CS101 Algorithms and Data Structures

Shortest Path-Dijkstra’s & Bellman-Ford

Textbook Ch 24, 25

Outline

- Definition and applications
- Dijkstra’s algorithm
- Bellman-Ford algorithm

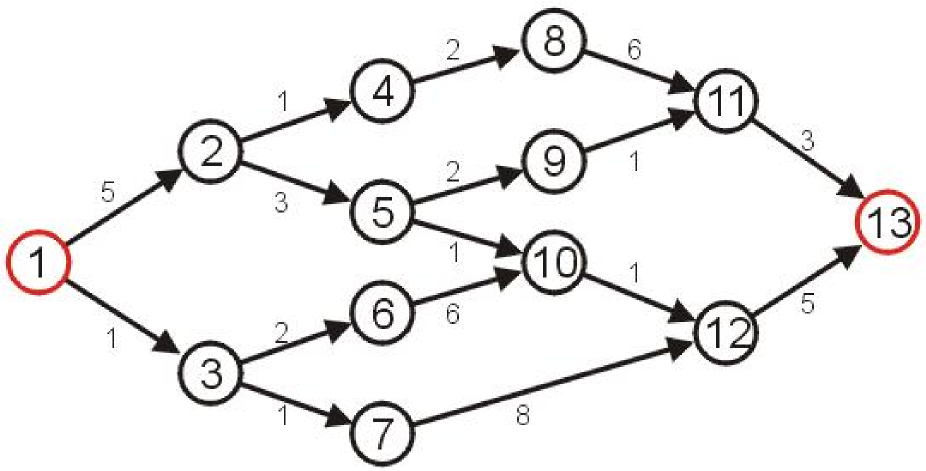
Shortest Path

Given a weighted directed graph, one common problem is finding the shortest path between two given vertices

- Recall that in a weighted graph, the *length* of a path is the sum of the weights of each of the edges in that path

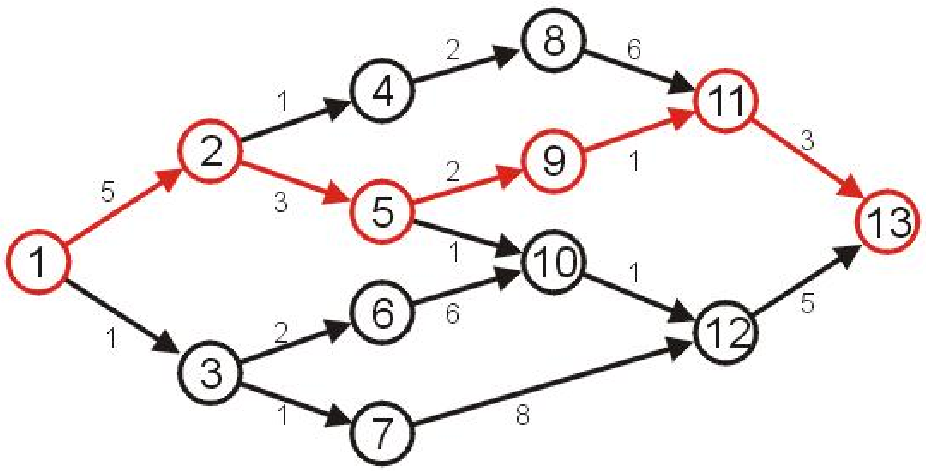
Shortest Path

Given the graph, suppose we wish to find the shortest path from vertex 1 to vertex 13



Shortest Path

After some consideration, we may determine that the shortest path is as follows, with length 14



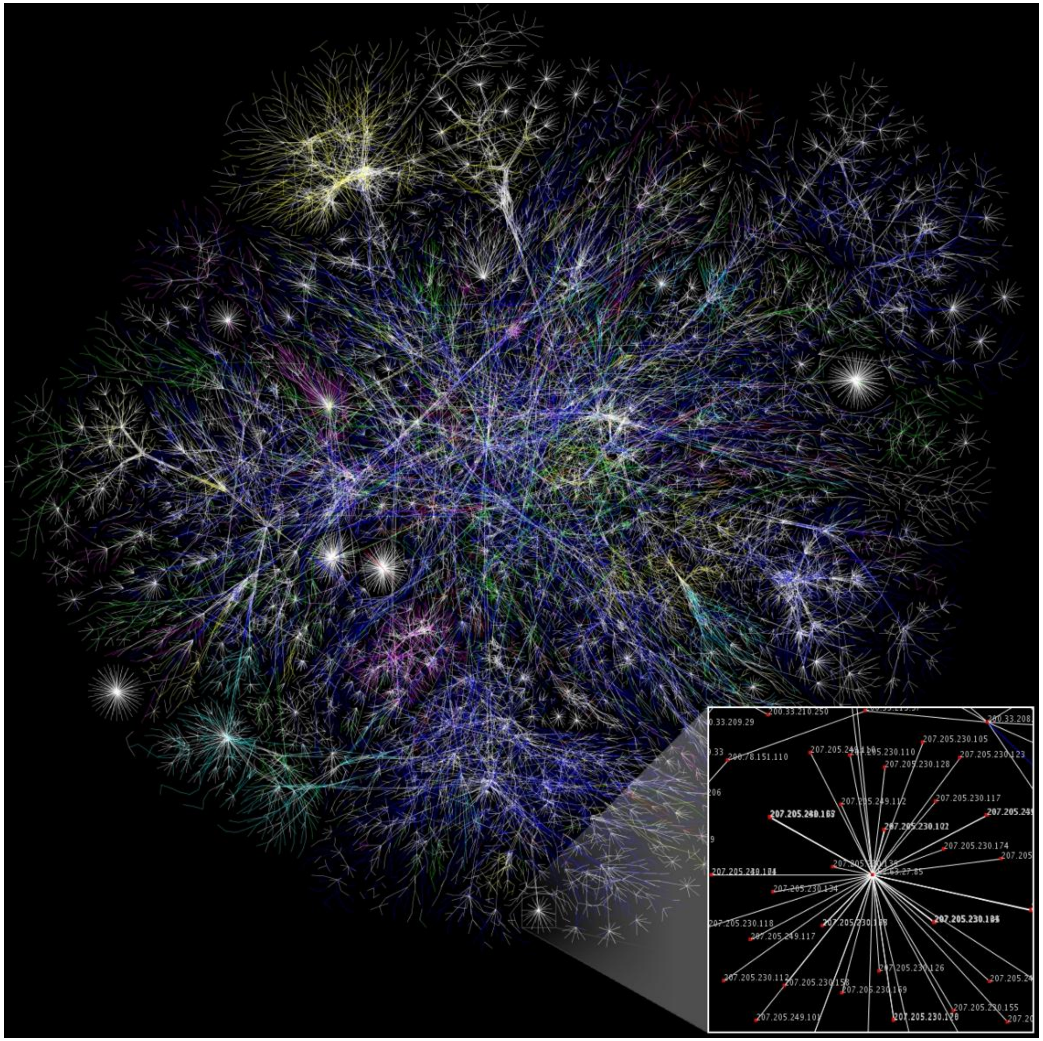
Other paths exists, but they are longer

Applications

The Internet is a collection of interconnected devices

- Routers, individual computers

These may be represented as graphs



Applications

Information is passed through *packets.*

Packets are passed from the source, through routers, to their destination.

We would like to pass packets through the shortest path.

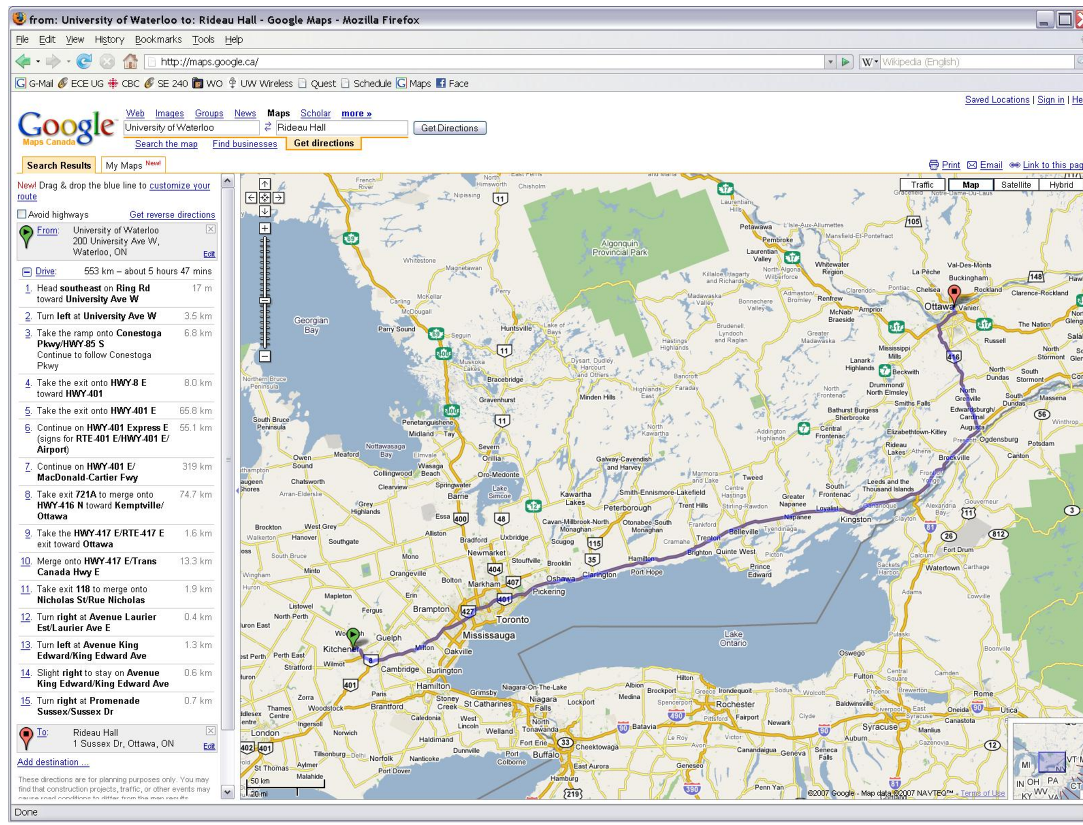
Metrics for measuring the shortest path may include:

- low latency (minimize time), or
- minimum hop count (all edges have weight 1)

Applications

Another obvious application is finding the shortest route between two points on a map

The shortest path using distance as a metric is obvious, however, a driver may be more interested in minimizing time



Applications

A company will be interested in minimizing the cost which includes the following factors:

- salary of the truck driver (overtime?)
- possible tolls and administrative costs
- bonuses for being early
- penalties for being late
- cost of fuel

Applications

Very quickly, the definition of the *shortest path* becomes time-dependant:

- rush hour
- long weekends

and situation dependant:

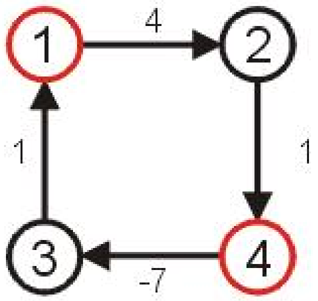
- scheduled events (*e*.*g*., road construction)
- unscheduled events (*e*.*g*., accidents)

Shortest Path

The goal is to find the shortest path and its length

We will make the assumption that the weights on all edges is a positive number

- Why this assumption?
- If we have negative weights, it may be possible to end up in a cycle whereby each pass through the cycle decreases the total *length*
- Thus, a shortest length would be undefined for such a graph
- Consider the shortest path from vertex 1 to 4...



Algorithms

Algorithms for finding the shortest path include:

- Dijkstra’s algorithm
- A* search algorithm
- Bellman-Ford algorithm
- Floyd-Warshall algorithm

Outline

- Definition and applications
- Dijkstra’s algorithm
- Bellman-Ford algorithm

Dijkstra’s algorithm

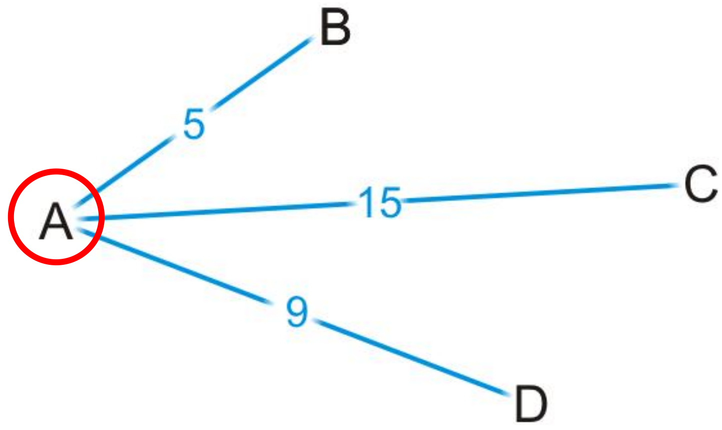
Dijkstra’s algorithm solves the single-source shortest path problem

- It is very similar to Prim’s algorithm
- Assumption: all the weights are positive

Strategy

Suppose you are at vertex A

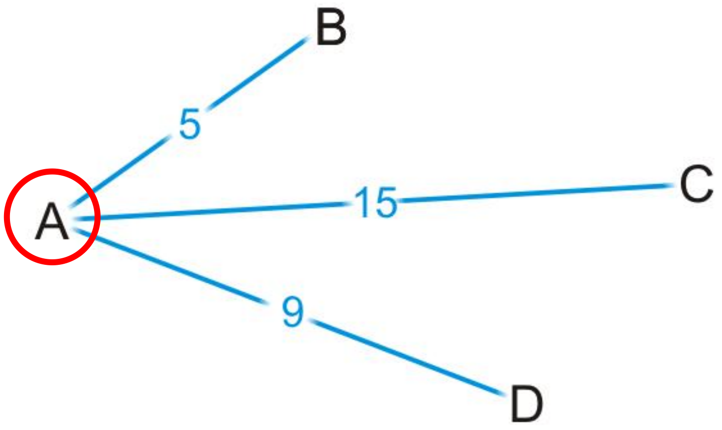
- You are aware of all vertices adjacent to it
- This information is either in an adjacency list or adjacency matrix



Strategy

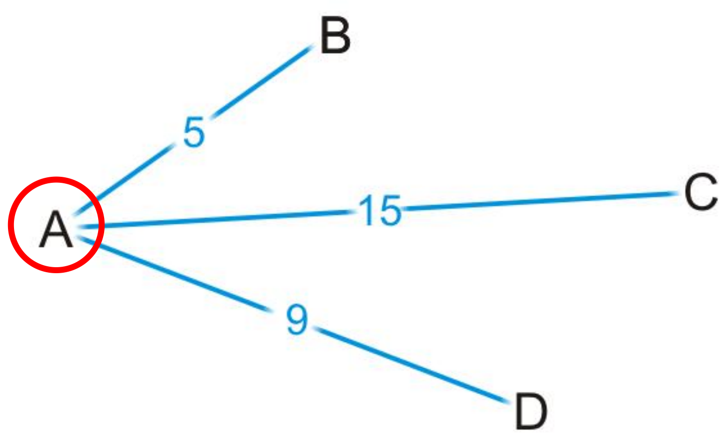
Is 5 the shortest distance to B via the edge (A, B)?

– Why or why not?



Strategy

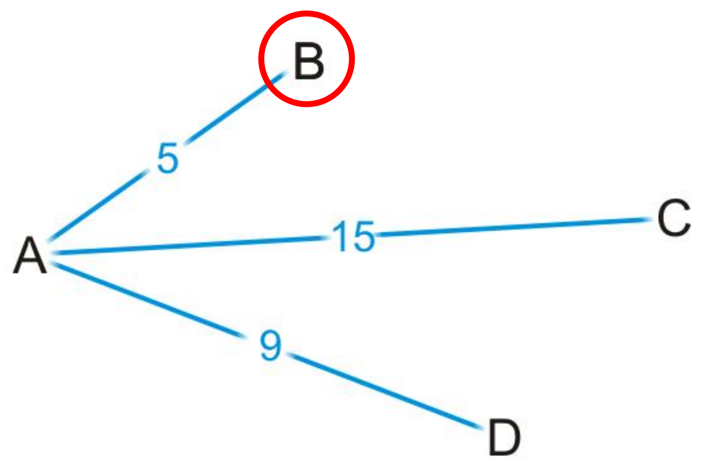
Are you guaranteed that the shortest path to C is (A, C), or that (A, D) is the shortest path to vertex D?



