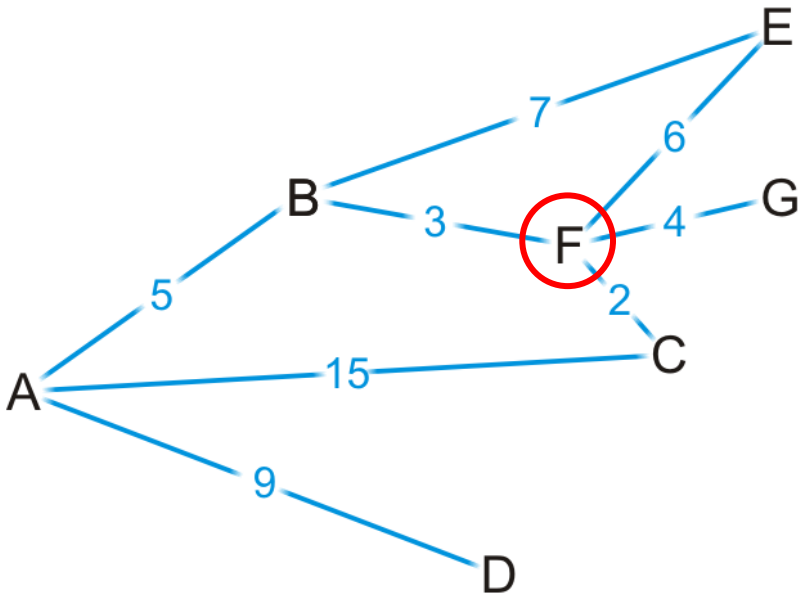
– We know the shortest path is (A, B, F) and it's of length 8



Strategy

There are three edges exiting vertex F, so we have paths:

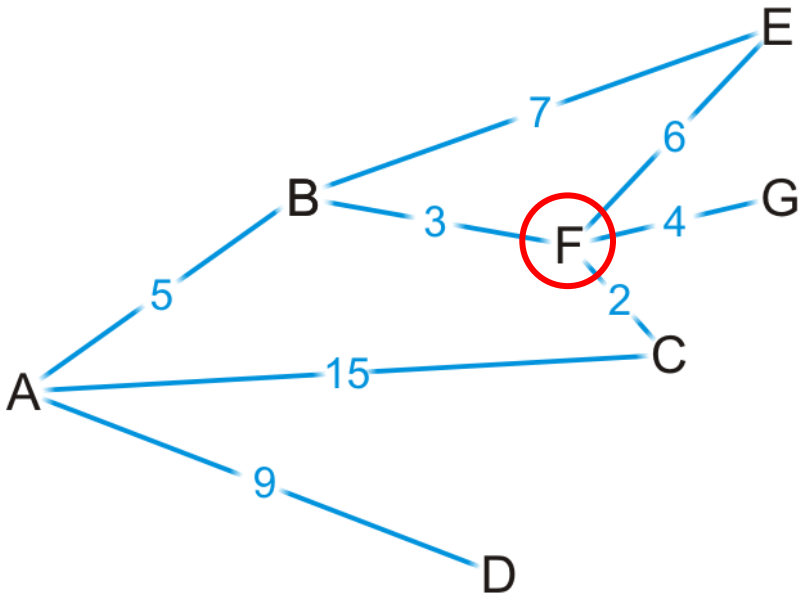
- (A, B, F, E) of length 8 + 6 = 14
- (A, B, F, G) of length 8 + 4 = 12
- (A, B, F, C) of length 8 + 2 = 10



Strategy

By observation:

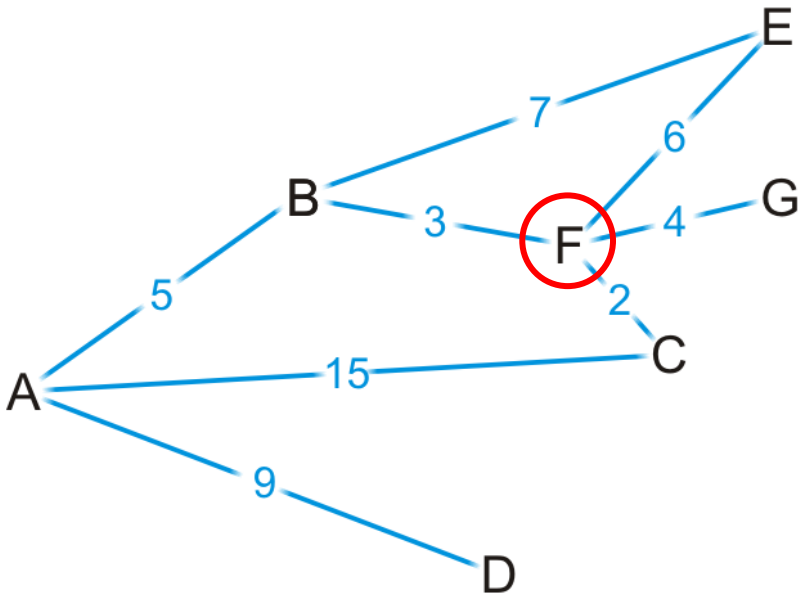
- The path (A, B, F, E) is longer than (A, B, E)
- The path (A, B, F, C) is shorter than the path (A, C)



Strategy

At this point, we've discovered the shortest paths to:

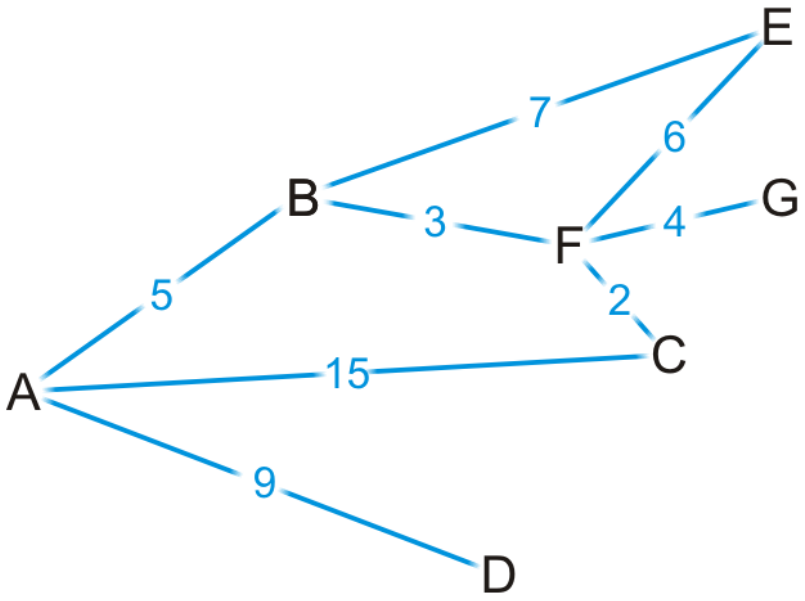
- Vertex B: (A, B) of length 5
- Vertex F: (A, B, F) of length 8



Strategy

At this point, we have the shortest distances to B and F

– Which remaining vertex are we currently guaranteed to have the shortest distance to?



Dijkstra's algorithm

We initially don't know the distance to any vertex except the initial vertex

- Each time we visit a vertex, we will examine all adjacent vertices
  - We need to track visited vertices—a Boolean table of size $|V|$
- Do we need to track the shortest path to each vertex?
  - That is, do I have to store (A, B, F) as the shortest path to vertex F?

- No, We really only have to record that the shortest path to vertex F came from vertex B
  - We would then determine that the shortest path to vertex B came from vertex A

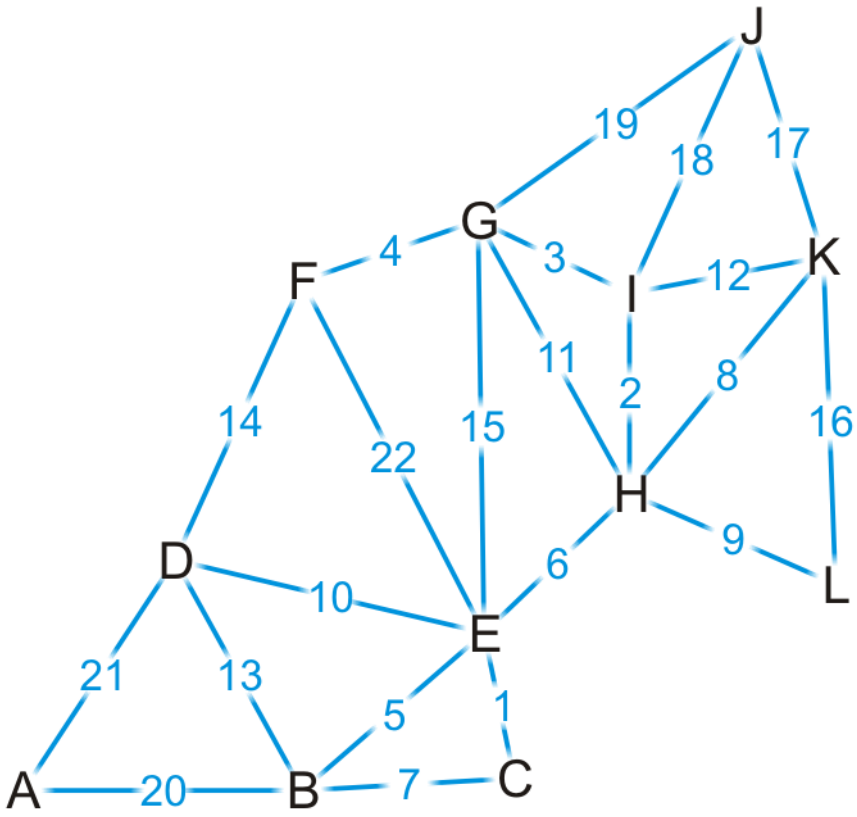
Dijkstra's algorithm

Thus, we will iterate $|V|$ times:

- Find that unvisited vertex v that has a minimum distance to it
- Mark it as having been visited
- Consider every adjacent vertex w that is unvisited:
  - Is the distance to v plus the weight of the edge (v, w) less than our currently known shortest distance to w
  - If so, update the shortest distance to w and record v as the previous pointer
- Continue iterating until all vertices are visited or all remaining vertices have a distance to them of infinity