Strategy

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

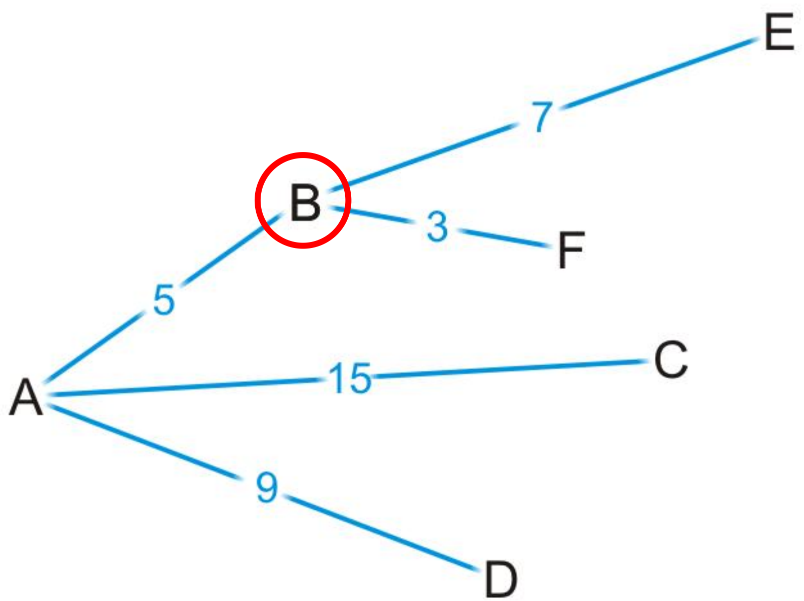
- Let's see where we can go from B



Strategy

By some simple arithmetic, we can determine that

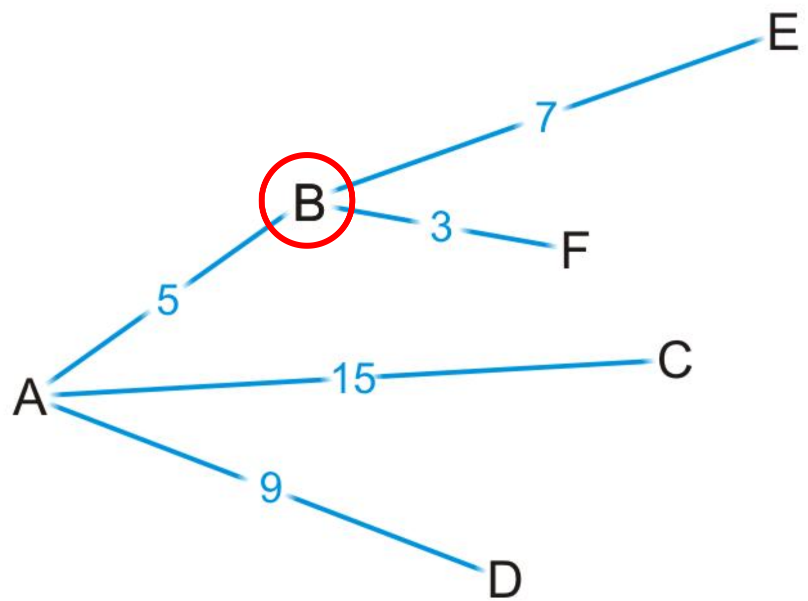
- There is a path (A, B, E) of length 5 + 7 = 12
- There is a path (A, B, F) of length 5 + 3 = 8



Strategy

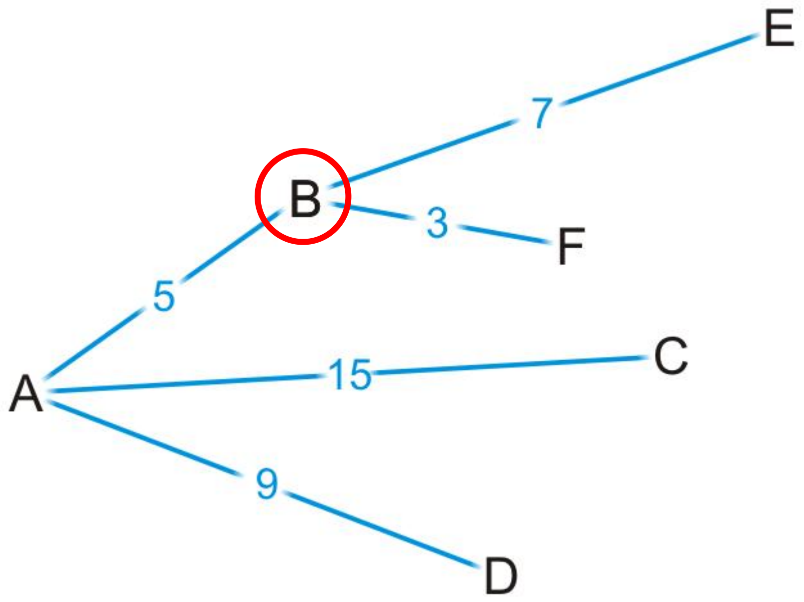
Is (A, B, F) is the shortest path from vertex A to F?

- Why or why not?



Strategy

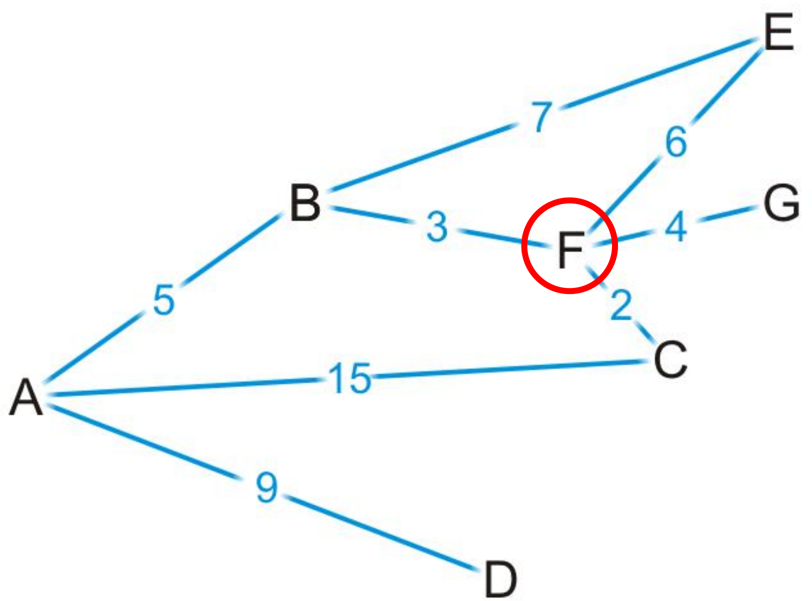
Are we guaranteed that any other path we are currently aware of is also going to be the shortest path?



Strategy

Okay, let's visit vertex F

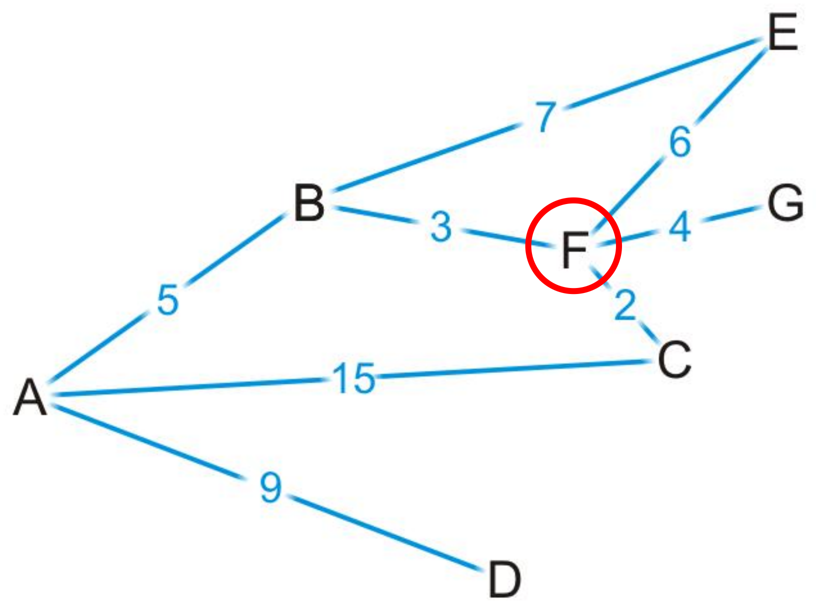
- We know the shortest path is (A, B, F) and it's of length 8



Strategy

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

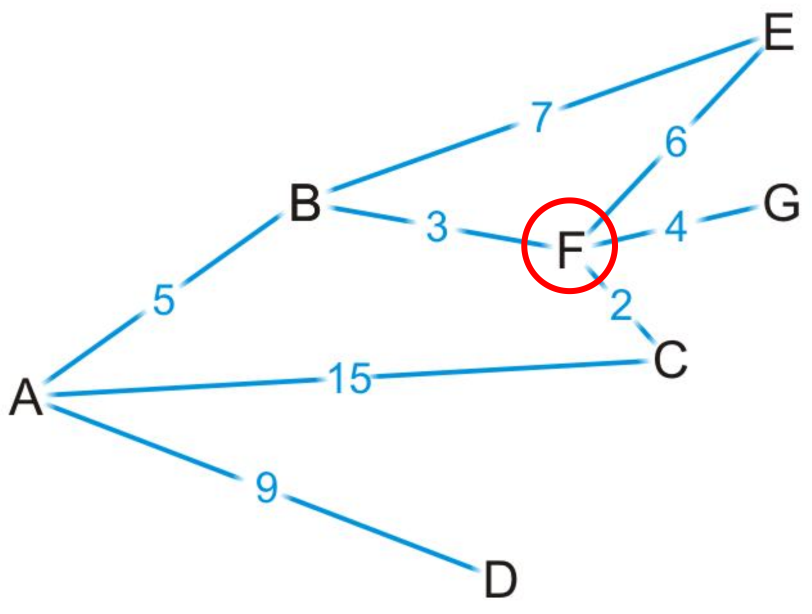
- (A, B, F, E) of length 8 + 6 = 14
- (A, B, F, G) of length 8 + 4 = 12
- (A, B, F, C) of length 8 + 2 = 10



Strategy

By observation:

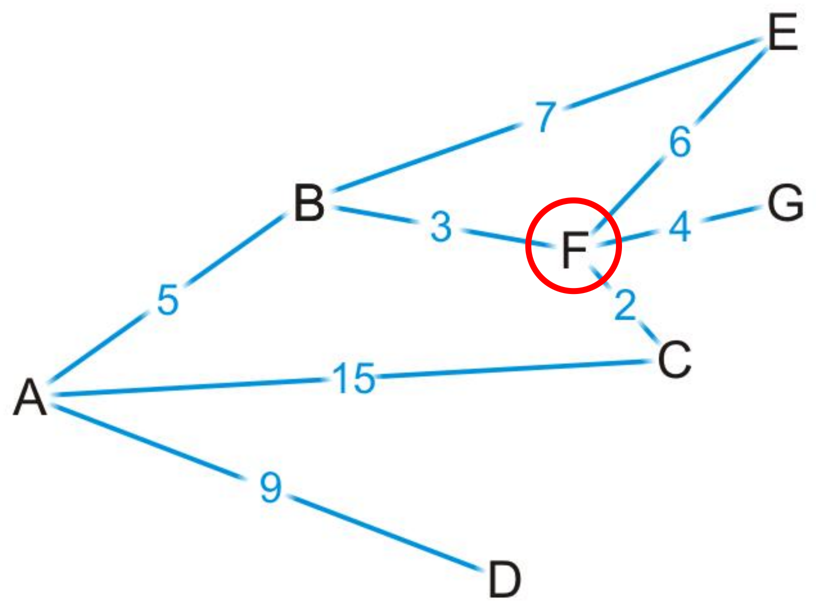
- The path (A, B, F, E) is longer than (A, B, E)
- The path (A, B, F, C) is shorter than the path (A, C)



Strategy

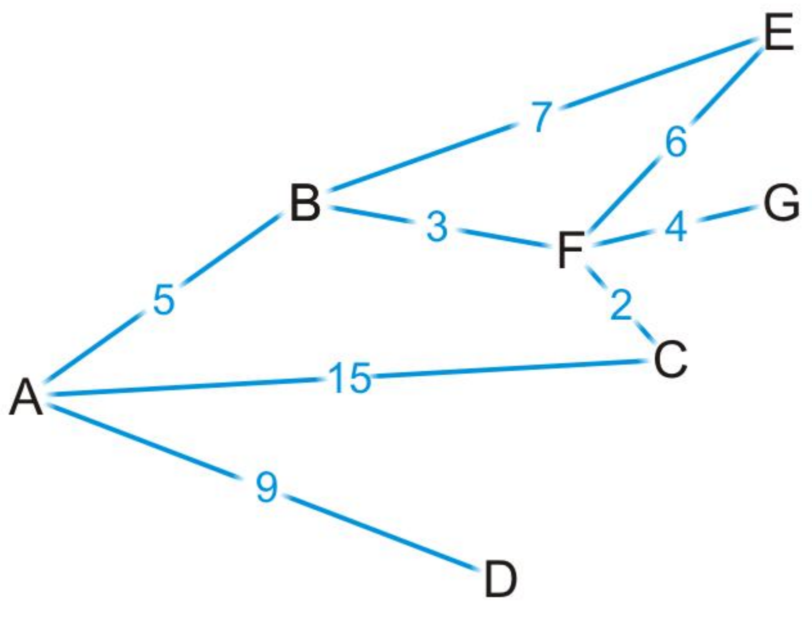
At this point, we've discovered the shortest paths to:

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



Strategy

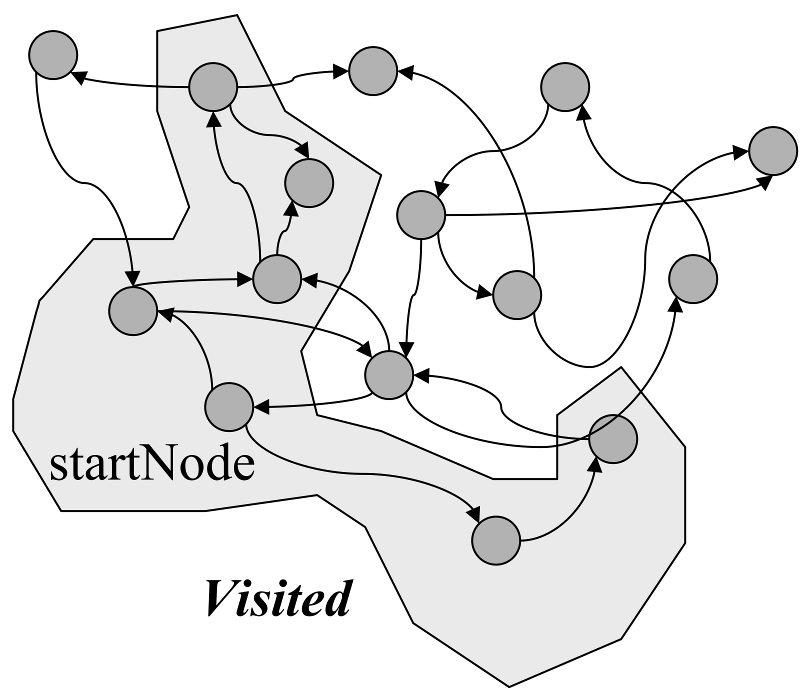
Which remaining vertex are we currently guaranteed to have the shortest distance to?



Strategy

In general

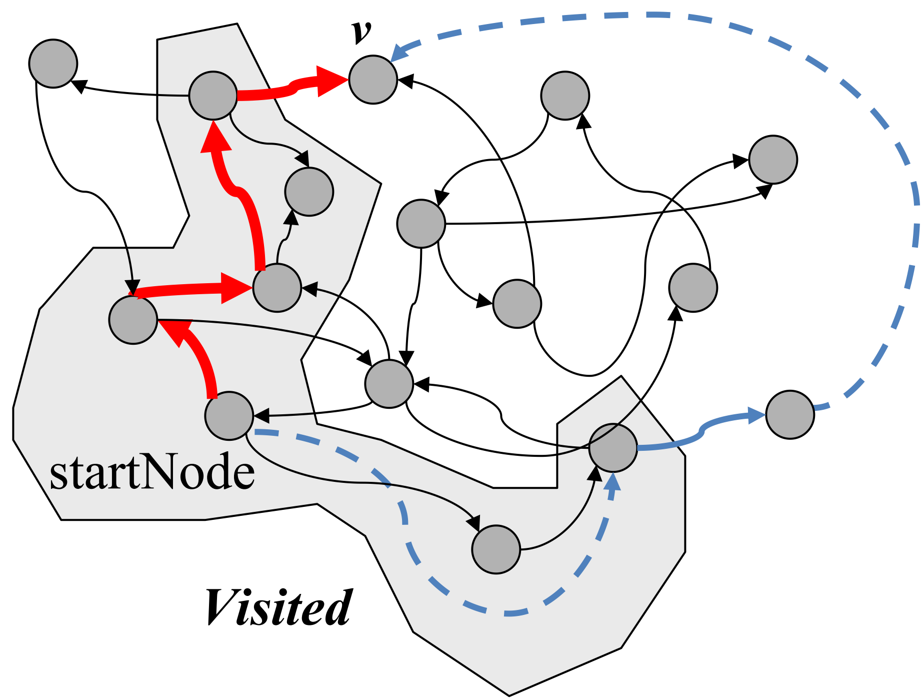
- We know the shortest distance to some of the vertices (marked as visited)
- We also know the shortest distance to each unvisited vertex **through visited vertices** (call this the “known distance”)



Strategy

Consider the unvisited vertex v that has the shortest known distance

- We are guaranteed that the known distance to v is the shortest distance from the start node to it
- Proof by contradiction



Dijkstra's algorithm

We need to track the known shortest distance to each vertex

- We require an array of distances, all initialized to infinity except for the source vertex, which is initialized to 0

Do we need to track the shortest path to each vertex?

- Ex: do I have to store (A, B, F) as the shortest path to vertex F?
- No. We only have to record that the shortest path to vertex F came from vertex B
  - The shortest path to F is the shortest path to B followed by the edge (B, F)
- Thus, we need an array of previous vertices, all initialized to null

We need to track visited vertices whose shortest paths have been found

- a Boolean table of size $|V|$

Dijkstra's algorithm

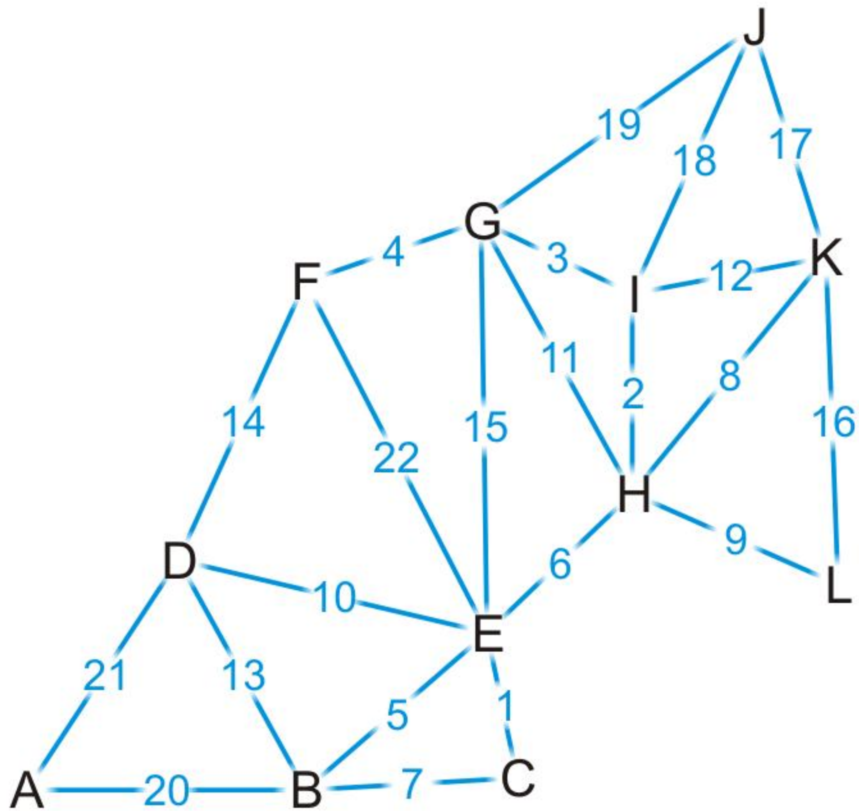
We will iterate $|V|$ times:

- Find the unvisited vertex v that has a minimum distance to it
- Mark it as visited
- Consider its 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 of infinity

Example

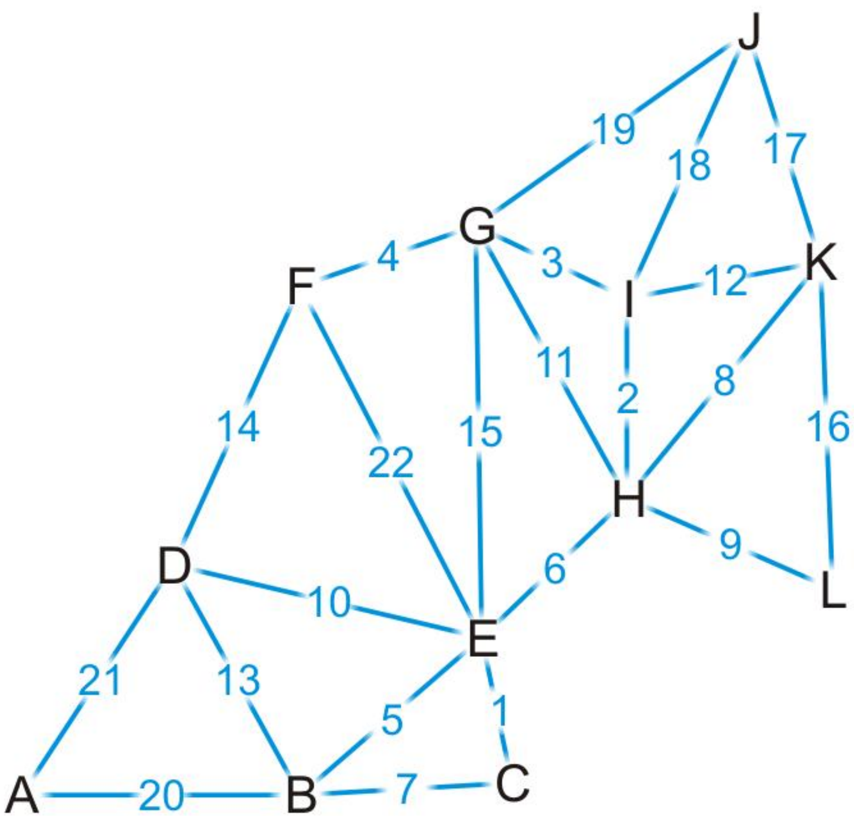
Find the shortest distance from K to every other vertex (BFS?)