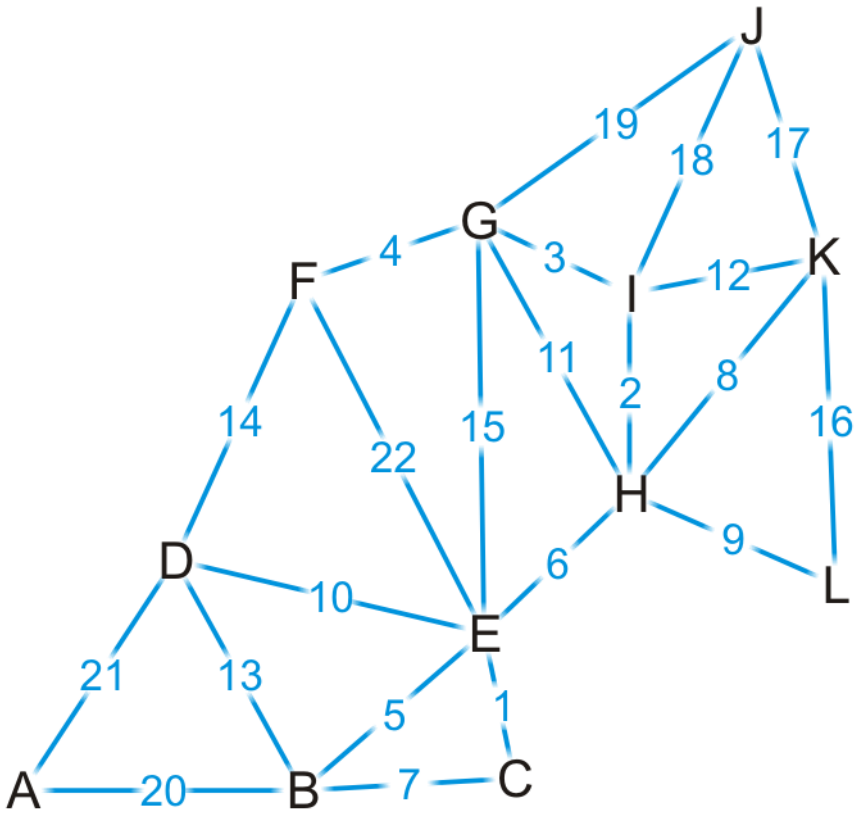
Example

Let us give a weight to each of the edges



Example

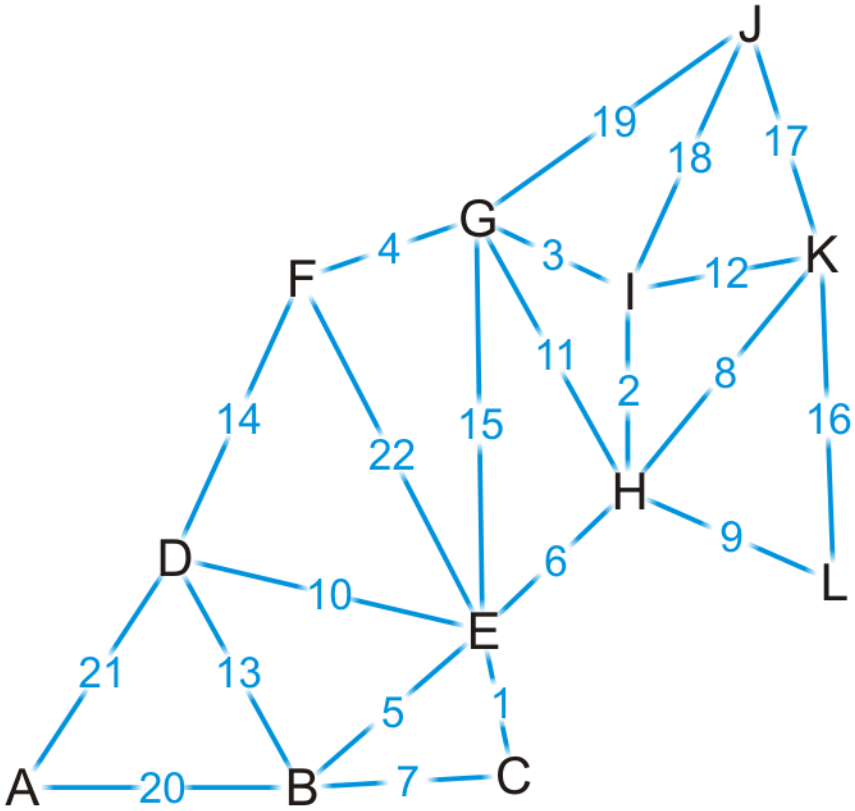
Find the shortest distance from K to every other Vertices



Example

We set up our table

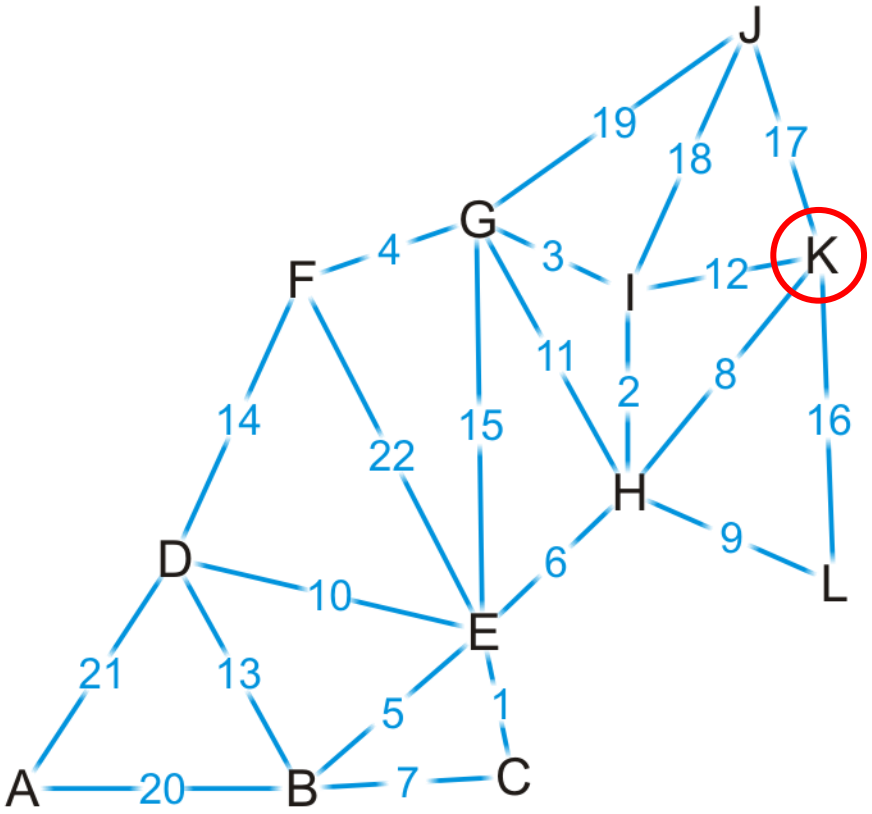
– Which unvisited vertex has the minimum distance to it?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	F	0	Ø
L	F	∞	Ø

Example

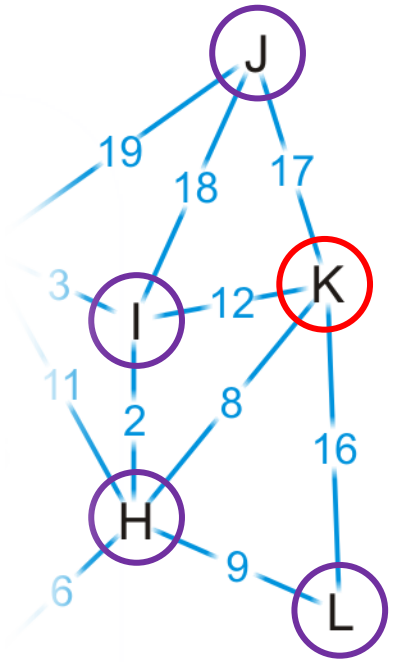
We visit vertex K



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	**T**	**0**	**Ø**
L	F	∞	Ø

Example

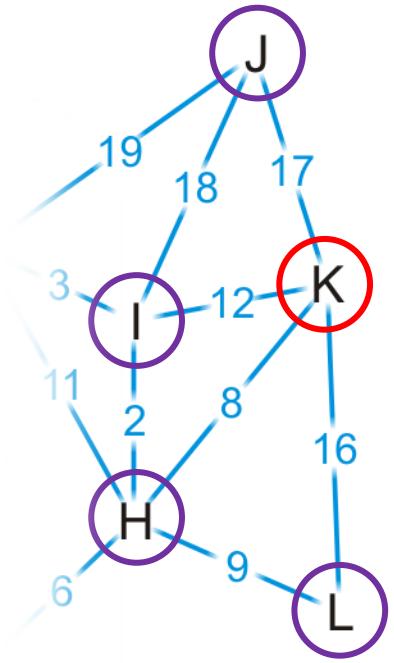
Vertex K has four neighbors: H, I, J and L



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	**T**	**0**	**Ø**
L	F	∞	Ø

Example

We have now found at least one path to each of these vertices

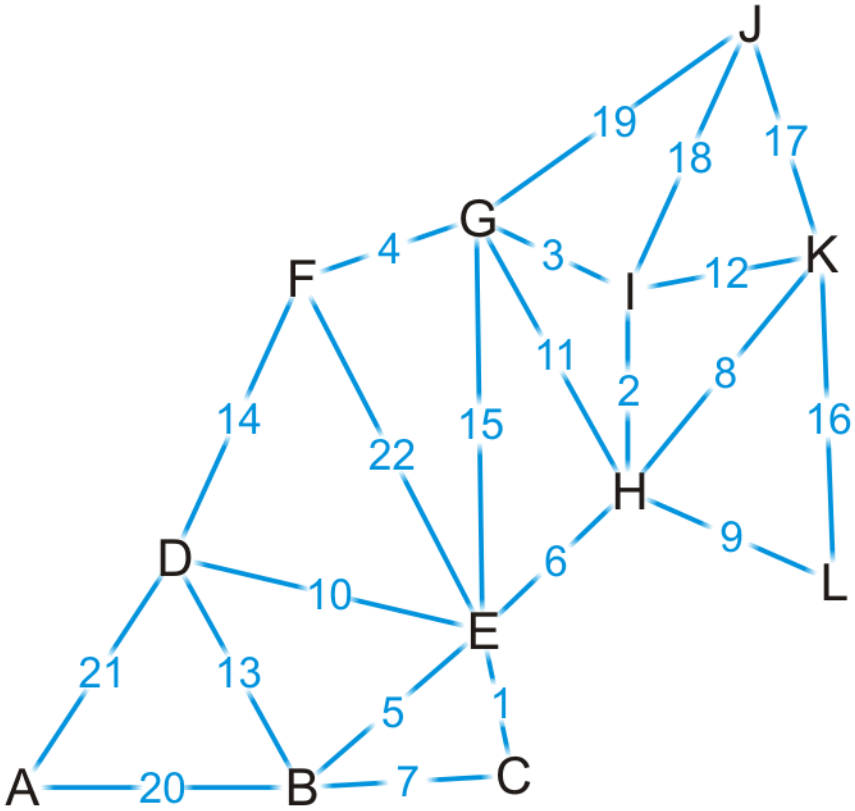


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We're finished with vertex K

– To which vertex are we now guaranteed we have the shortest path?