Example

We set up our table

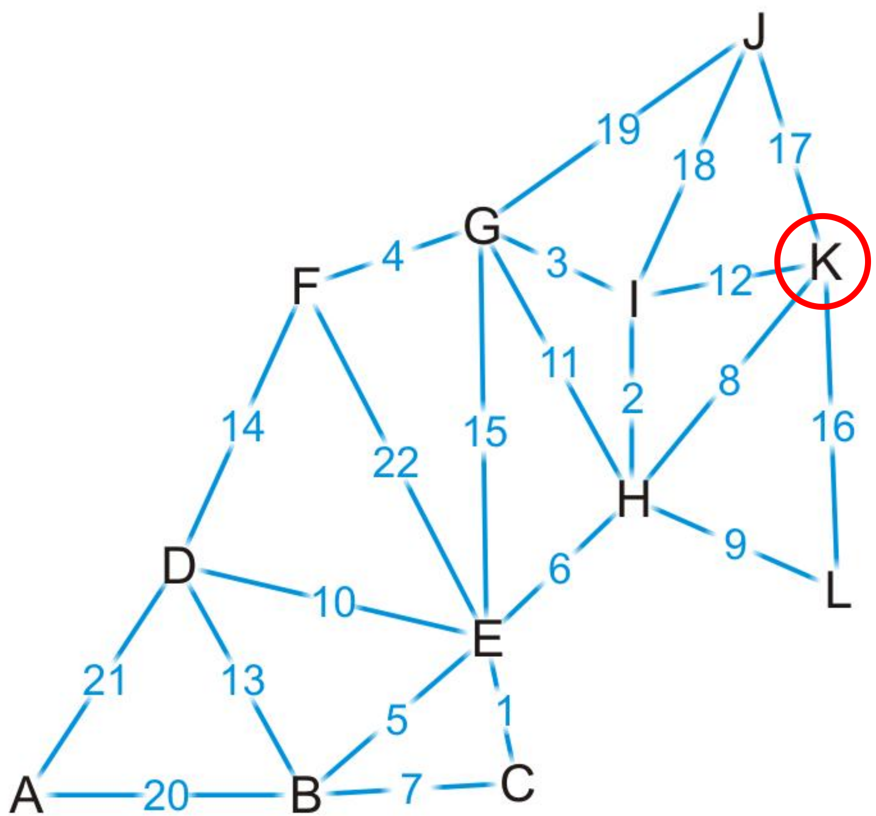
– Which unvisited vertex has the minimum distance to it?



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

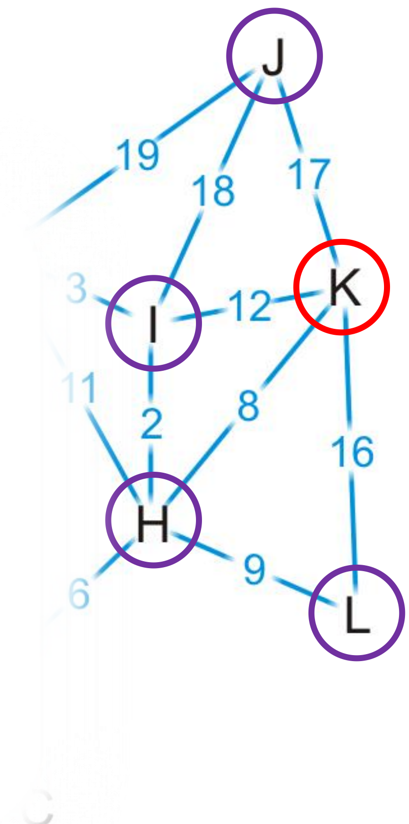
We visit vertex K



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

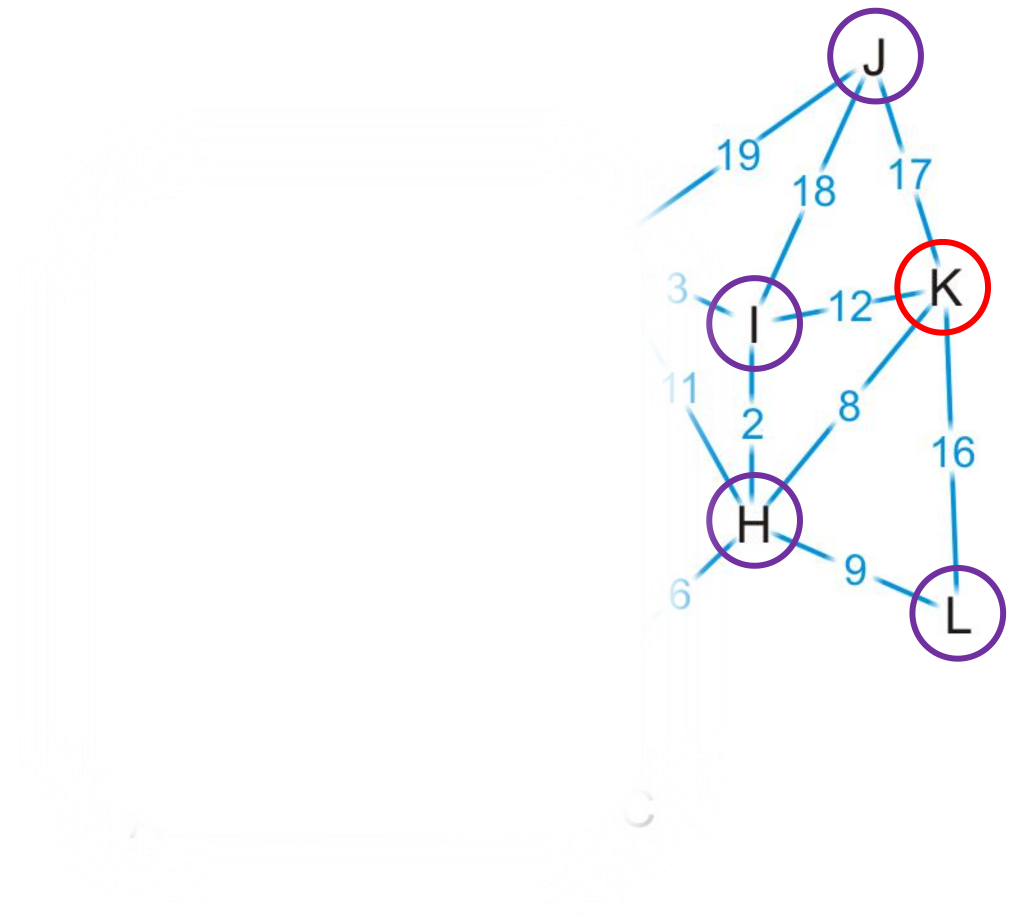
Vertex K has four neighbors: H, I, J and L



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

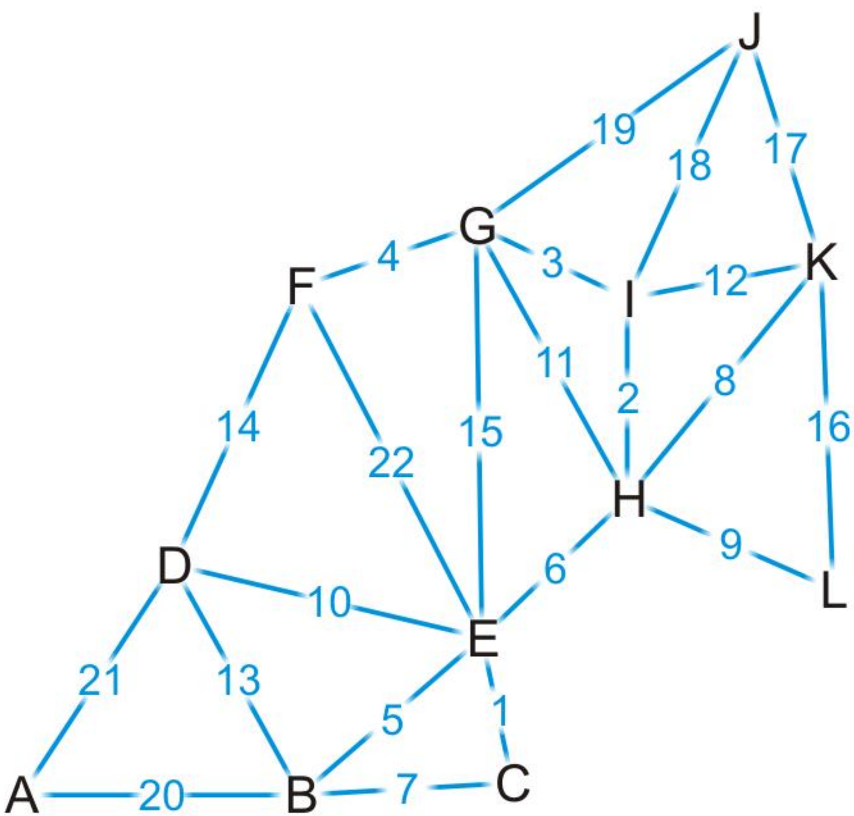


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	**8**	**K**
I	F	**12**	**K**
J	F	**17**	**K**
K	**T**	**0**	**Ø**
L	F	**16**	**K**

Example

We're finished with vertex K

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

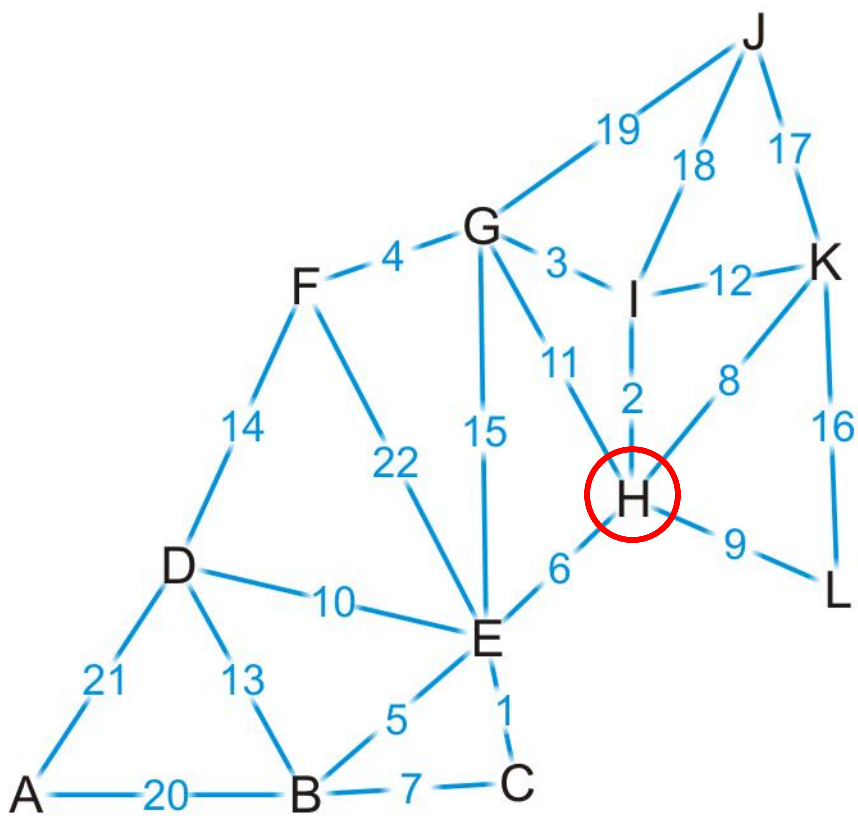


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

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:

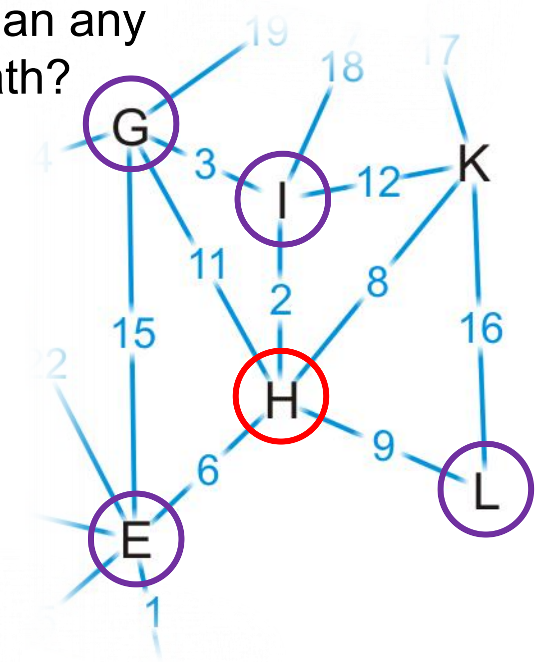
(K, H, E) of length 8 + 6 = 14

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

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

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

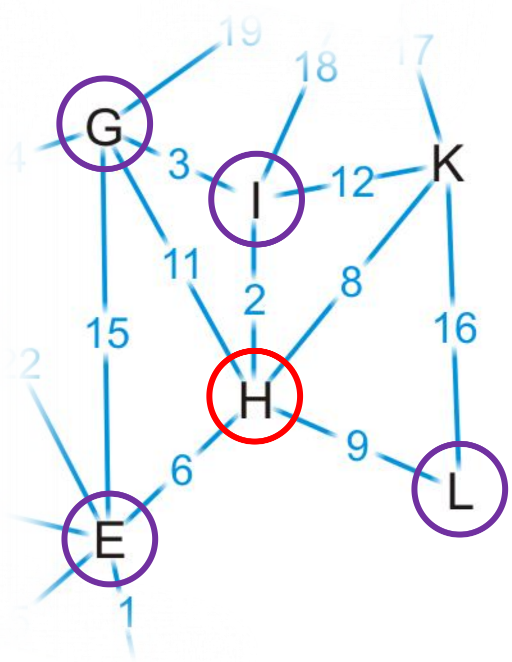
– Which of these are shorter than any known path?



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

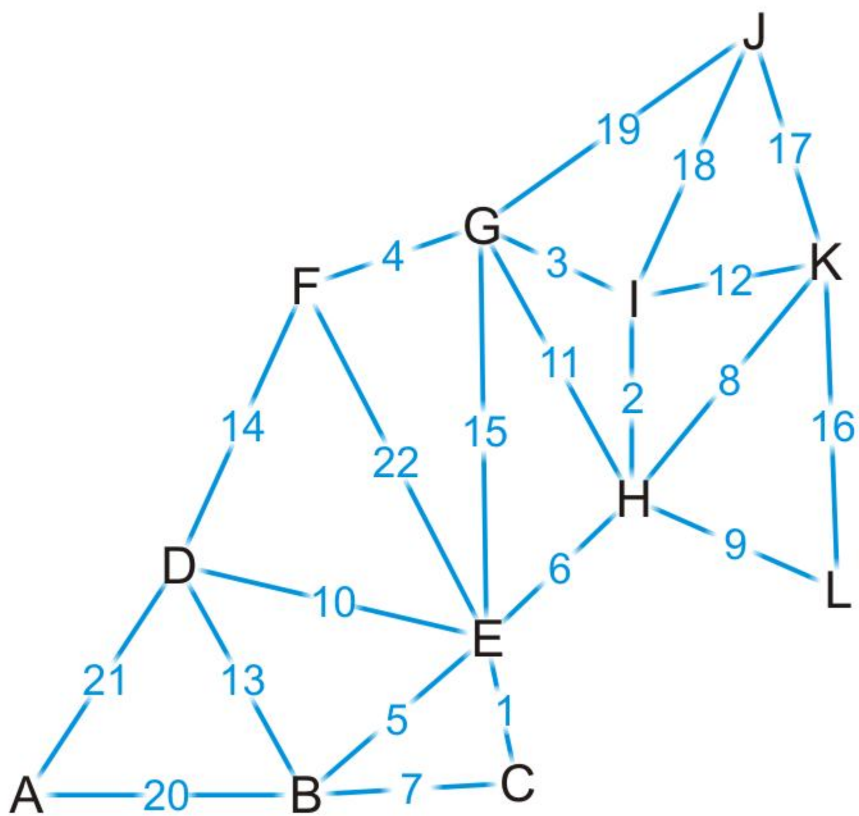


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	**14**	**H**
F	F	∞	Ø
G	F	**19**	**H**
H	**T**	**8**	**K**
I	F	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We are finished with vertex H

– Which vertex do we visit next?