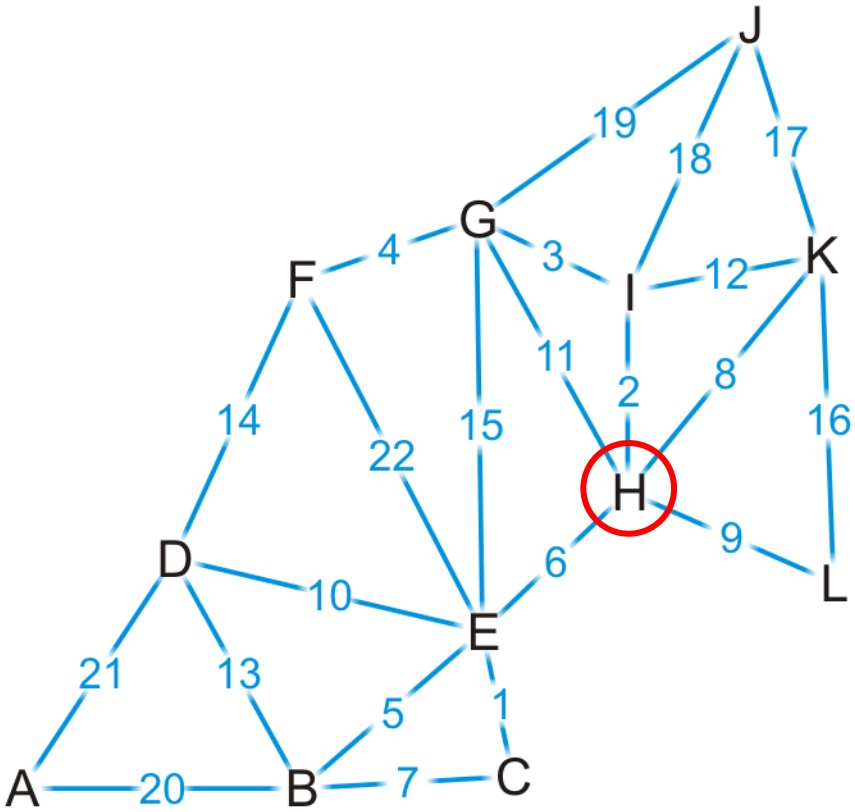


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We visit vertex H: the shortest path is (K, H) of length 8

– Vertex H has four unvisited neighbors: E, G, I, L



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	**T**	**8**	**K**
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

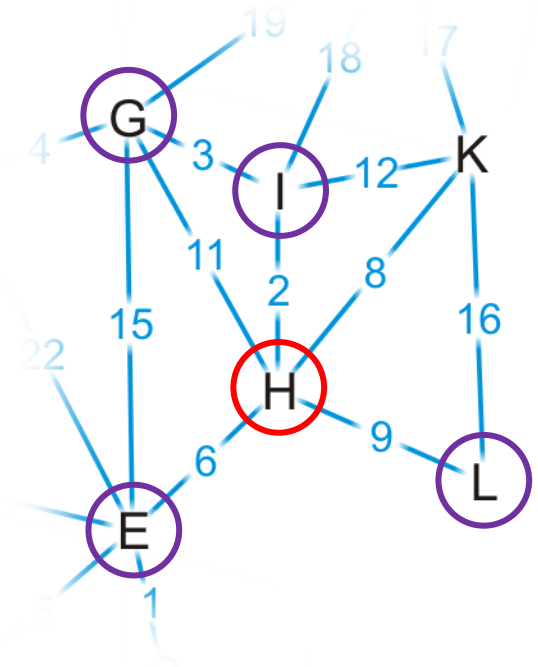
Example

Consider these paths:

(K, H, E) of length 8 + 6 = 14 (K, H, G) of length 8 + 11 = 19

(K, H, I) of length 8 + 2 = 10 (K, H, L) of length 8 + 9 = 17

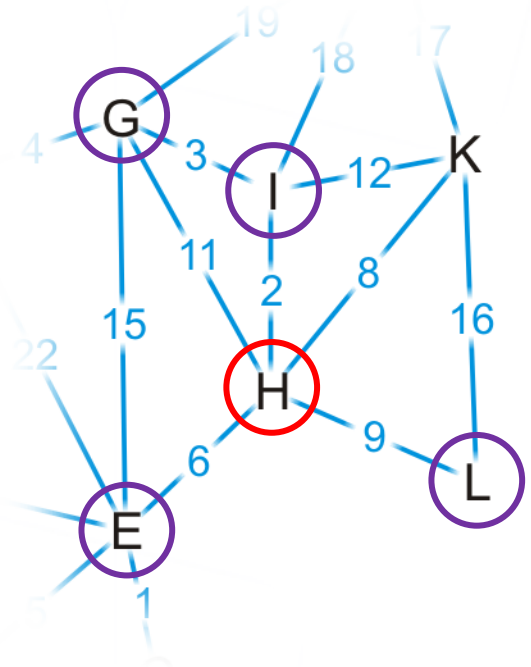
– Which of these are shorter than any known path?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	**T**	**8**	**K**
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We already have a shorter path (K, L), but we update the other three

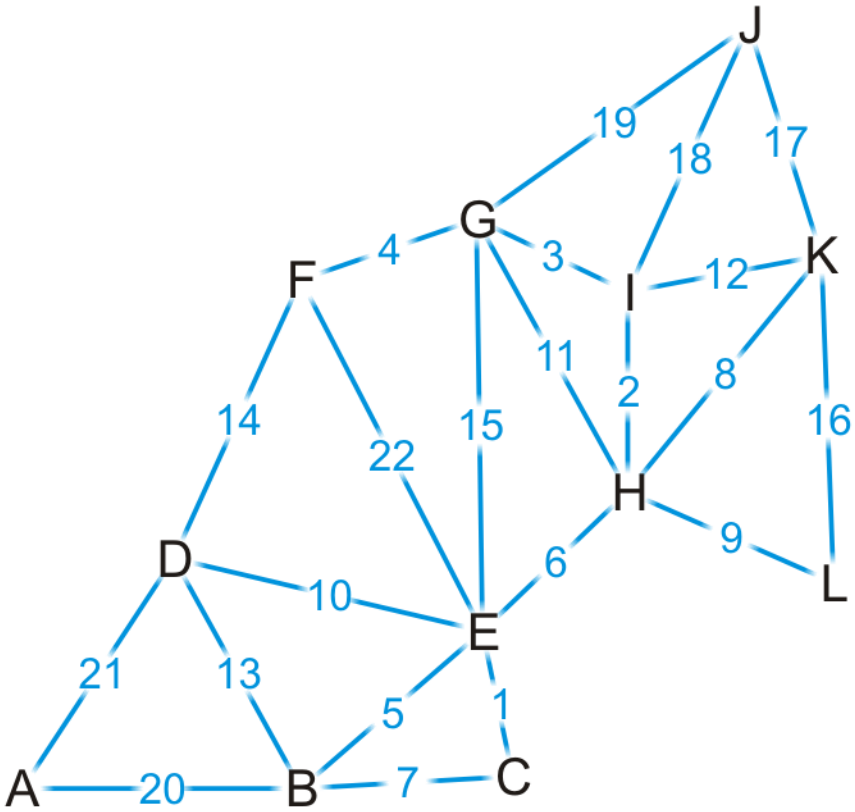


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	**14**	**H**
F	F	∞	Ø
G	F	**19**	**H**
H	**T**	**8**	**K**
I	F	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We are finished with vertex H

– Which vertex do we visit next?

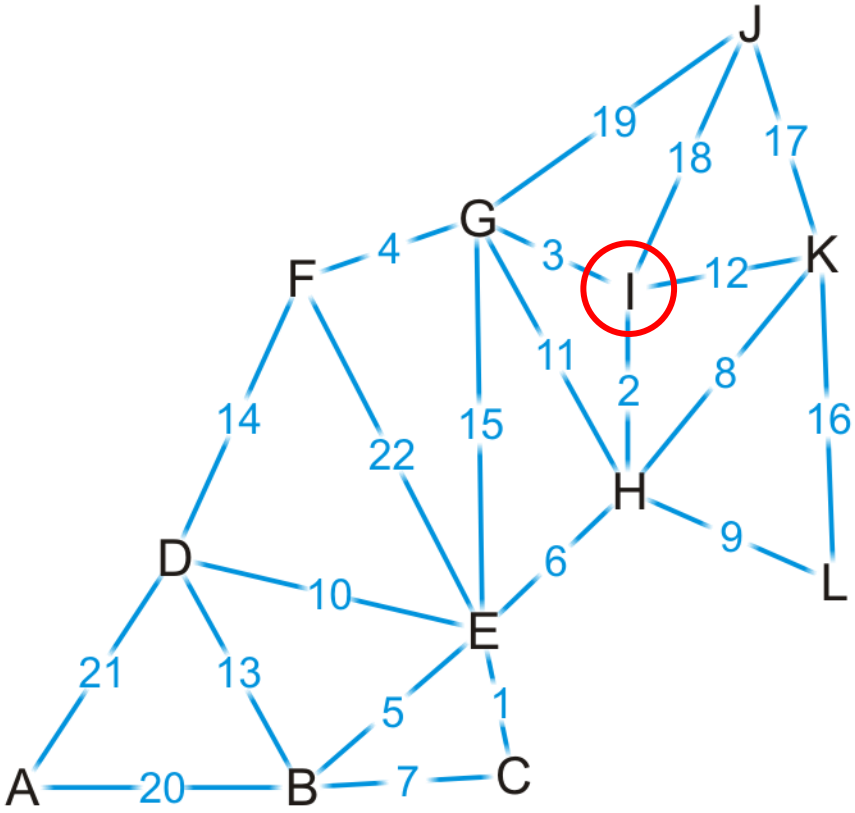


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	F	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, I) is the shortest path from K to I of length 10

– Vertex I has two unvisited neighbors: G and J

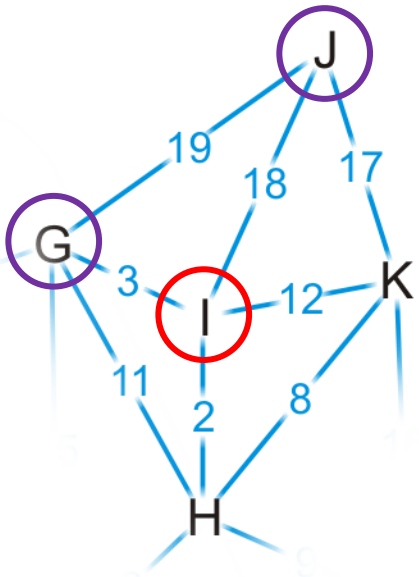


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

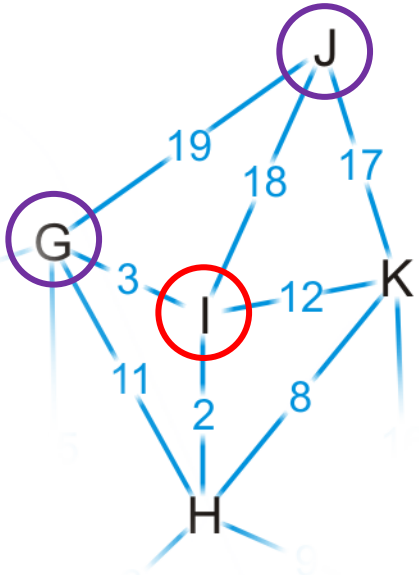
(K, H, I, G) of length 10 + 3 = 13 (K, H, I, J) of length 10 + 18 = 28



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

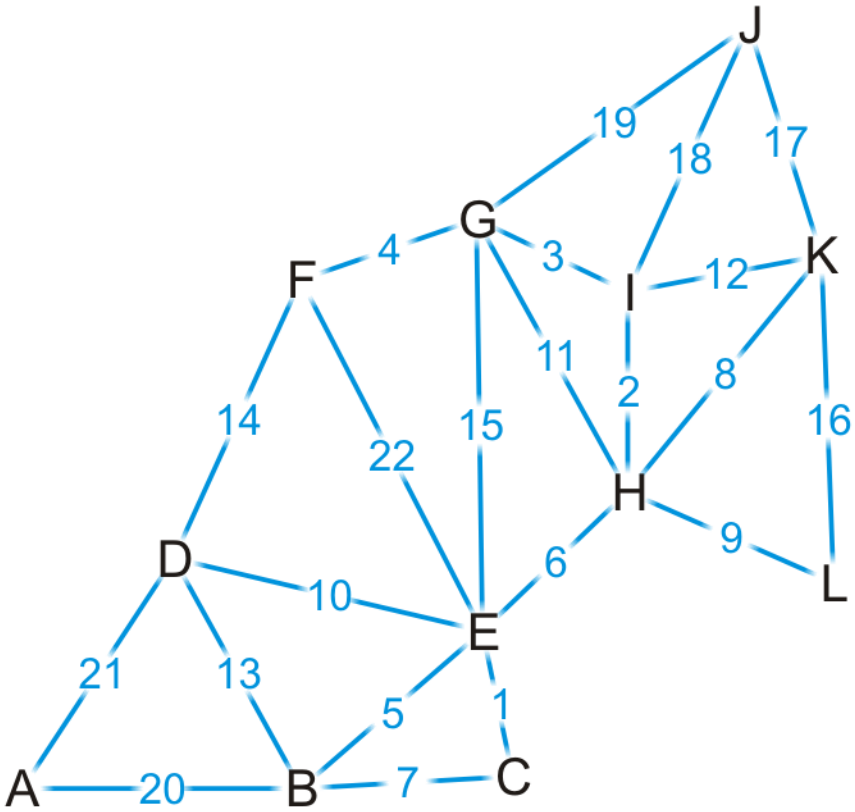
We have discovered a shorter path to vertex G, but (K, J) is still the shortest known path to vertex J



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	**13**	**I**
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Which vertex can we visit next?

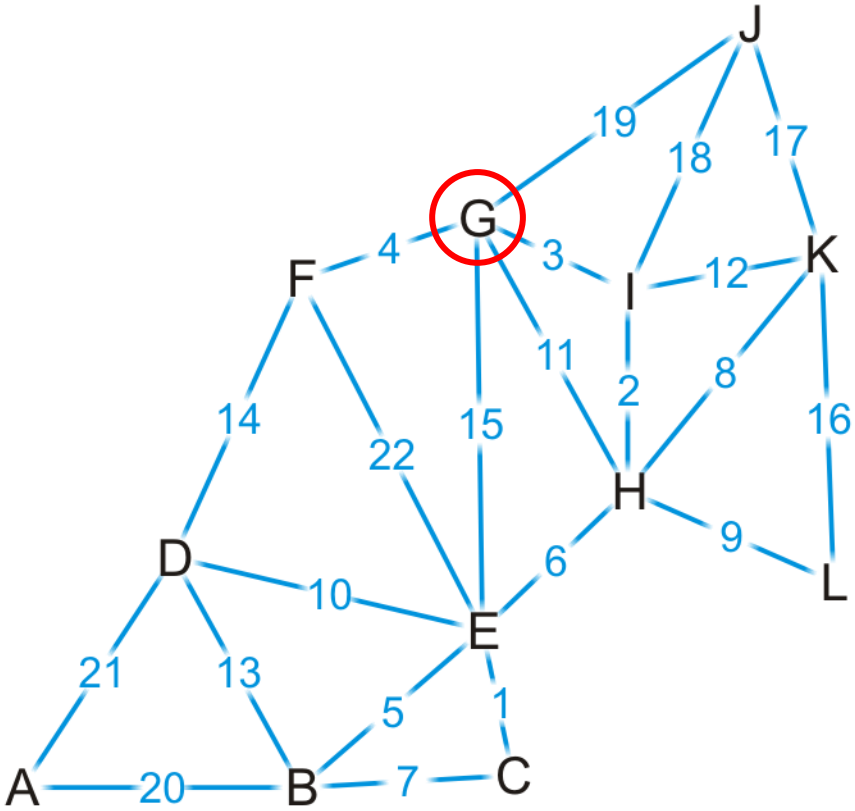


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, I, G) is the shortest path from K to G of length 13

– Vertex G has three unvisited neighbors: E, F and



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

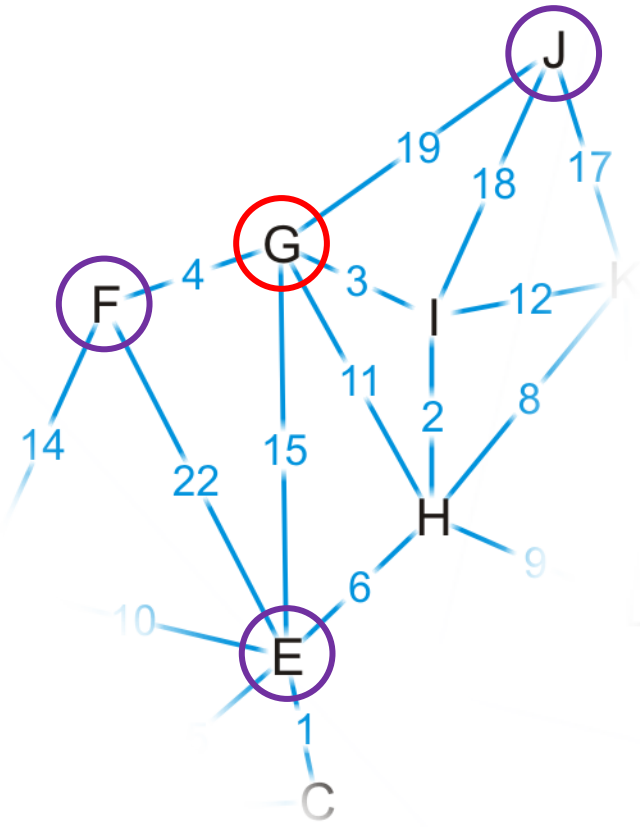
Example

Consider these paths:

(K, H, I, G, E) of length 13 + 15 = 28 (K, H, I, G, F) of length 13 + 4 = 17

(K, H, I, G, J) of length 13 + 19 = 32

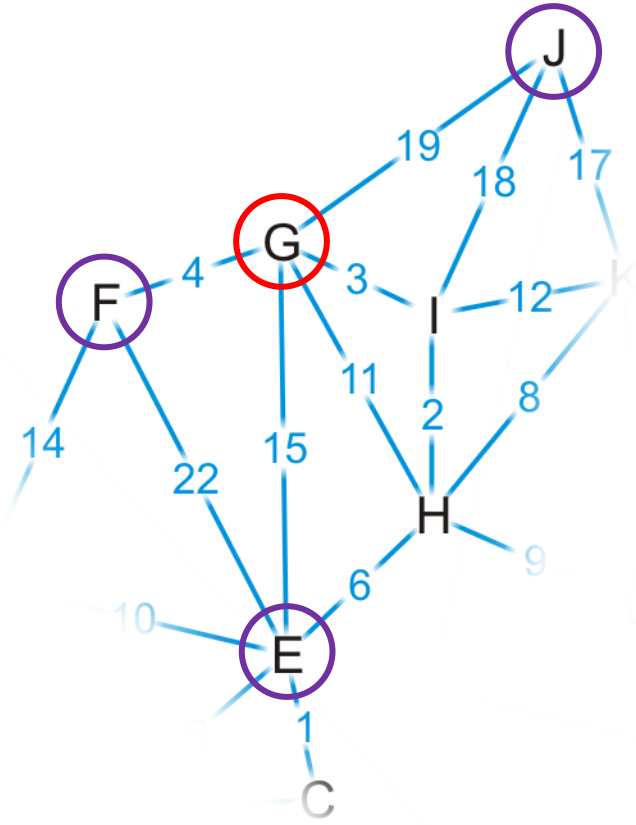
– Which do we update?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

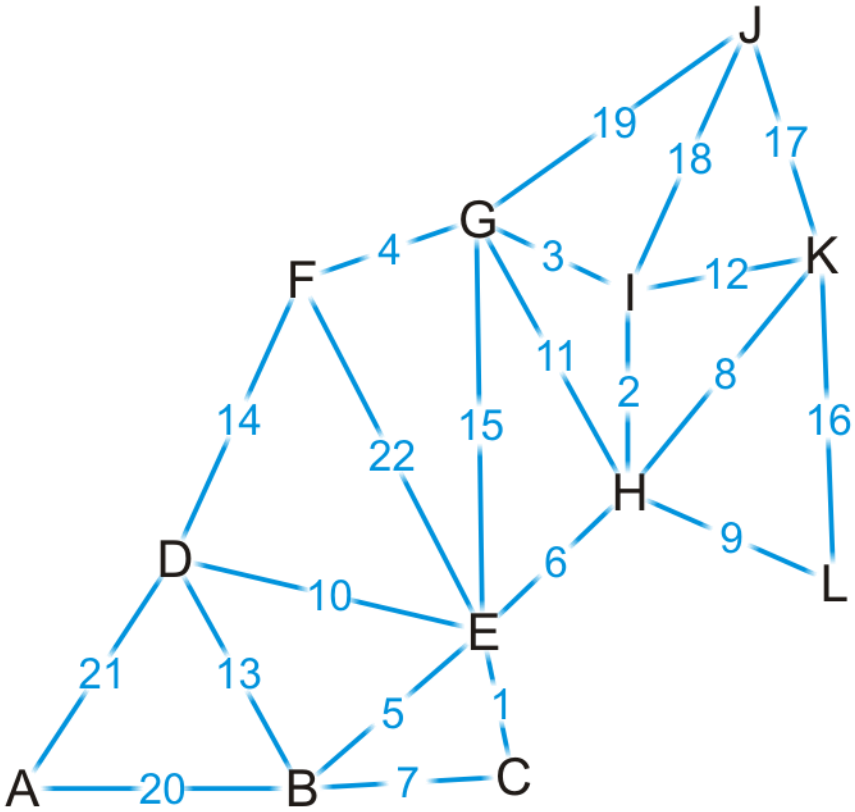
We have now found a path to vertex F



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	**17**	**G**
G	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Where do we visit next?