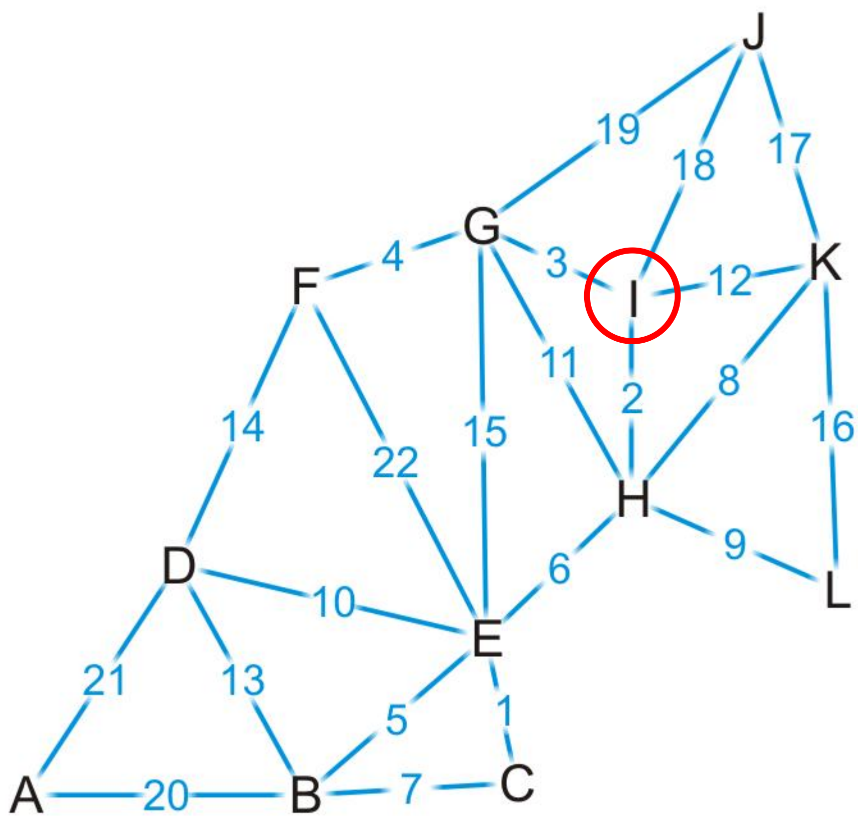


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	F	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

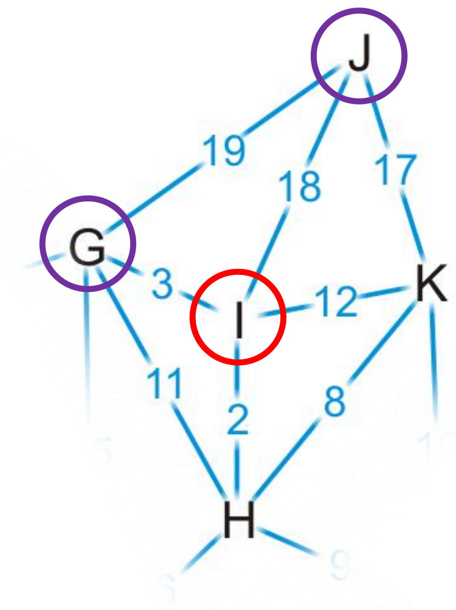


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

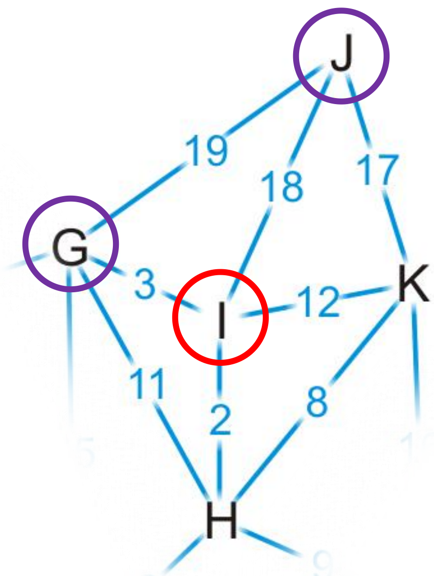
(K, H, I, G) of length 10 + 3 = 13 (K, H, I, J) of length 10 + 18 = 28



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

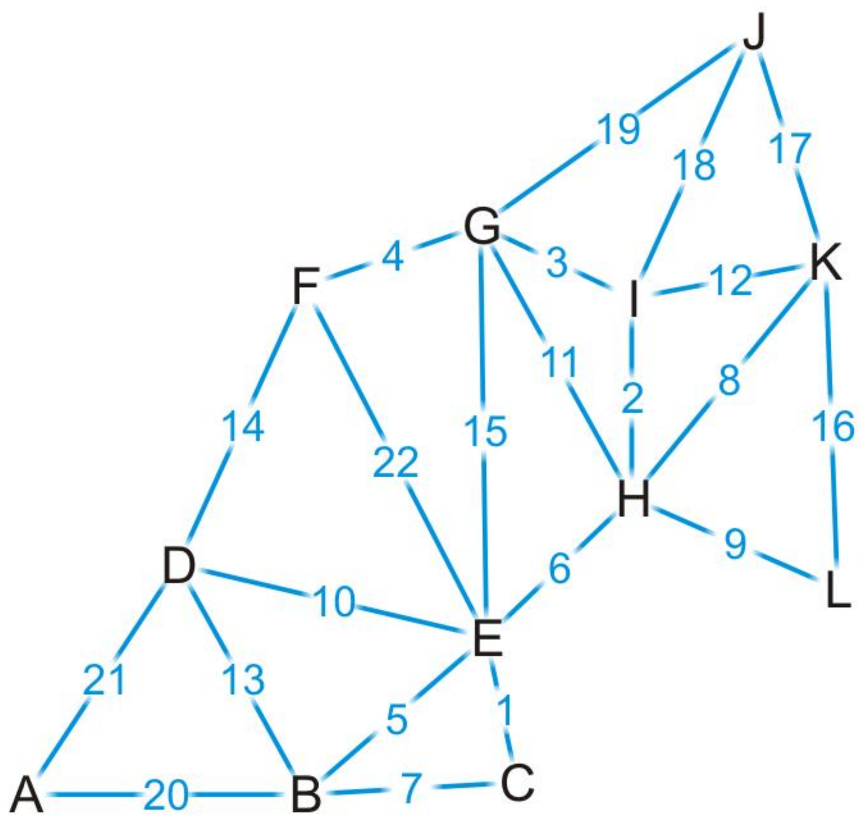
We have discovered a shorter path to vertex G, but (K, J) is still the shortest known path to vertex J



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?

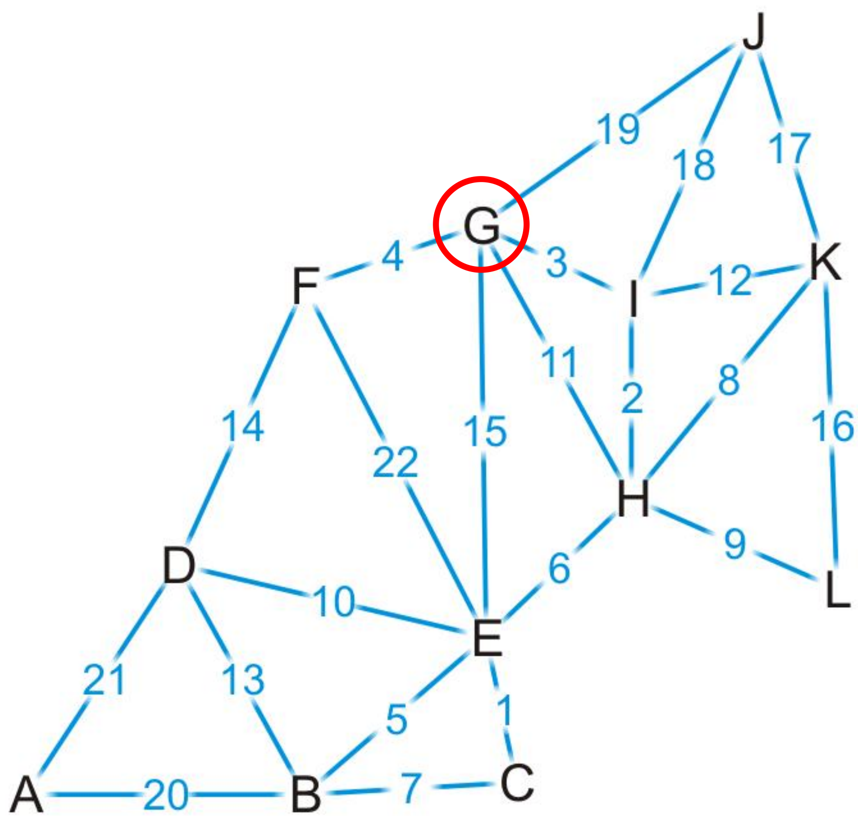


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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 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	**T**	**13**	**I**
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

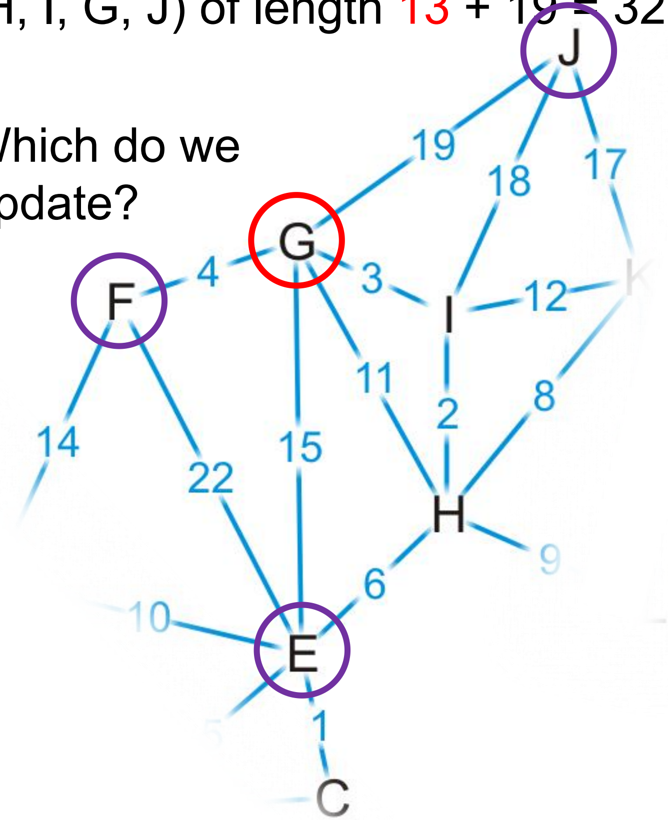
Example

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

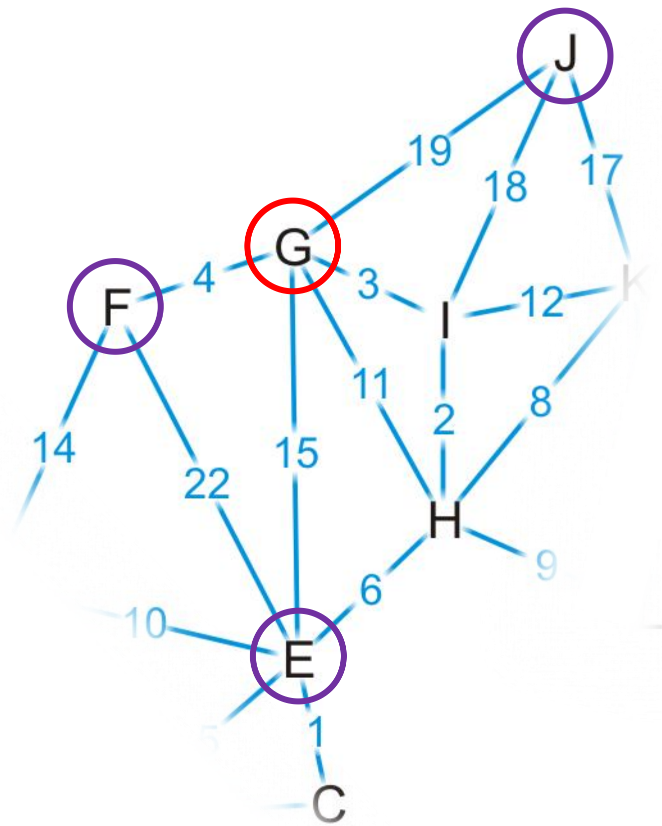
– Which do we update?



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

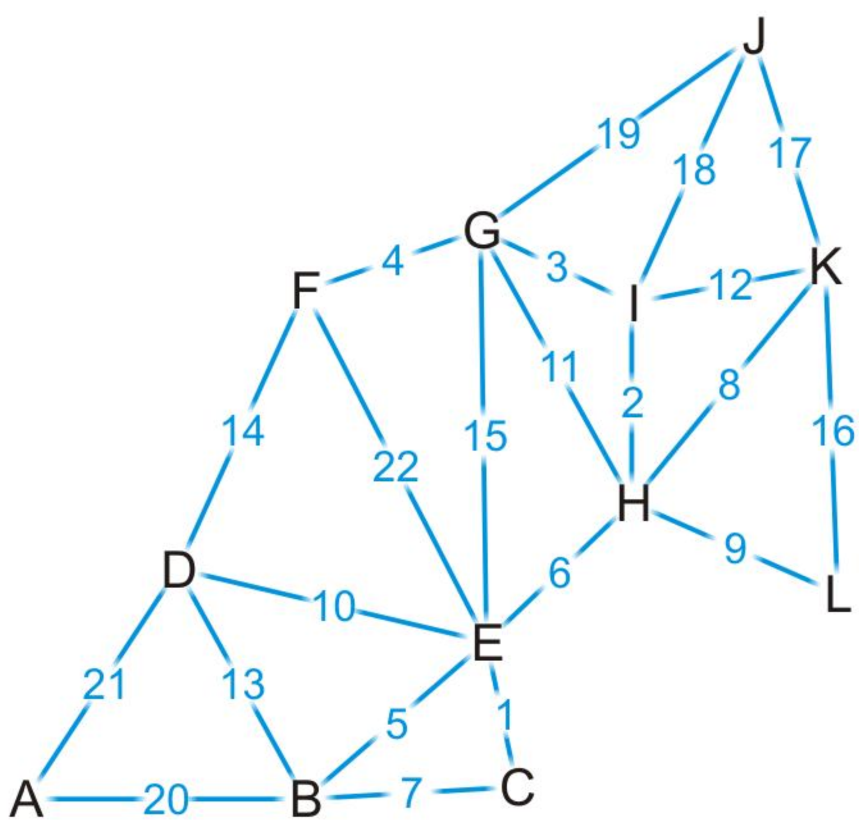
We have now found a path to vertex F



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?

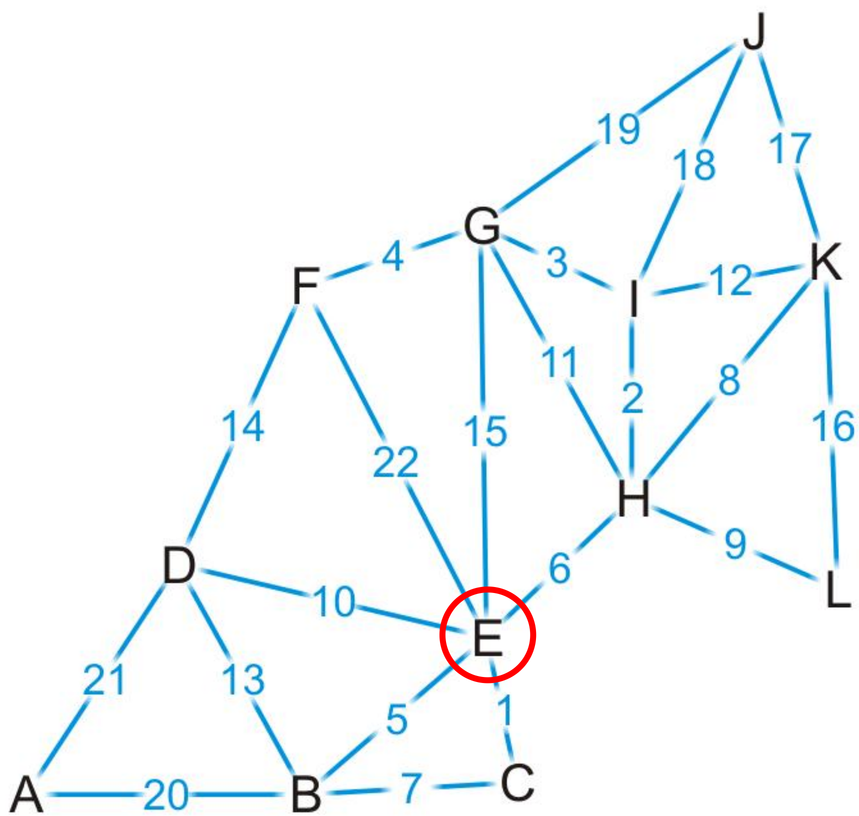


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

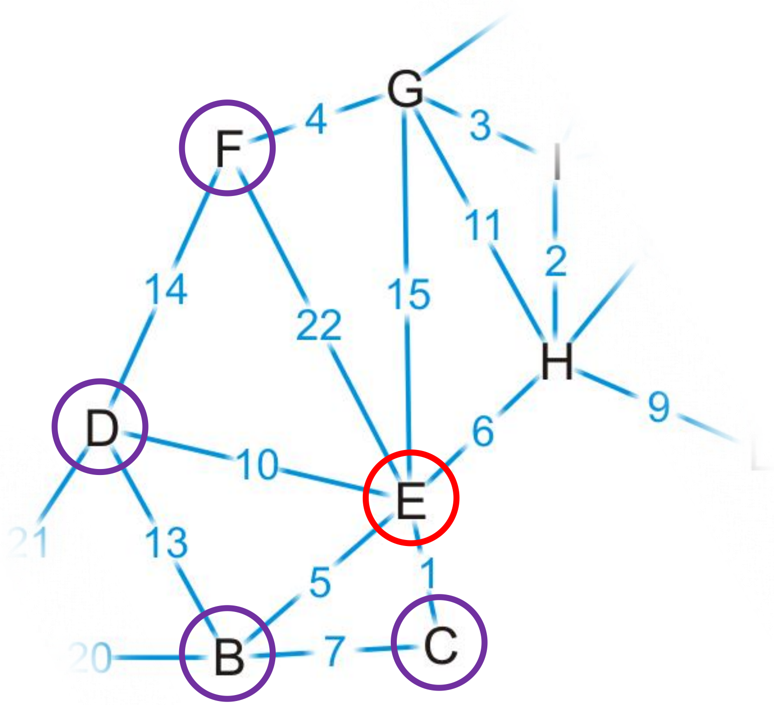


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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:

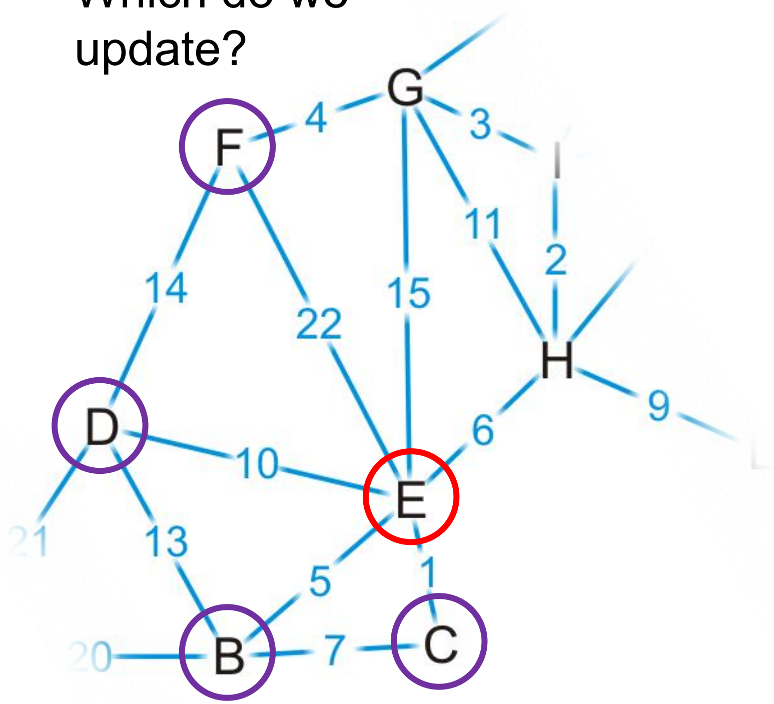
(K, H, E, B) of length 14 + 5 = 19

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

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

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

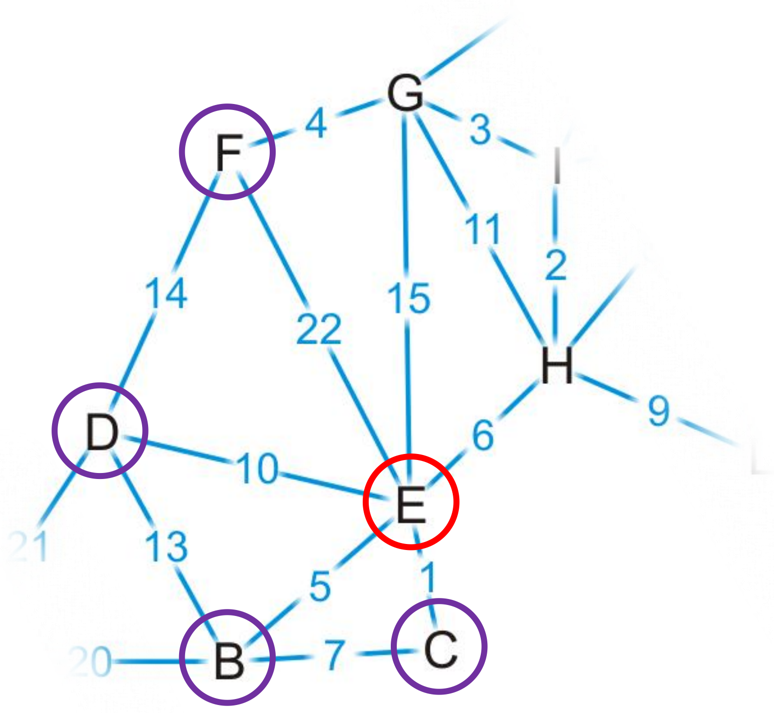
– Which do we update?



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

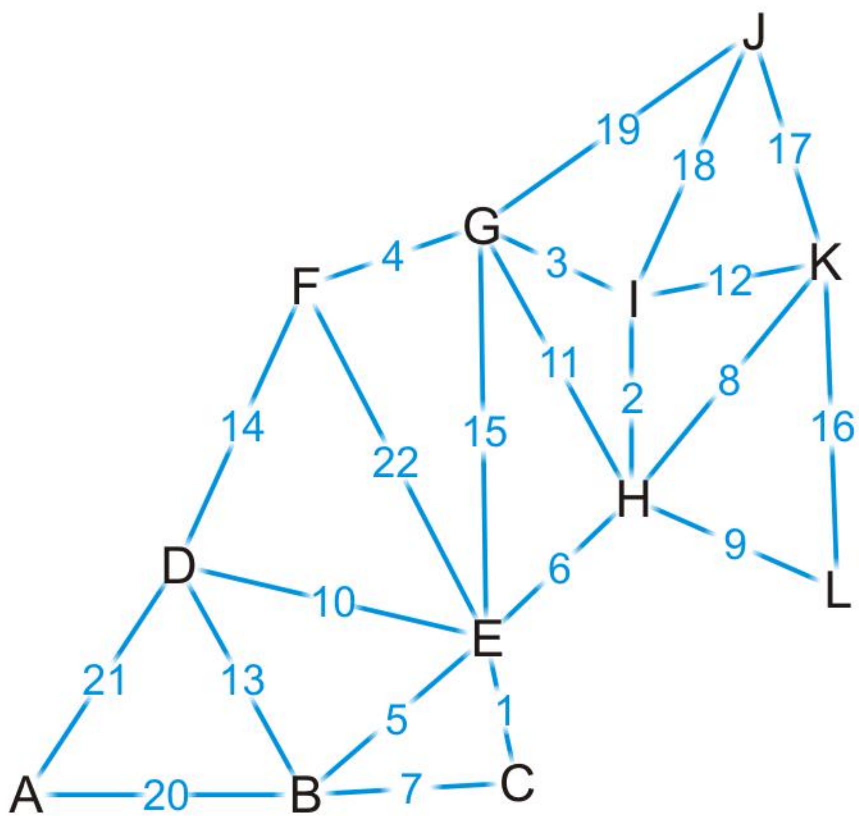
We’ve discovered paths to vertices B, C, D



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?

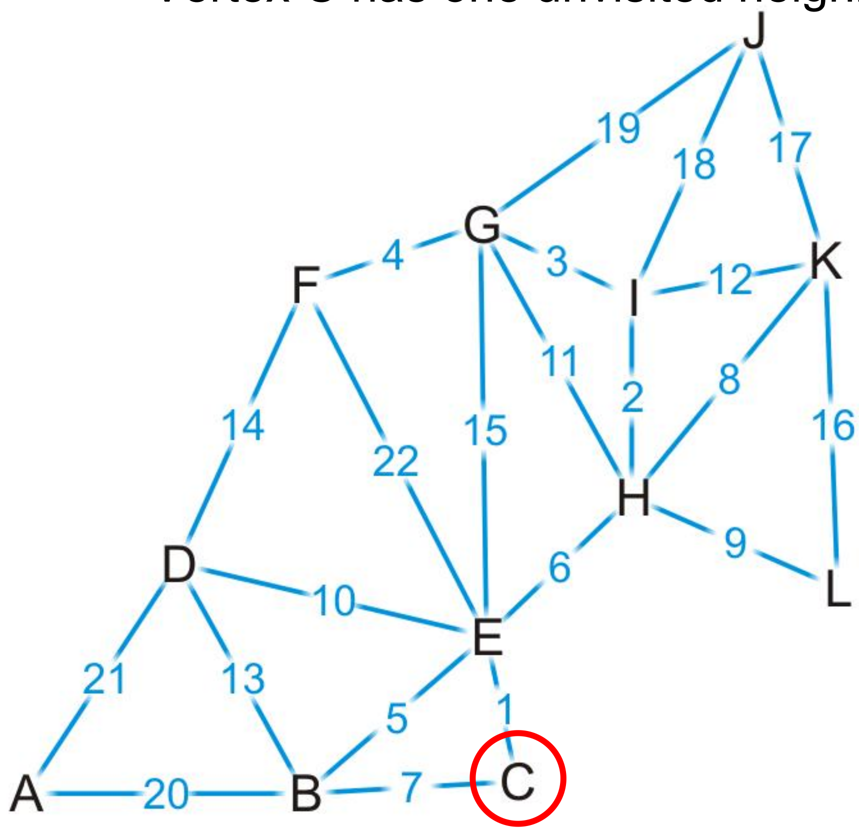


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	F	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

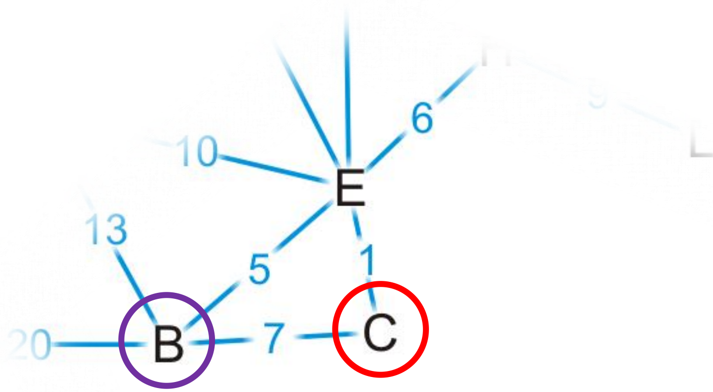


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

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

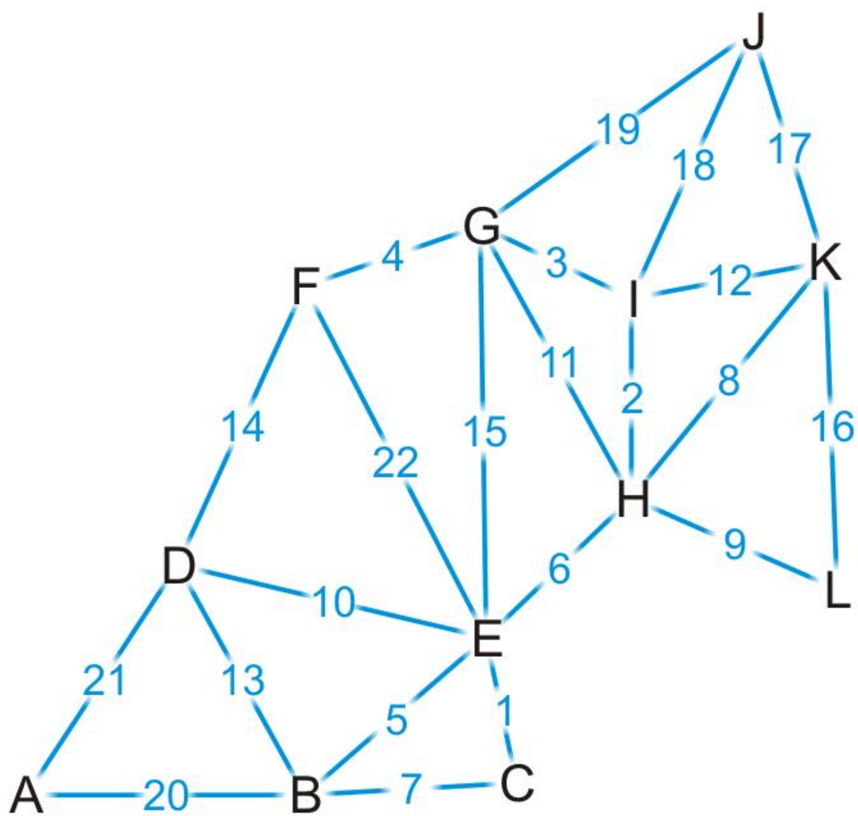
– We have already discovered a shorter path through vertex E



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?

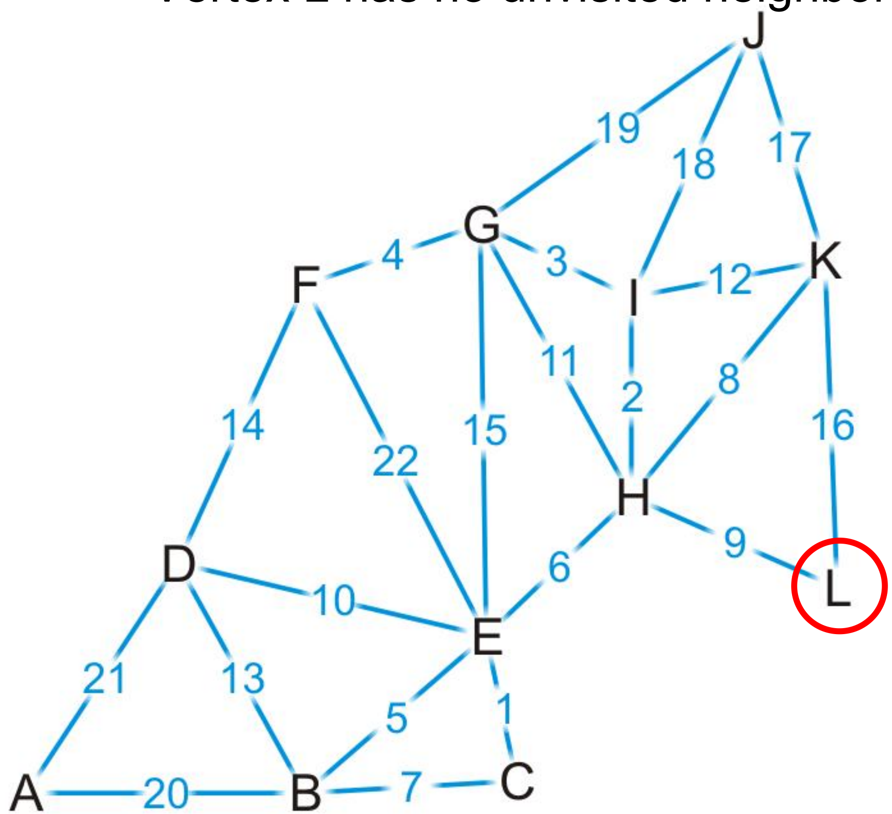


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

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

– Vertex L has no unvisited neighbors

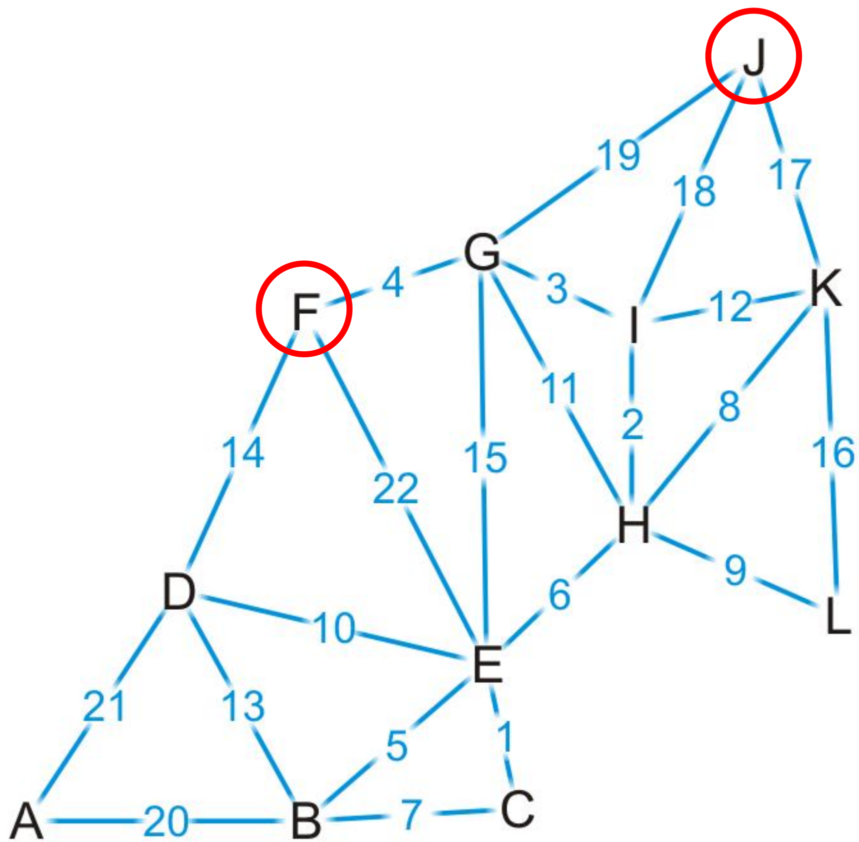


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	**T**	**16**	**K**

Example

Where to next?