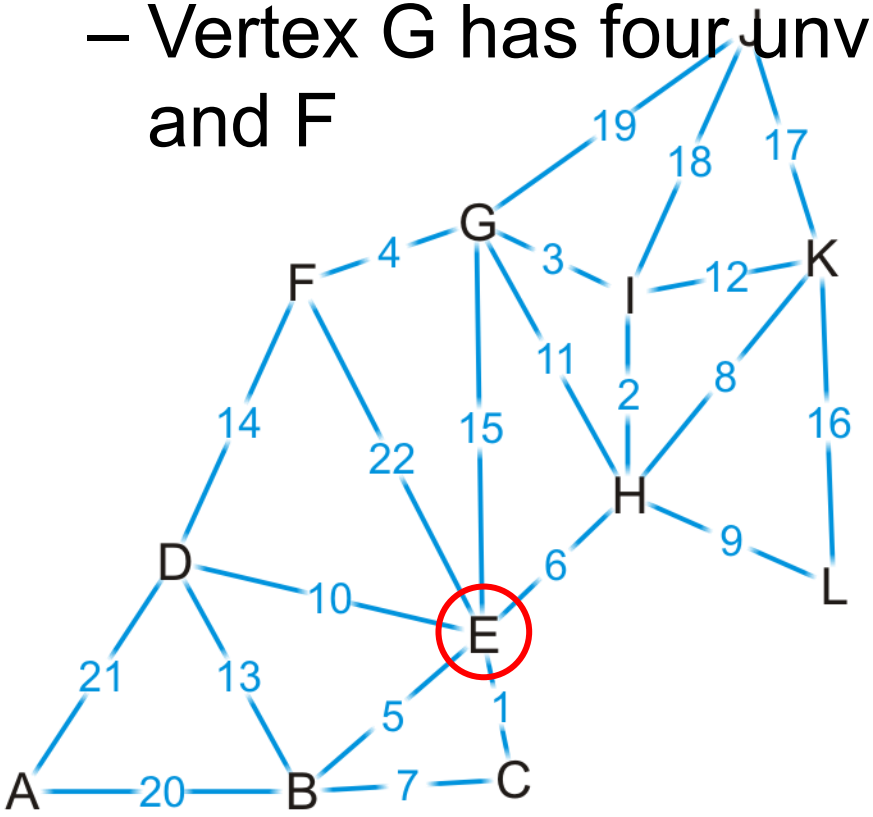


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, E) is the shortest path from K to E of length 14

– Vertex G has four unvisited neighbors: B, C, D and F

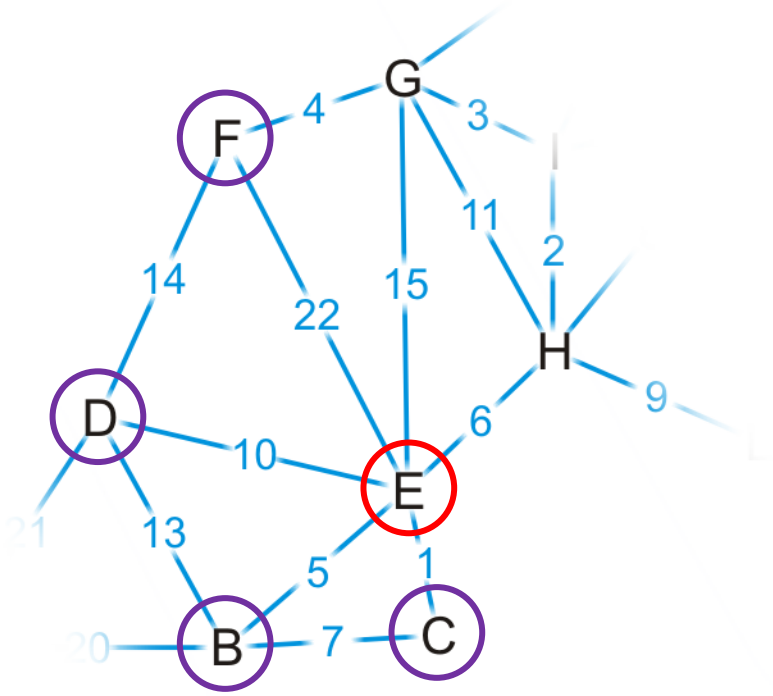


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, E) is the shortest path from K to E of length 14

– Vertex G has four unvisited neighbors: B, C, D and F



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

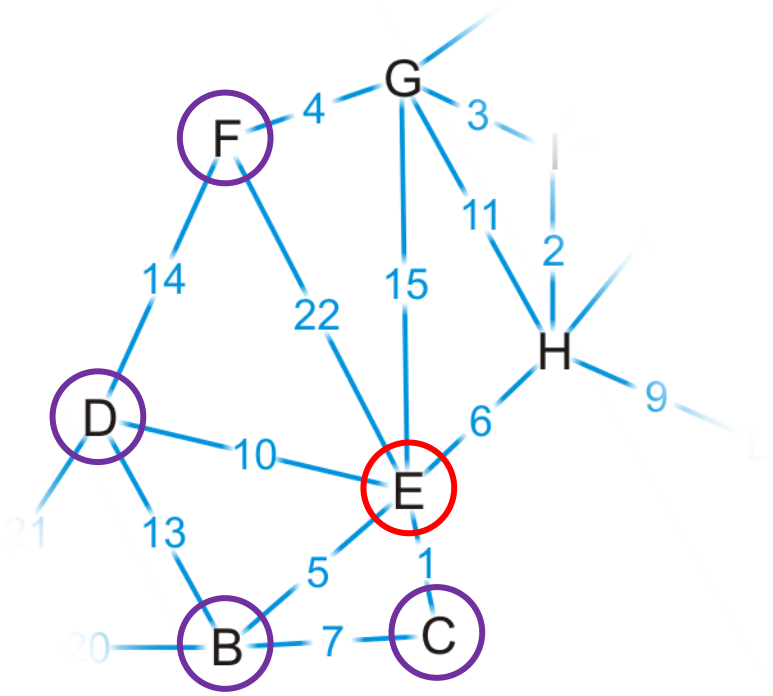
(K, H, E, B) of length 14 + 5 = 19

(K, H, E, C) of length 14 + 1 = 15

(K, H, E, D) of length 14 + 10 = 24

(K, H, E, F) of length 14 + 22 = 36

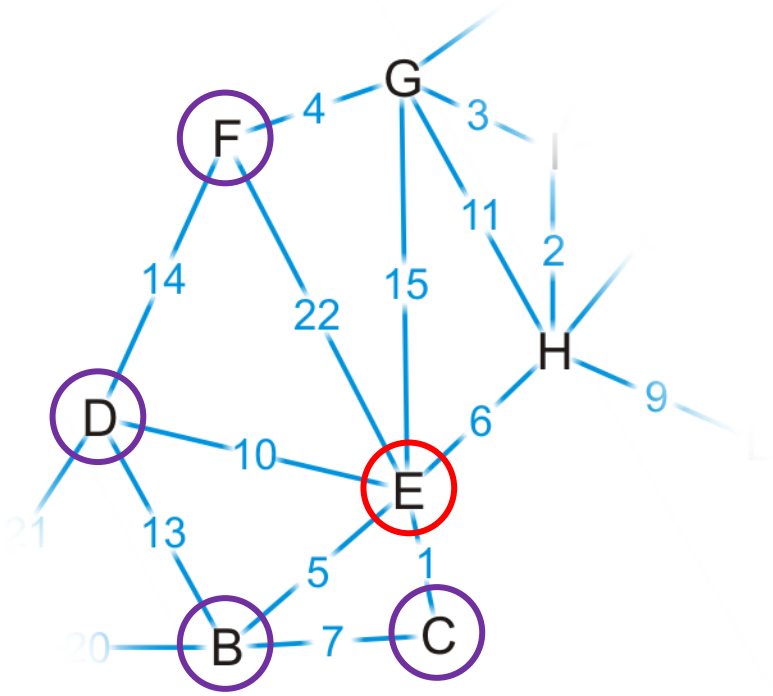
– Which do we update?



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

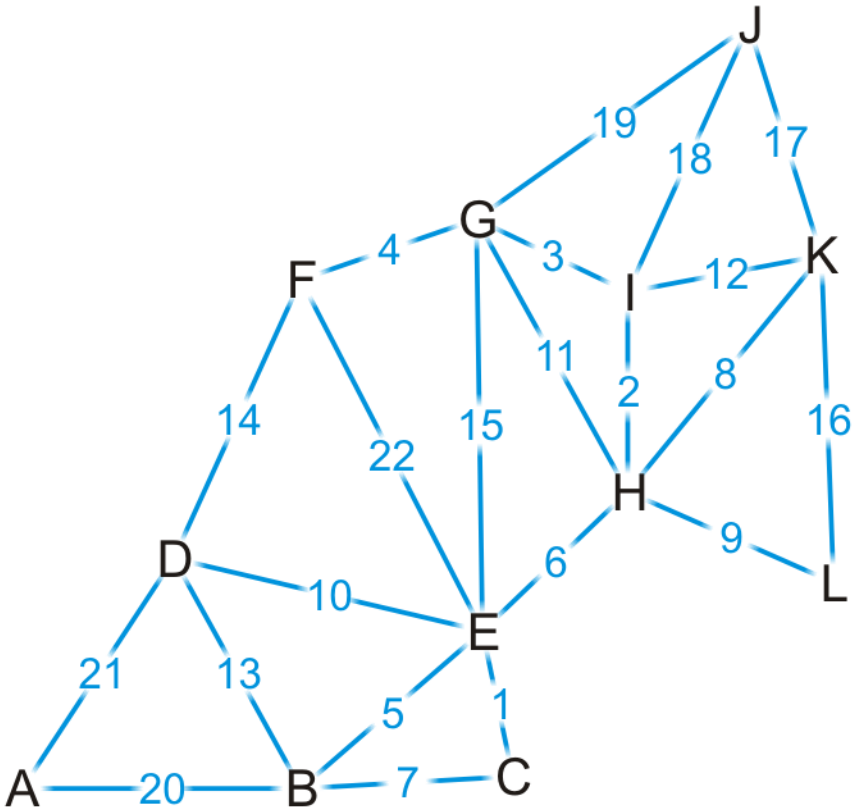
We've discovered paths to vertices B, C, D



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	**19**	**E**
C	F	**15**	**E**
D	F	**24**	**E**
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Which vertex is next?

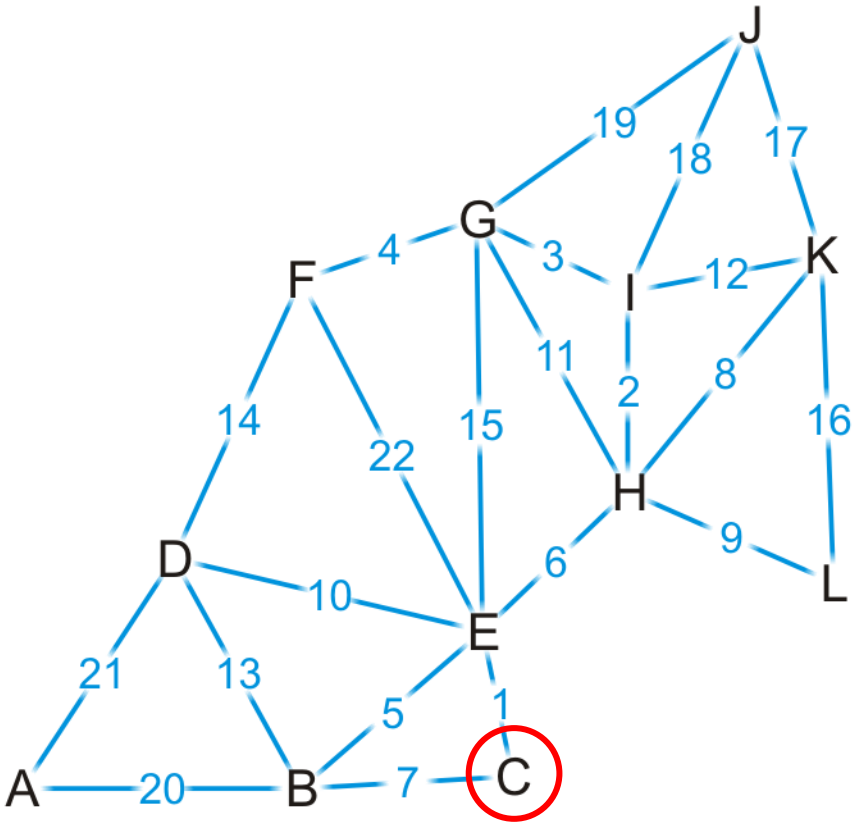


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	F	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We've found that the path (K, H, E, C) of length 15 is the shortest path from K to C

– Vertex C has one unvisited neighbor, B

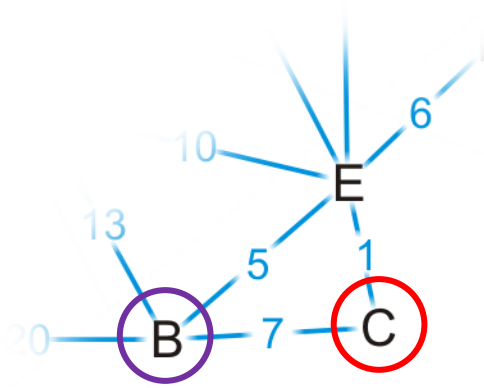


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

The path (K, H, E, C, B) is of length 15 + 7 = 22

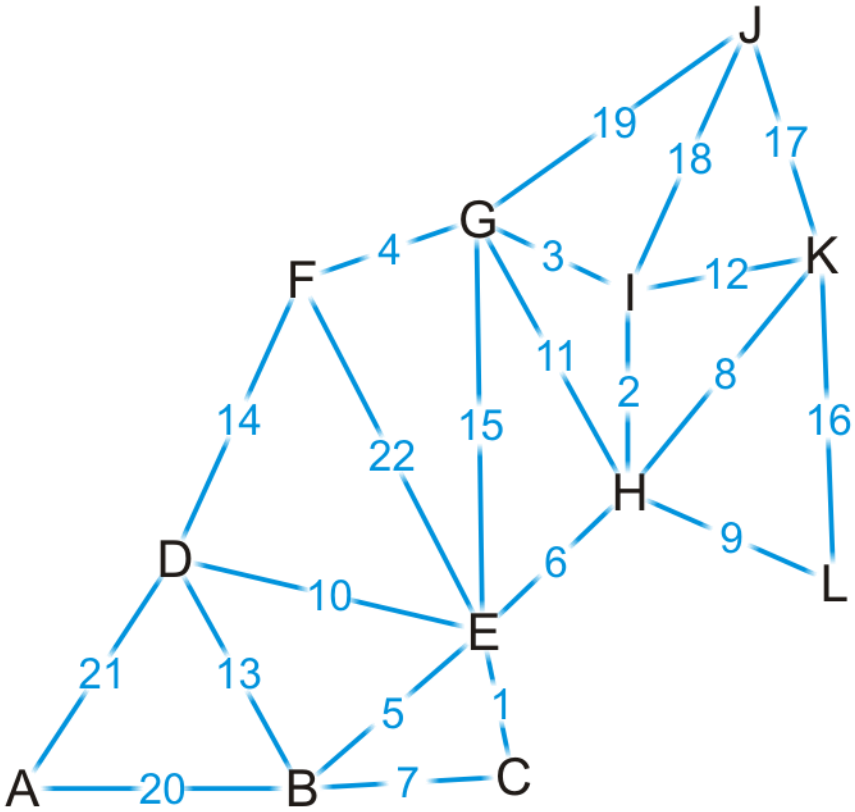
– We have already discovered a shorter path through vertex E



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Where to next?

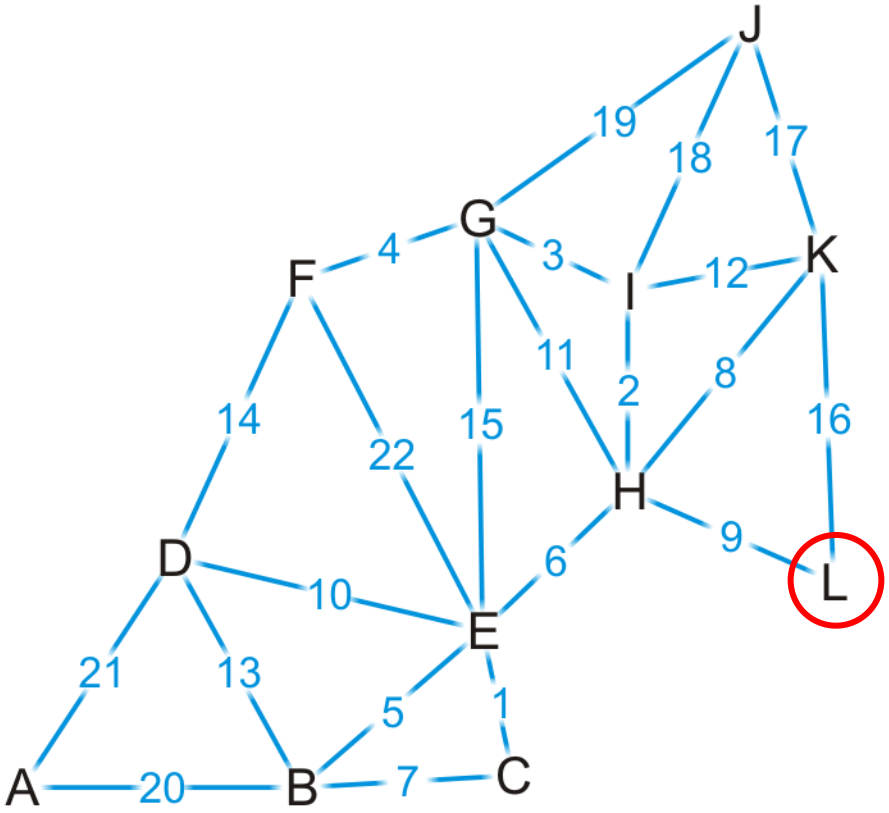


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We now know that (K, L) is the shortest path between these two points

– Vertex L has no unvisited neighbors

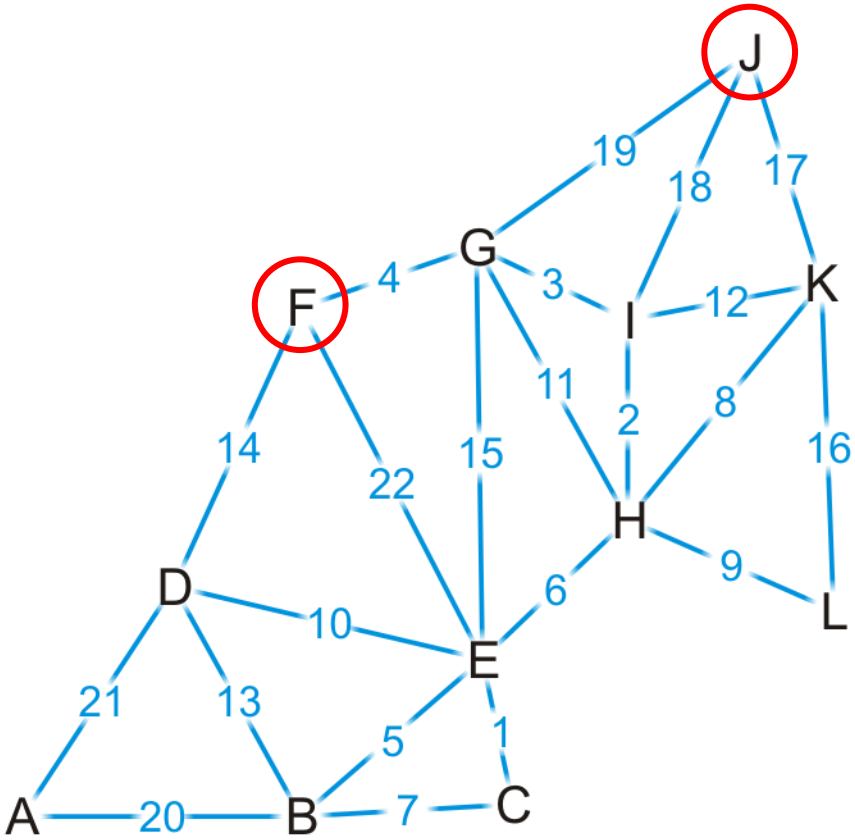


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	**T**	**16**	**K**

Example

Where to next?

– Does it matter if we visit vertex F first or vertex J first?