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

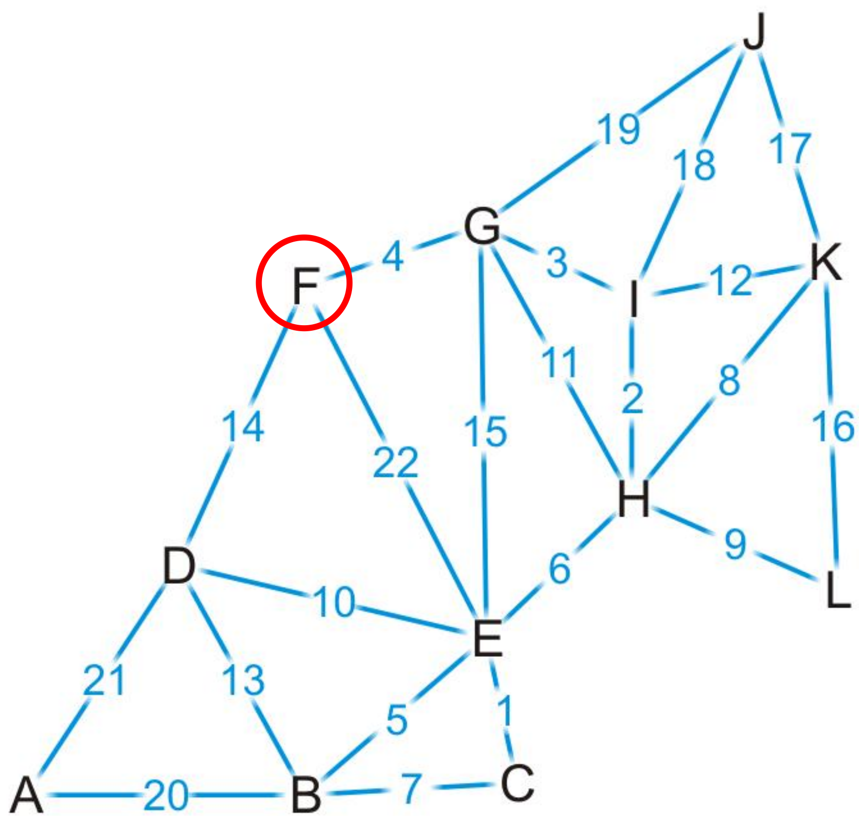


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Let's visit vertex F first

– It has one unvisited neighbor, vertex D

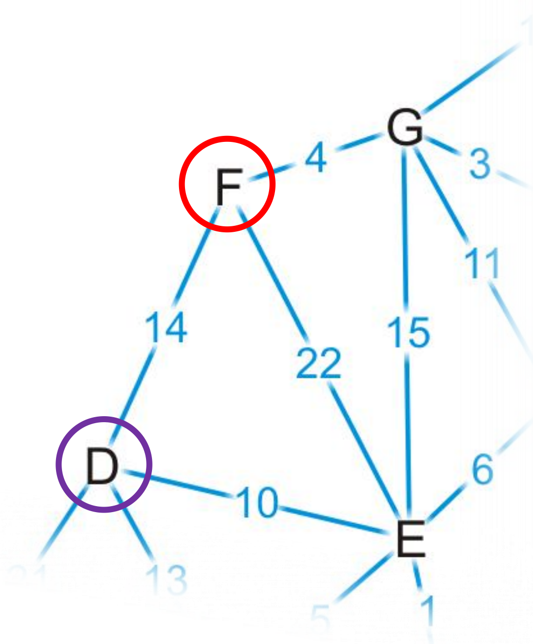


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

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

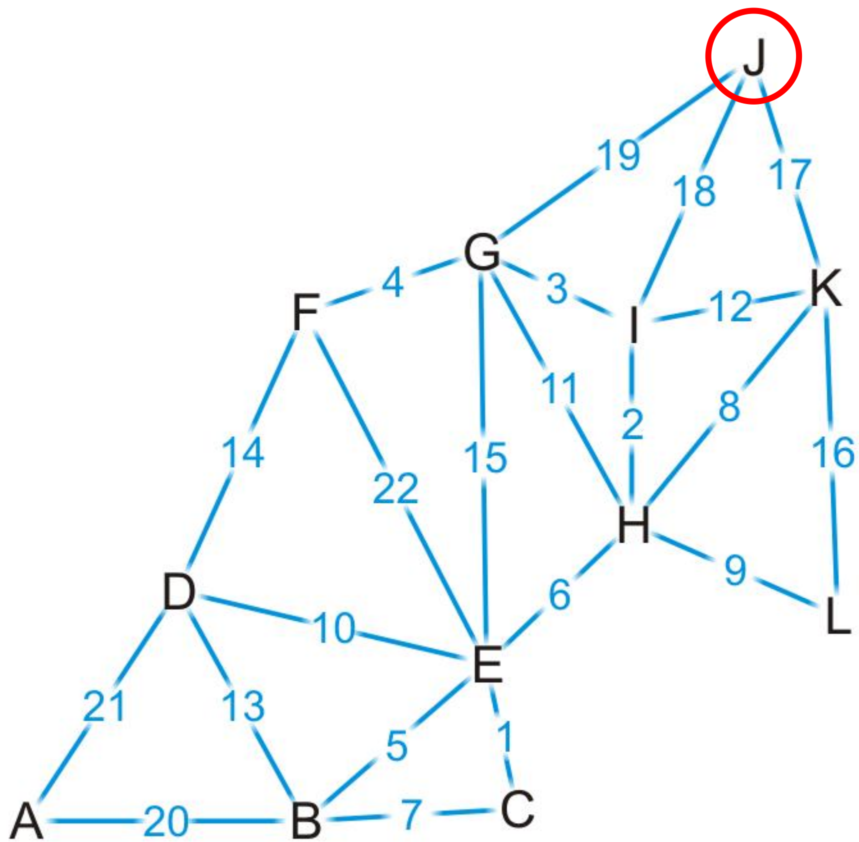


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Now we visit vertex J

– It has no unvisited neighbors



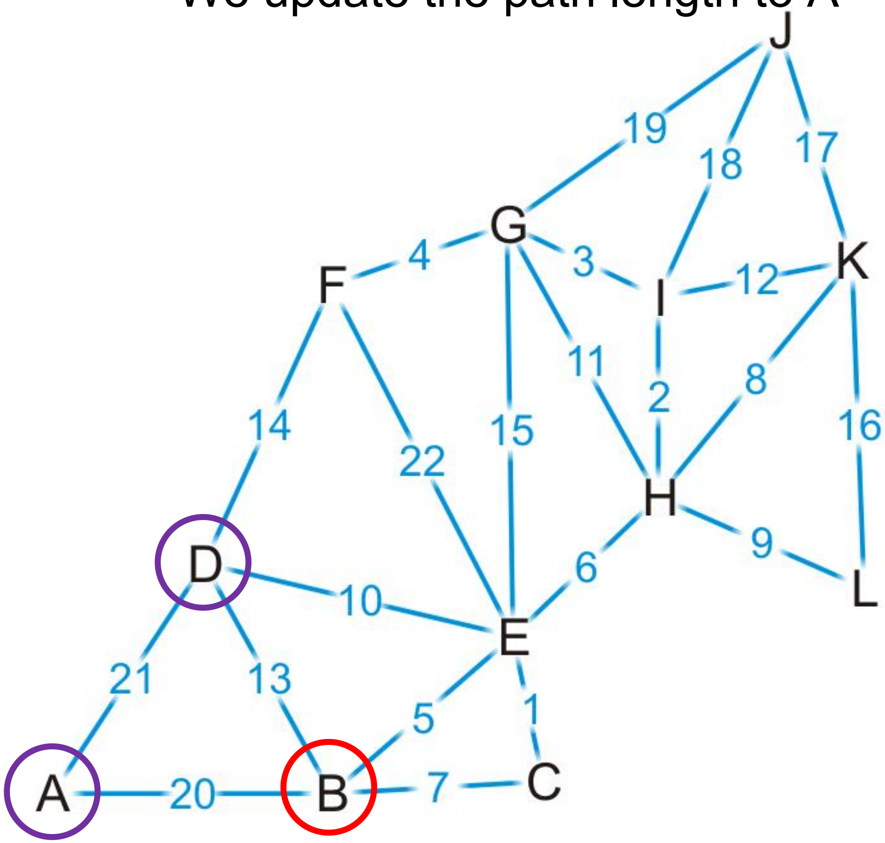
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	**T**	**17**	**K**
K	T	0	Ø
L	T	16	K

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

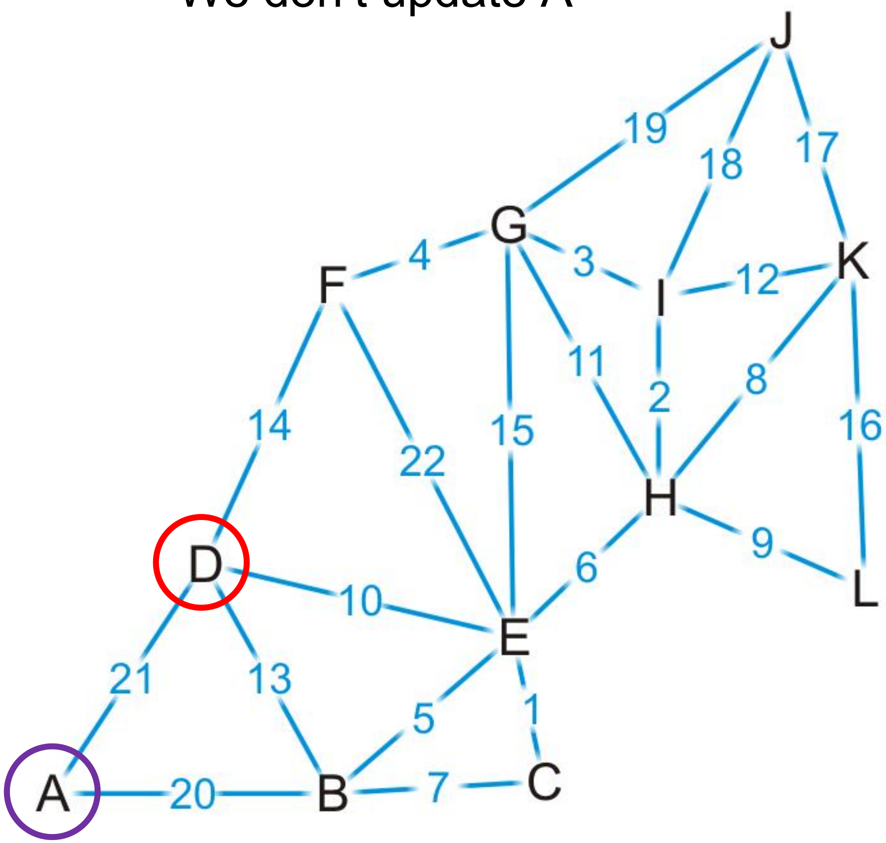


Vertex	Visited	Distance	Previous
A	F	**39**	**B**
B	**T**	**19**	**E**
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex D

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

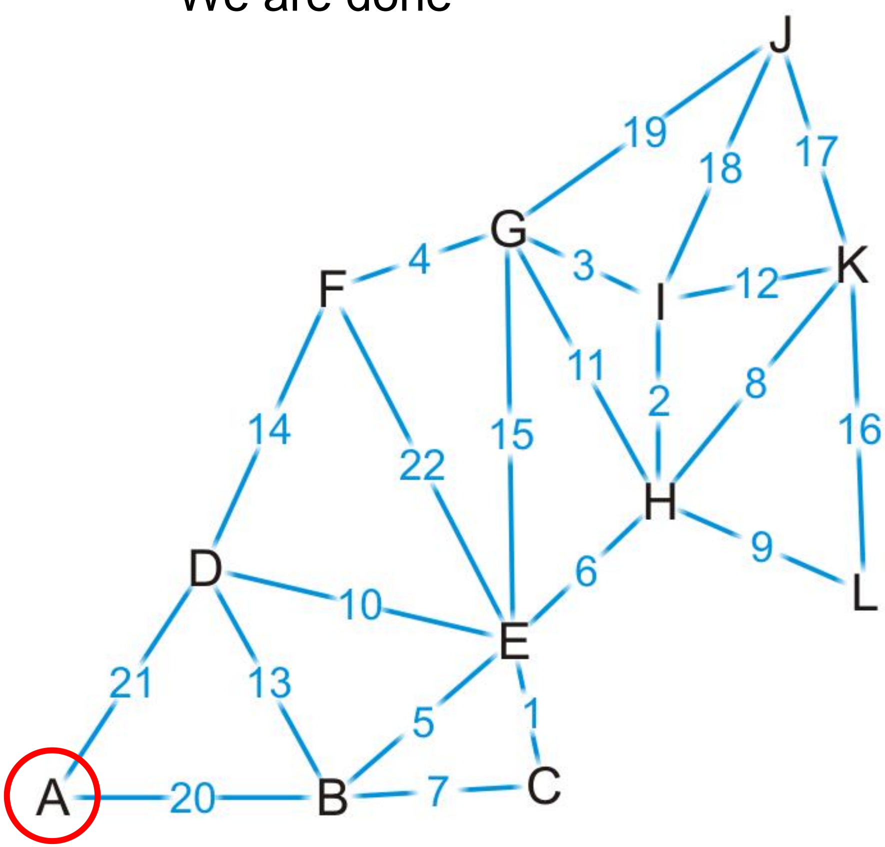


Vertex	Visited	Distance	Previous
A	F	39	B
B	T	19	E
C	T	15	E
D	**T**	**24**	**E**
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Finally, we visit vertex A

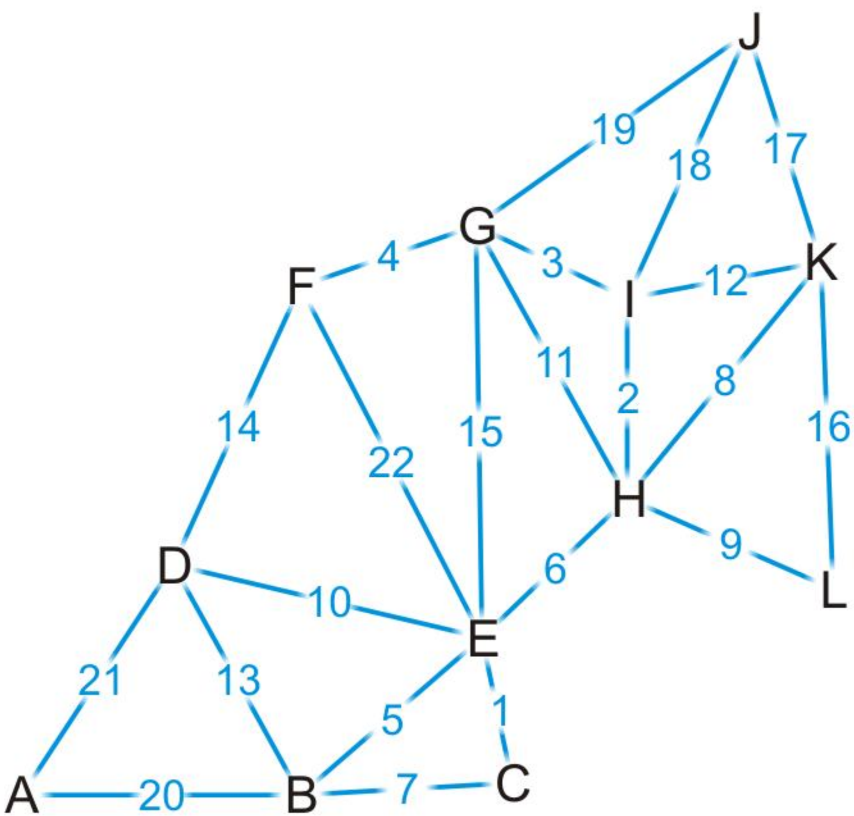
- It has no unvisited neighbors and there are no unvisited vertices left
- We are done



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

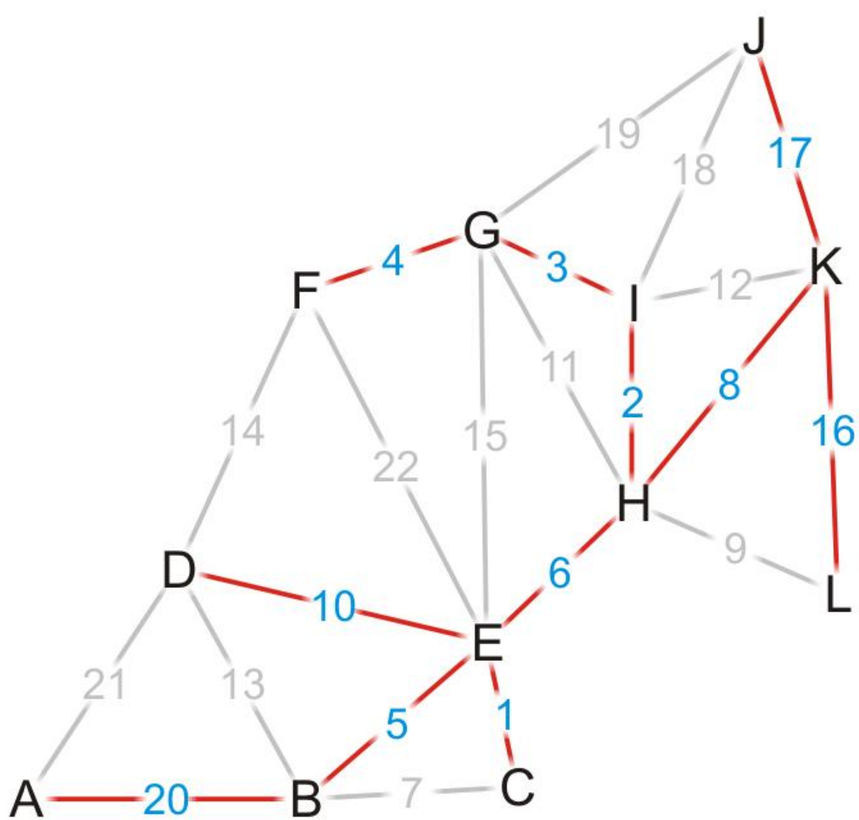
Thus, we have found the shortest path from vertex K to each of the other vertices



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

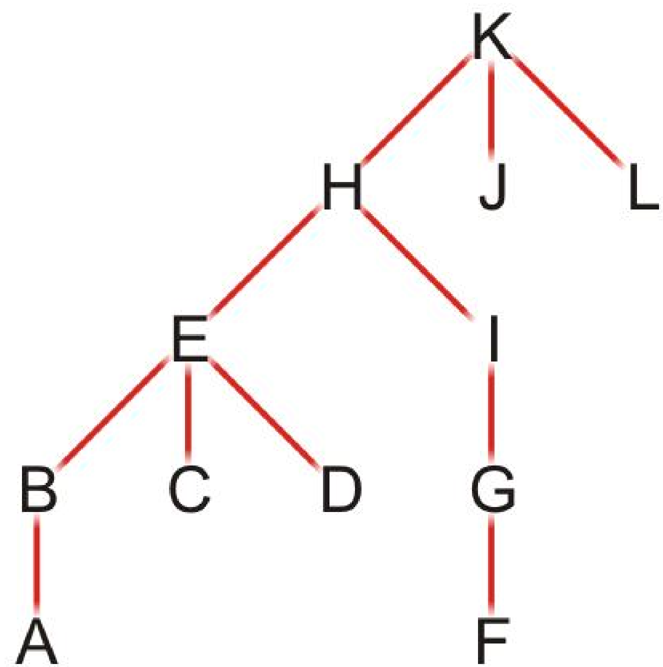


Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Note that this table defines a rooted parental tree (is it also a minimum spanning 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 $|V|$ times, each time finding next closest vertex to the source

- Iterating through the table requires $\Theta(|V|)$ 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 $|E| = \mathrm{O}(|V|^2)$

Can we do better?

- How about using a priority queue to find the closest vertex?
  - Assume we are using a binary heap

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?