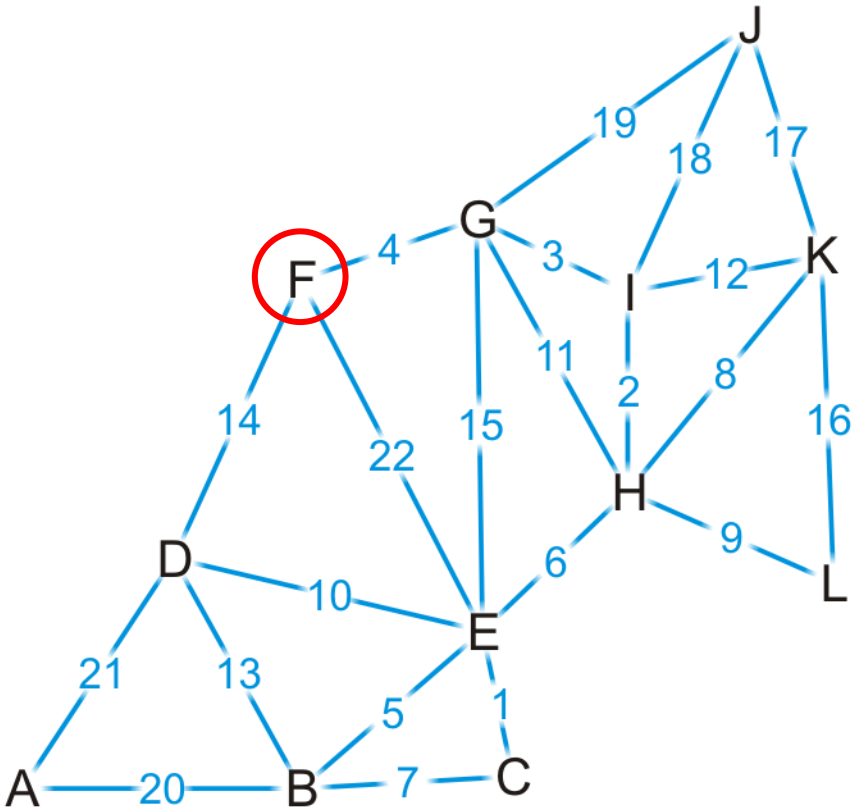


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Let's visit vertex F first

– It has one unvisited neighbor, vertex D

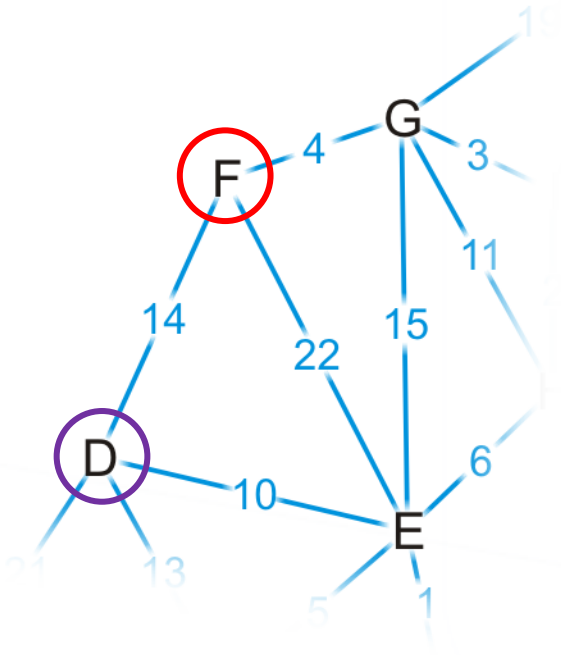


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

The path (K, H, I, G, F, D) is of length 17 + 14 = 31

– This is longer than the path we've already discovered

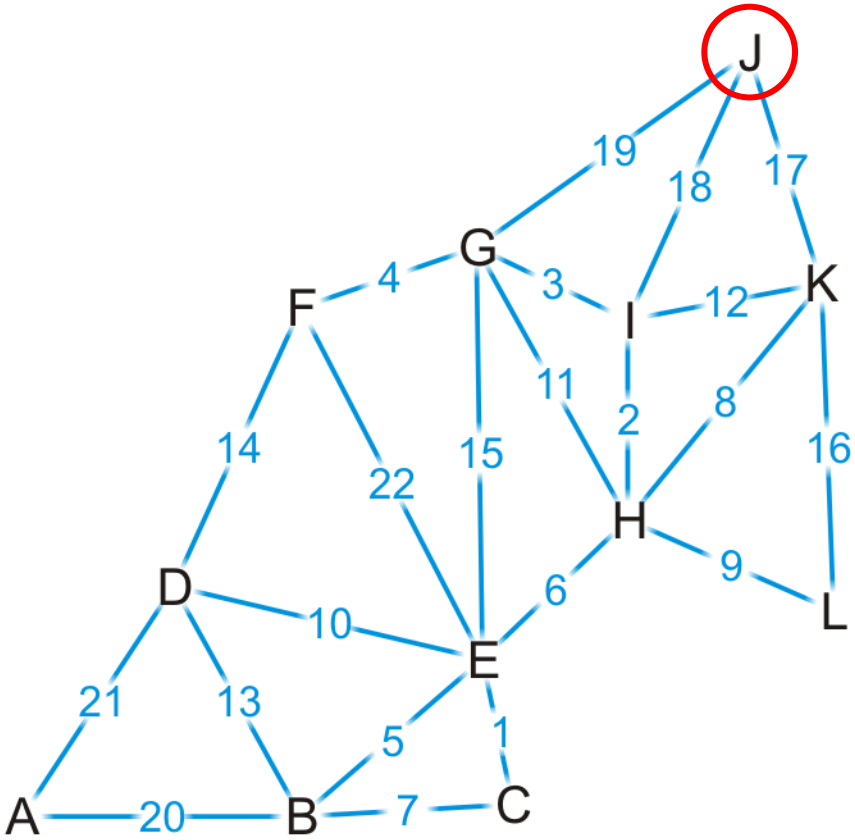


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Now we visit vertex J

– It has no unvisited neighbors



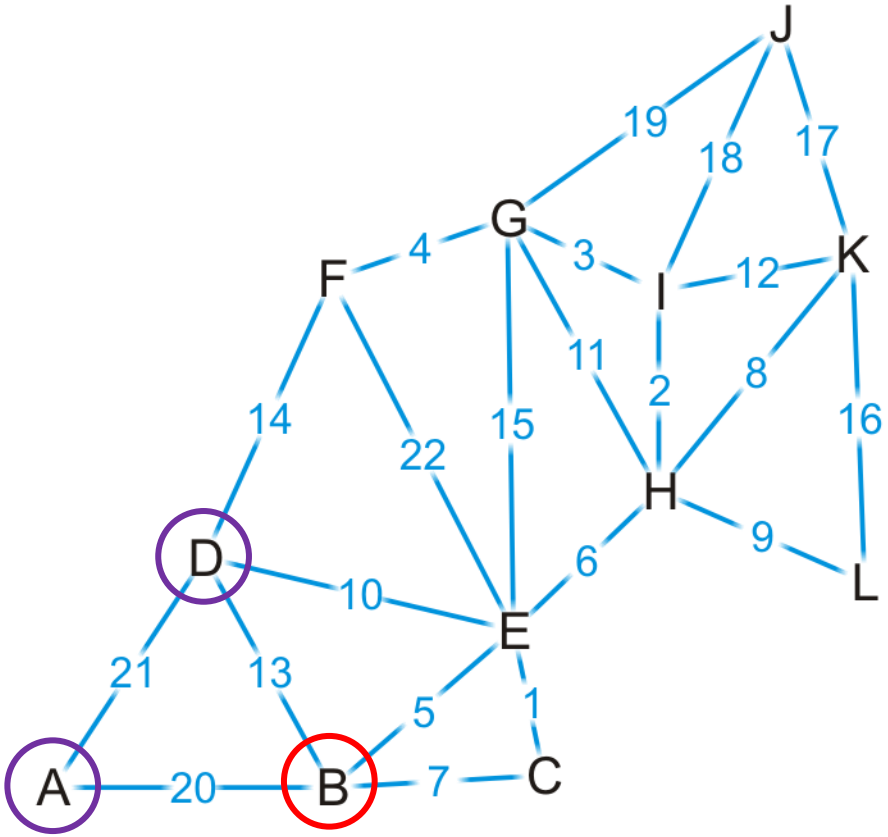
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	**T**	**17**	**K**
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex B, which has two unvisited neighbors:

(K, H, E, B, A) of length 19 + 20 = 39 (K, H, E, B, D) of length 19 + 13 = 32

– We update the path length to A

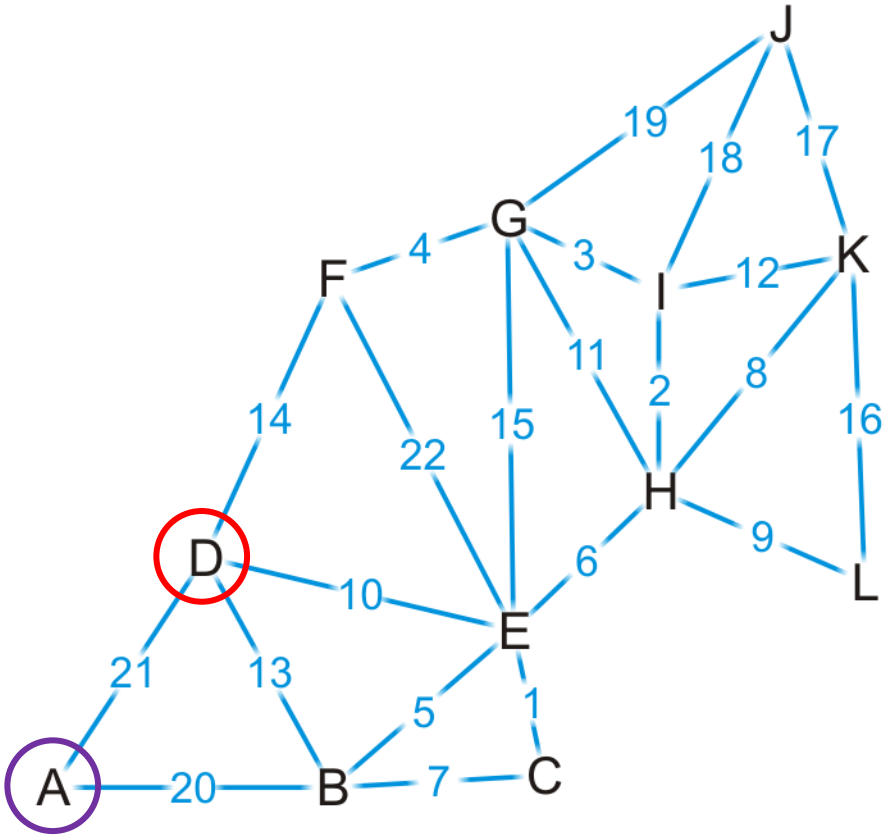


Vertex	Visited	Distance	Previous
A	F	**39**	**B**
B	**T**	**19**	**E**
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex D

- The path (K, H, E, D, A) is of length 24 + 21 = 45
- We don't update A

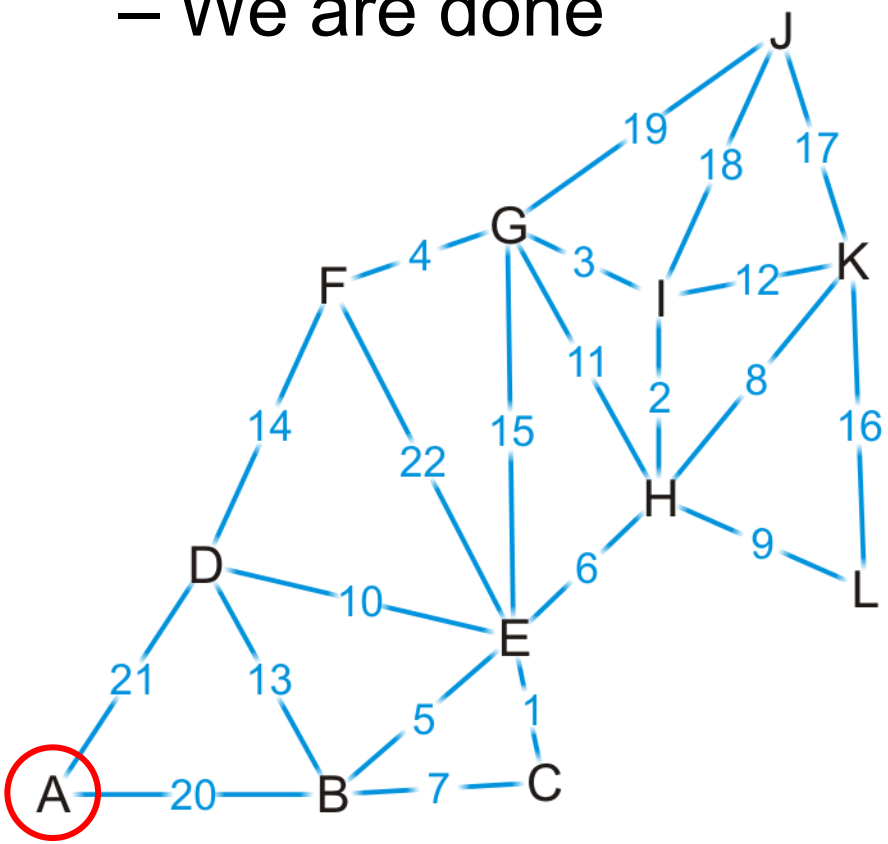


Vertex	Visited	Distance	Previous
A	F	39	B
B	T	19	E
C	T	15	E
D	**T**	**24**	**E**
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Finally, we visit vertex A

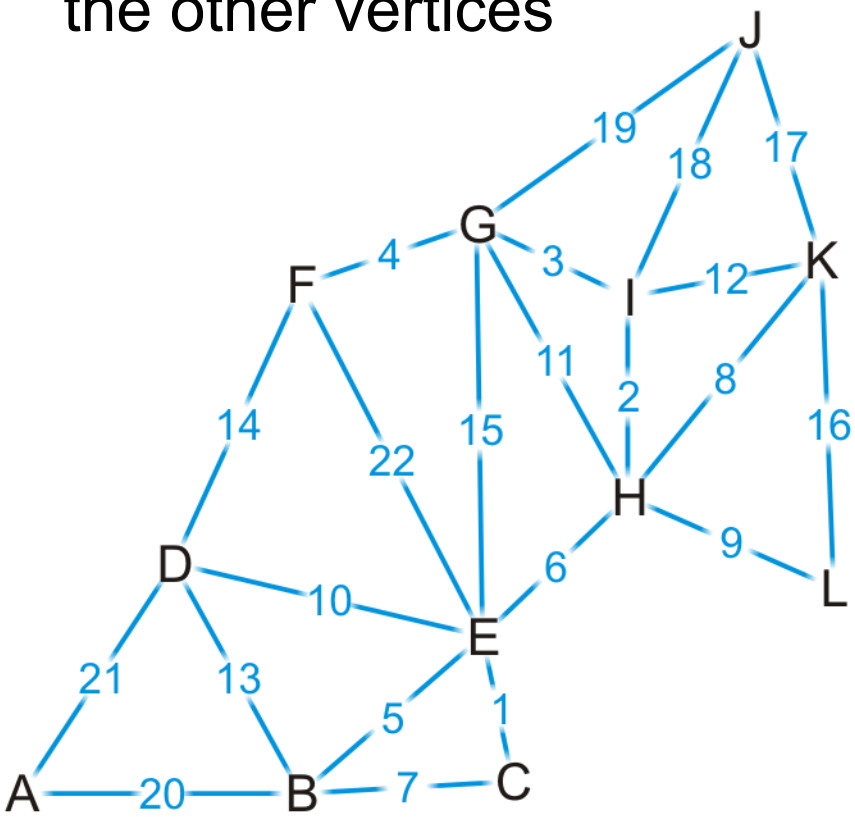
- It has no unvisited neighbors and there are no unvisited vertices left
- We are done



Vertex	Visited	Distance	Previous
A	**T**	**39**	**B**
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

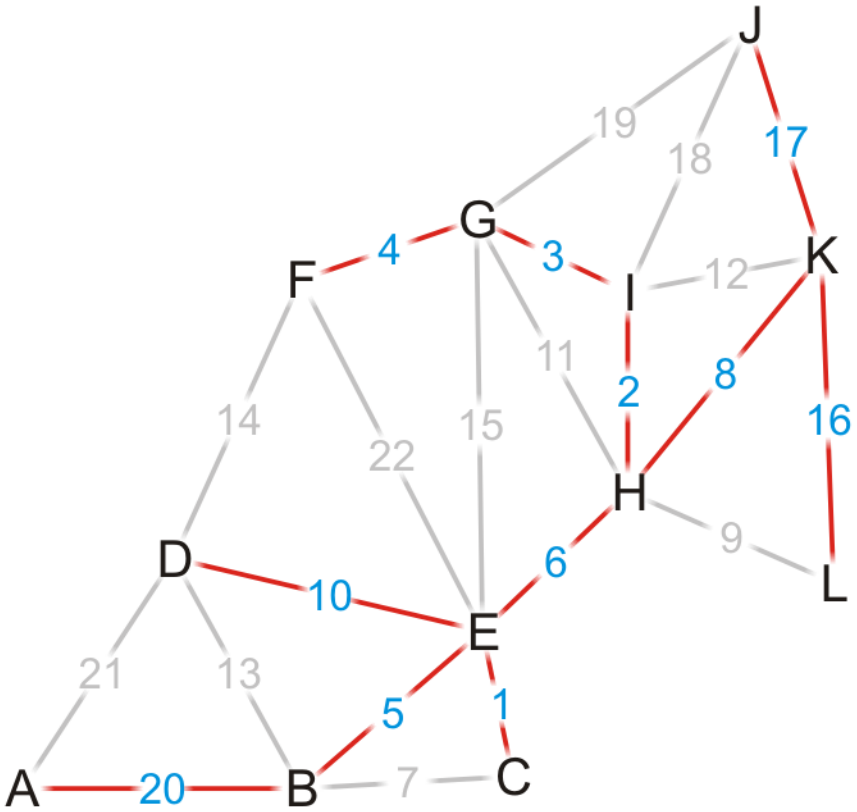
Thus, we have found the shortest path from vertex K to each of
the other vertices



Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Using the *previous* pointers, we can reconstruct the paths

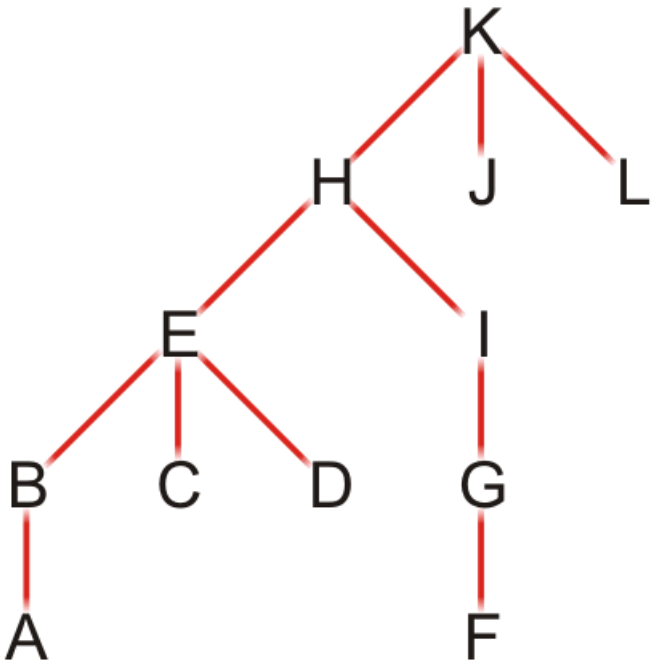


Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Note that this table defines a rooted parental tree

- The source vertex K is at the root
- The previous pointer is the *parent* of the vertex in the tree



Vertex	Previous
A	B
B	E
C	E
D	E
E	H
F	G
G	I
H	K
I	H
J	K
K	Ø
L	K

Comments on Dijkstra's algorithm

Questions:

- What if at some point, all unvisited vertices have a distance ∞?
  - This means that the graph is unconnected
  - We have found the shortest paths to all vertices in the connected subgraph containing the source vertex
- What if we just want to find the shortest path between vertices v_j and v_k?
  - Apply the same algorithm, but stop when we are visiting vertex v_k
- Does the algorithm change if we have a directed graph?
  - No

Implementation and analysis

The initialization requires $\Theta(|V|)$ memory and run time

We iterate $|V| - 1$ times, each time finding next closest vertex to the source

- Iterating through the table requires is $\Theta(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors
- With an adjacency matrix, the run time is $\Theta(|V|(|V| + |V|)) = \Theta(|V|^2)$
- With an adjacency list, the run time is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ as $|E| = O(|V|^2)$

Can we do better?

- Recall, we only need the closest vertex
- How about a priority queue?

Pseudocode (Using PQ)

```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;  // can use Integer.MAX_VALUE
    queue.enqueue(v);
    v.pred = null;
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

```
void relax(u, v) {
  if (u.dist + w(u,v) < v.dist) {
    v.dist = u.dist + w(u,v);
    v.pred = u;
  }
}
```

Source: http://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html

Implementation and analysis

The initialization still requires $\Theta(|V|)$ memory and run time
- The priority queue will also requires $O(|V|)$ memory
- We must use an adjacency list, not an adjacency matrix

We iterate $|V|$ times, each time finding the *closest* vertex to the source
- Place the distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Thus, the work required for this is $O(|V| \ln(|V|))$

Is this all the work that is necessary?
- Recall that each edge visited may result in updating the priority queue
- Thus, the work required for this is $O(|E| \ln(|V|))$

Thus, the total run time is $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$

Summary

We have seen an algorithm for finding single-source shortest paths

- Start with the initial vertex
- Continue finding the next vertex that is closest

Dijkstra's algorithm always finds the next closest vertex

- It solves the problem in $O(|E| \ln(|V|))$ time using Priority Queue
- The algorithm can further improved to run in time $O((|V|\ln(|V|) + |E|)$ --- (using Fibonacci Heap; We will discuss this later if time permits)

Acknowledgement

- Douglas Wilhelm Harder.
  - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course

- Prof. Hung Q. Ngo:
  - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.

- Many of these slides are taken from these two sources.