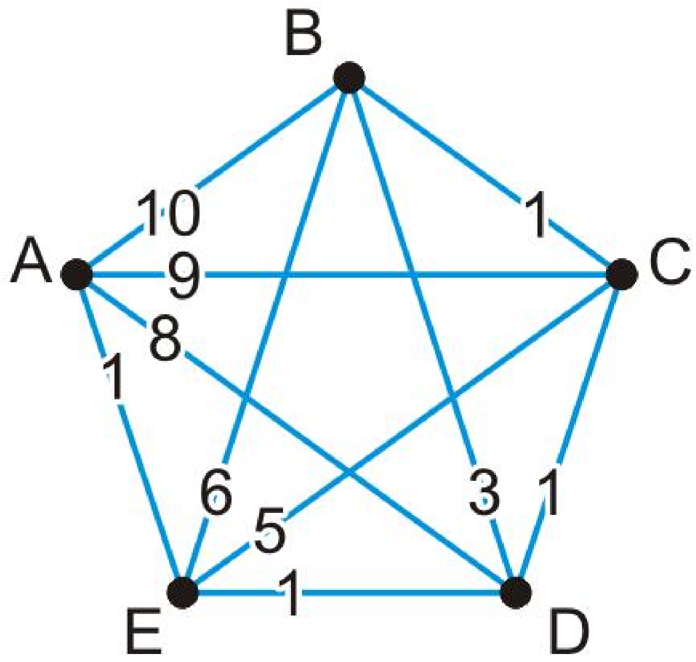
- Recall that each edge visited may result in a distance being updated
- Thus, the work required for this is $O(|E| \ln(|V|))$

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

Implementation and analysis

Here is an example of a worst-case scenario:

- Immediately, all of the vertices are placed into the queue
- Each time a vertex is visited, all the remaining vertices are checked, and in succession, each is pushed to the top of the binary heap



Implementation and analysis

We could use a different heap structure:

– A Fibonacci heap is a node-based heap
– Pop is still $O(\ln(|V|))$, but inserting and moving a key is $\Theta(1)$
– Thus, because we are only calling pop $|V| - 1$ times, the overall run-time reduces to $O(|E| + |V| \ln(|V|))$

Implementation and analysis

Thus, we have two run times when using

- A binary heap: $\mathrm{O}(|E| \ln(|V|))$
- A Fibonacci heap: $\mathrm{O}(|E| + |V| \ln(|V|))$

Questions: Which is faster if $|E| = \Theta(|V|)$? How about if $|E| = \Theta(|V|^2)$?

Summary

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

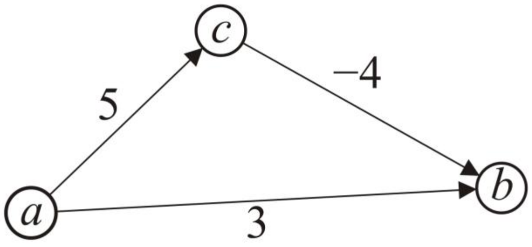
- Start with the initial vertex
- Continue finding the next vertex that is closest

Dijkstra's algorithm always finds the next closest vertex

- It solves the problem in $\mathrm{O}(|E| + |V| \ln(|V|))$ time

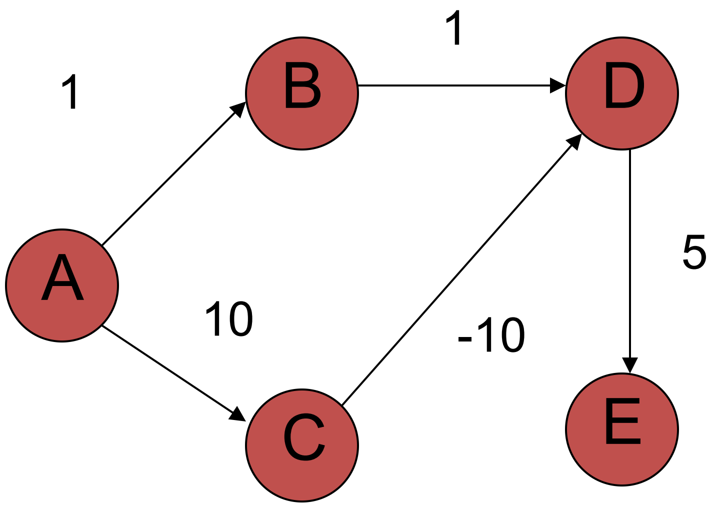
Negative Weights

If some of the edges have negative weight, so long as there are no cycles with negative weight, the Bellman-Ford algorithm will find the minimum distance

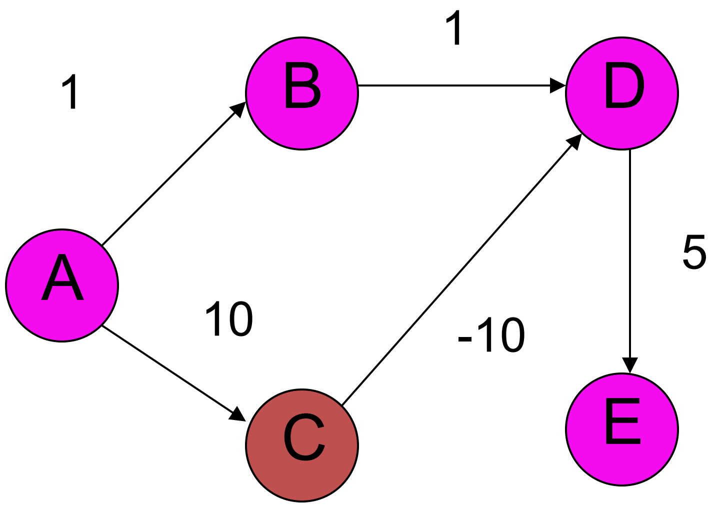


– It is slower than Dijkstra's algorithm

What about Dijkstra's on…?

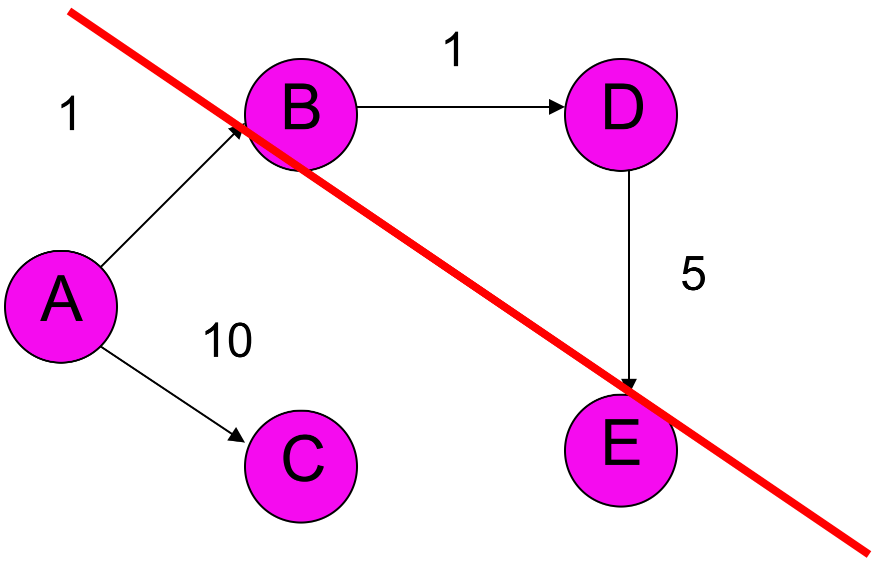


What about Dijkstra's on…?



What about Dijkstra's on…?

Dijkstra's algorithm only works for positive edge weights



Bounding the distance

- Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance
  - start of at ∞
  - only update the value if we find a shorter distance
- An update procedure

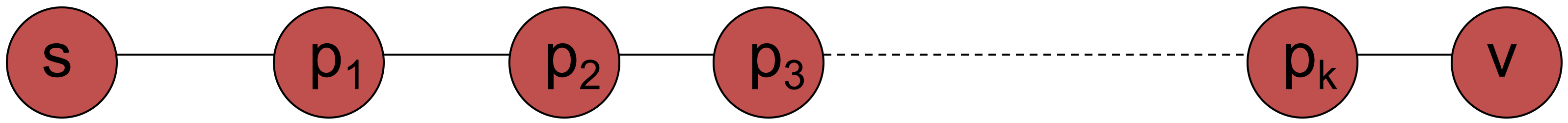
$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- Can we ever go wrong applying this update rule?
  - We can apply this rule as many times as we want and will never underestimate dist[v]
- When will dist[v] be right?
  - If u is along the shortest path to v and dist[u] is correct

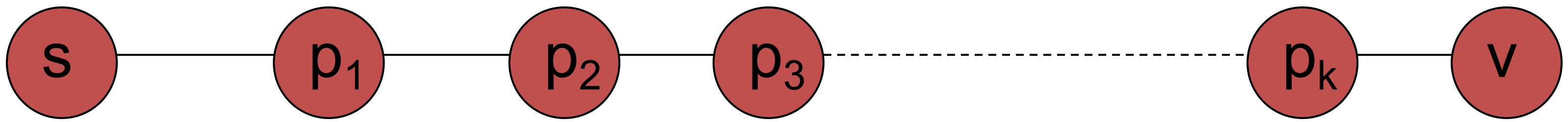
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- Consider the shortest path from s to v



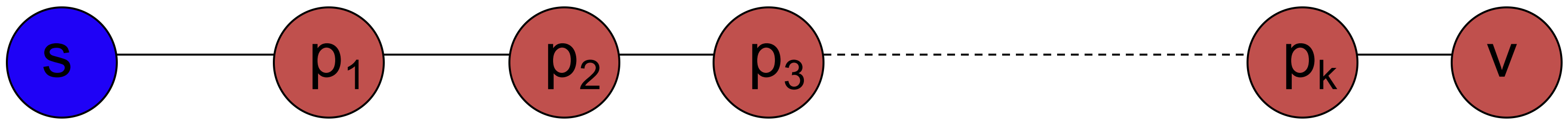
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

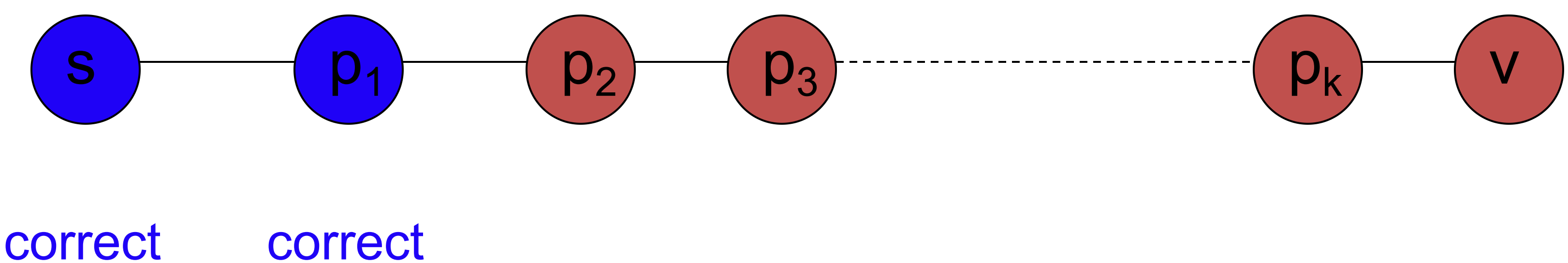
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



correct

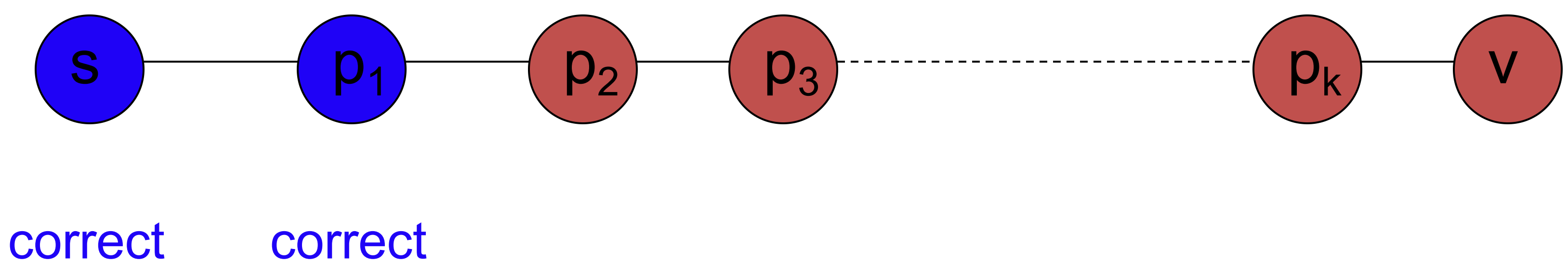
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



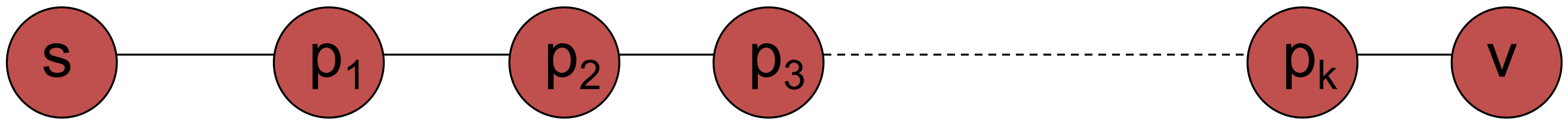
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- Does the order that we update the vertices matter?



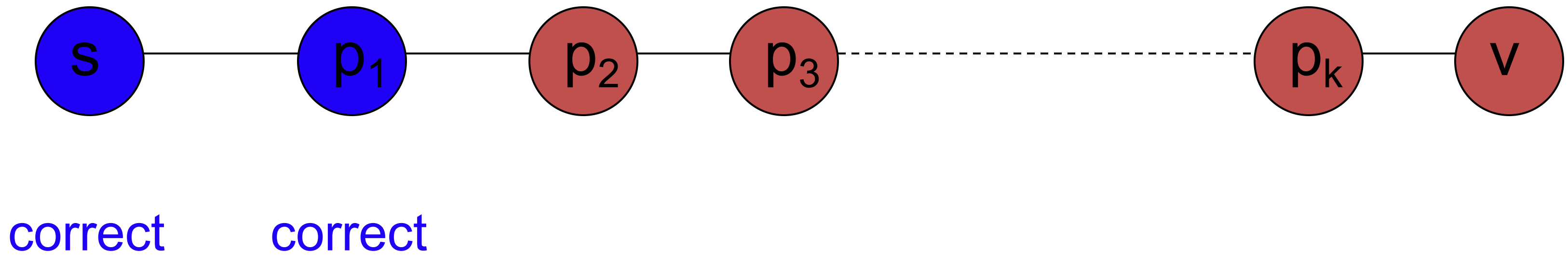
$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



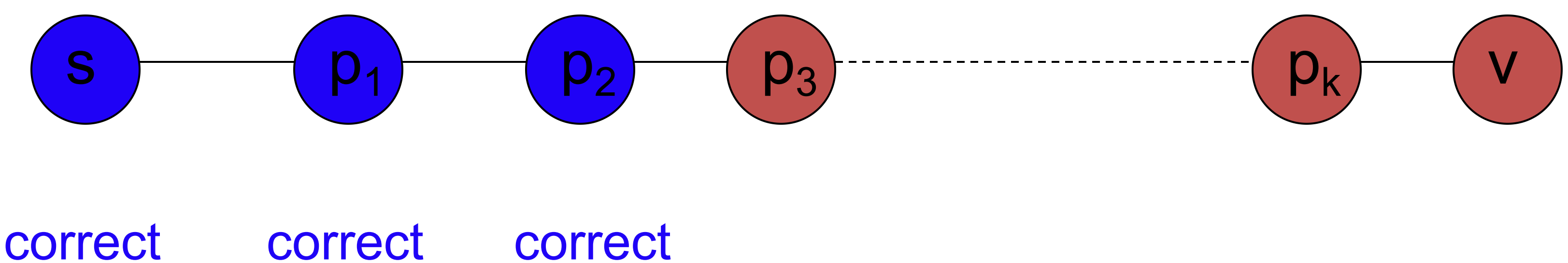
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



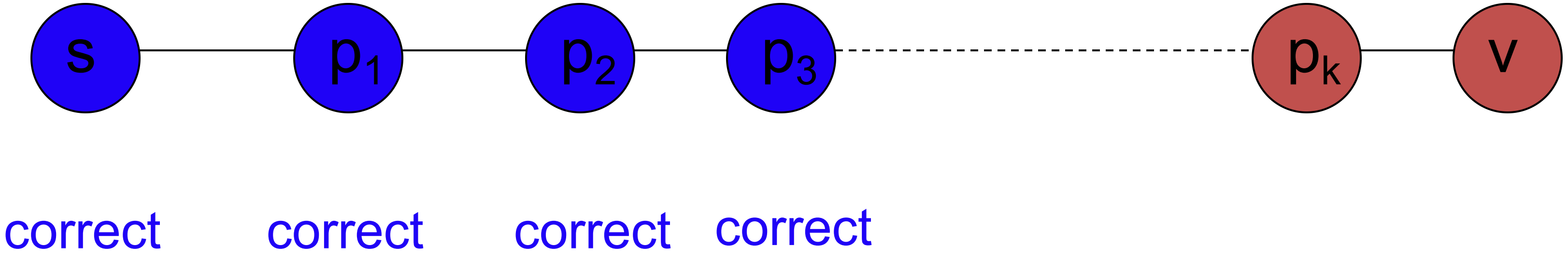
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



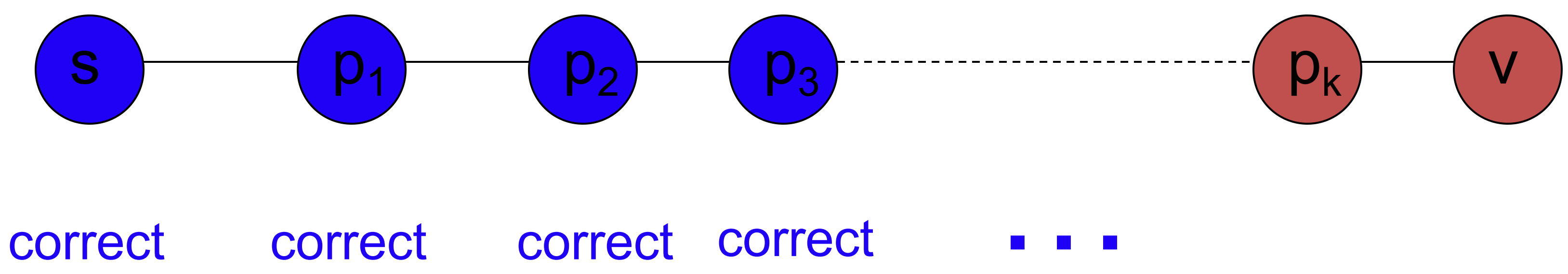
$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



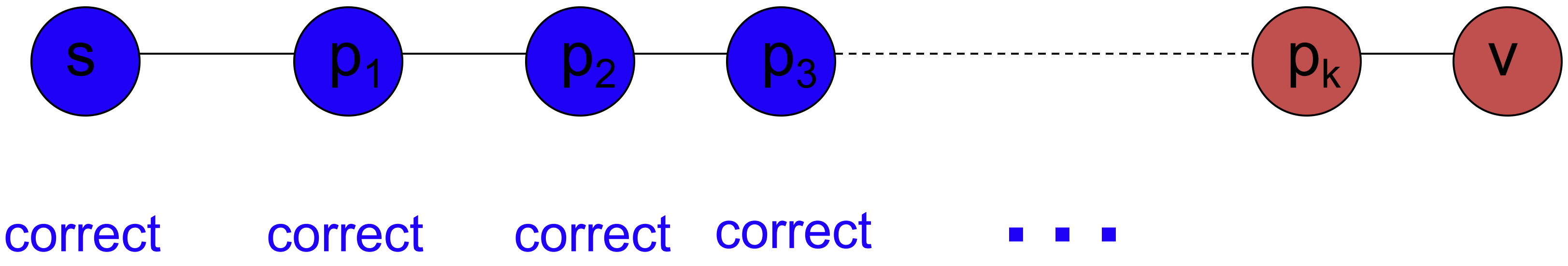
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What is the longest (vetex-wise) the path from s to any node v can be?
  - |V| - 1 edges/vertices



Bellman-Ford algorithm

BELLMAN-FORD(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V| - 1$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
        return $false$

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Initialize all the distances

Bellman-Ford algorithm

BELLMAN-FORD(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V| - 1$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
        return $false$

iterate over all edges/vertices and apply update rule

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

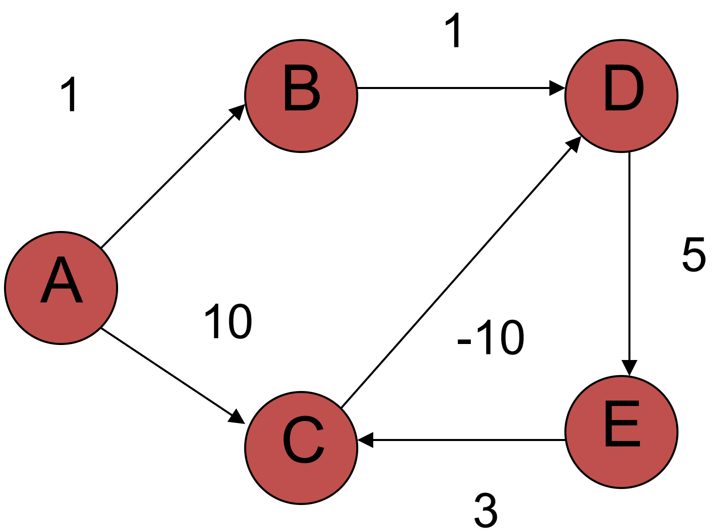
Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

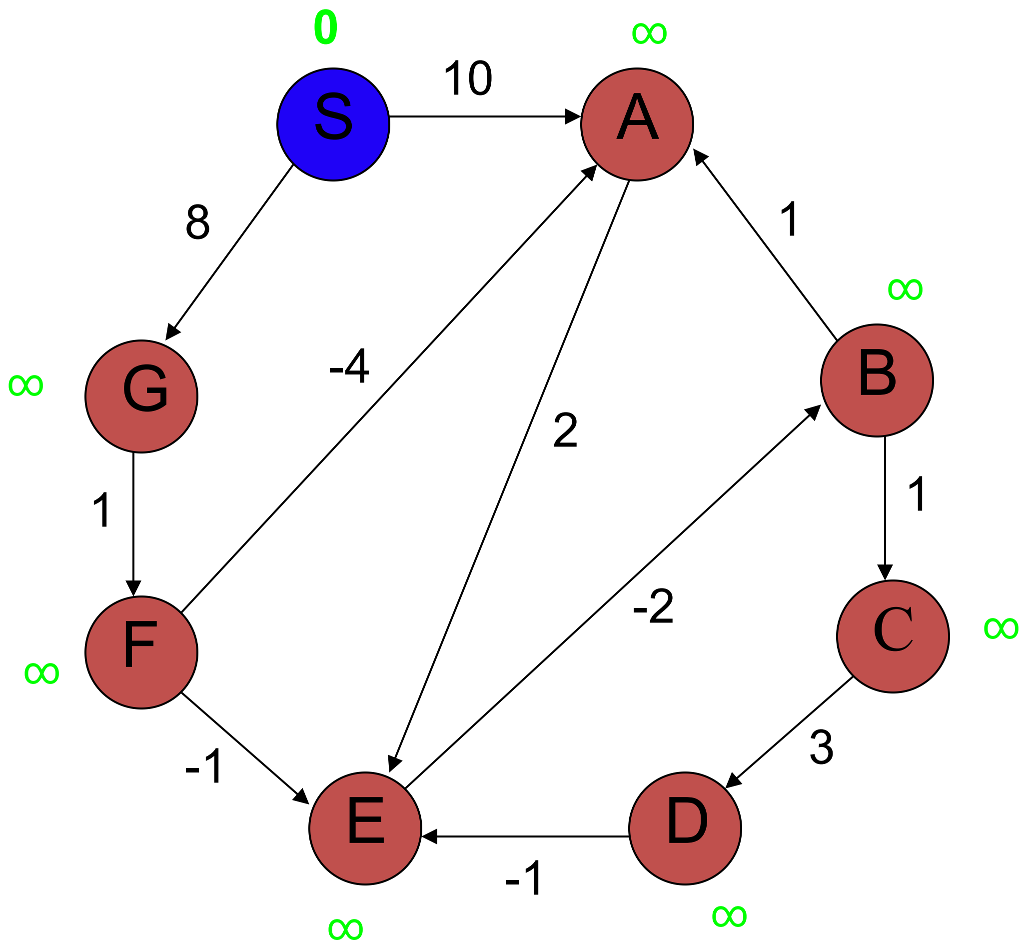
check for negative cycles

Negative cycles

What is the shortest path from a to e?

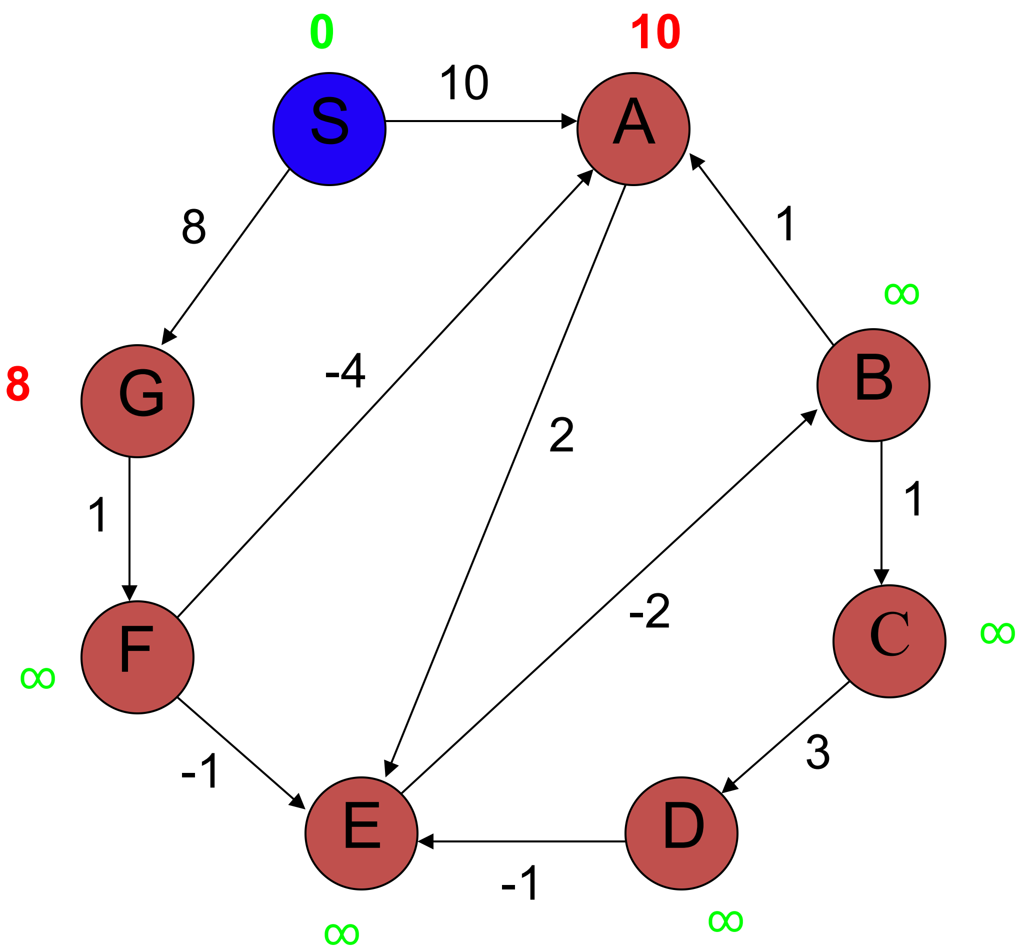


Bellman-Ford algorithm



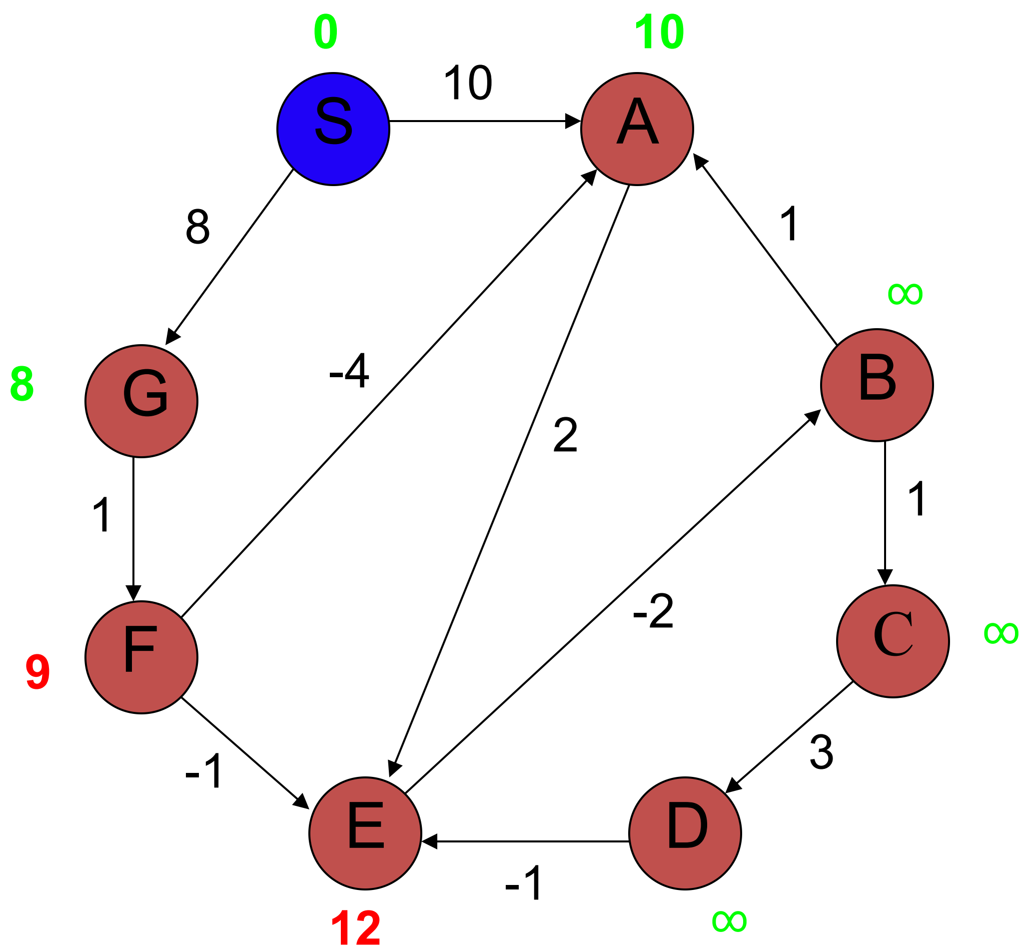
Iteration: 0

Bellman-Ford algorithm



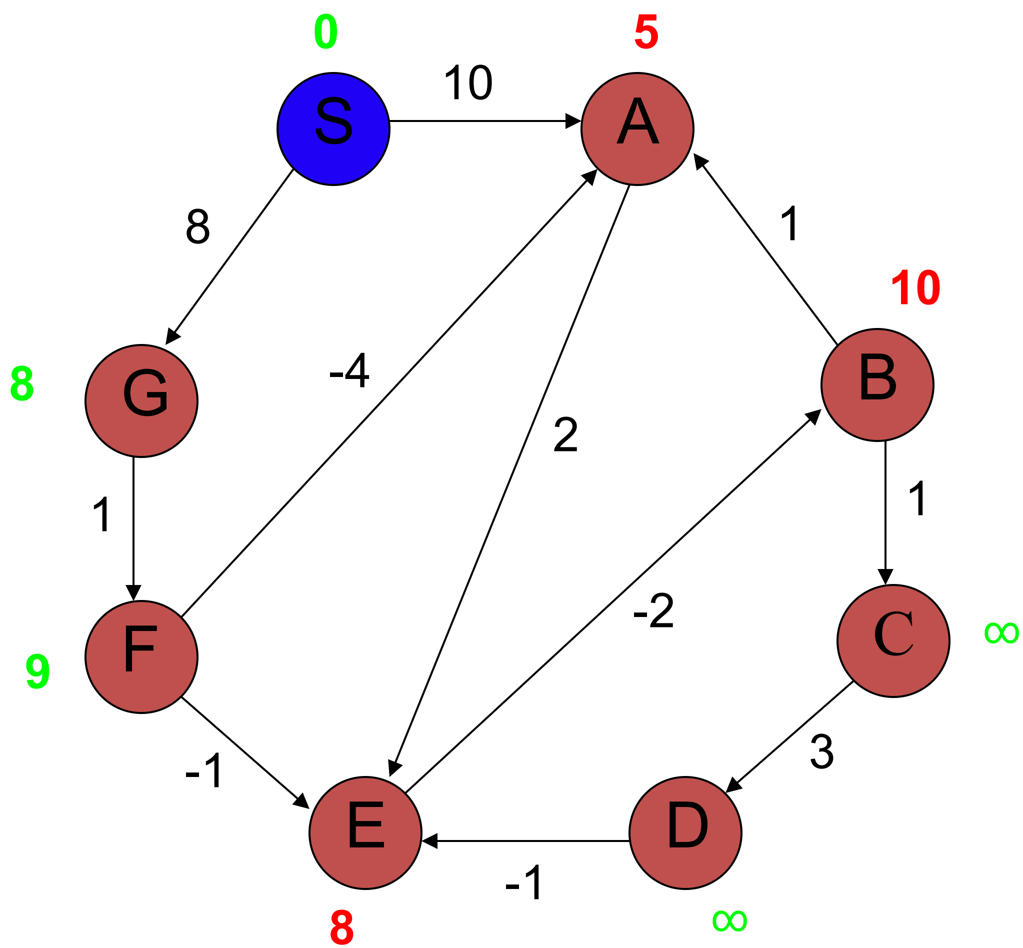
Iteration: 1

Bellman-Ford algorithm



Iteration: 2

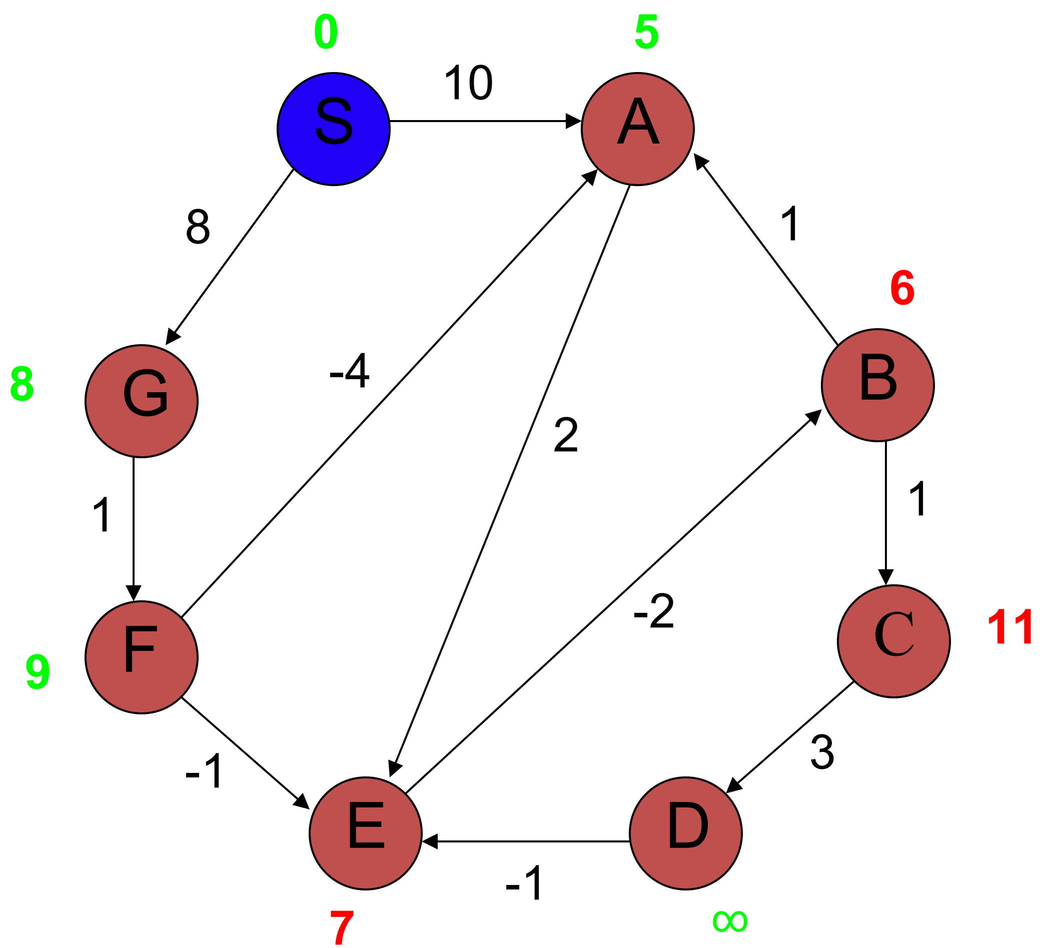
Bellman-Ford algorithm



Iteration: 3

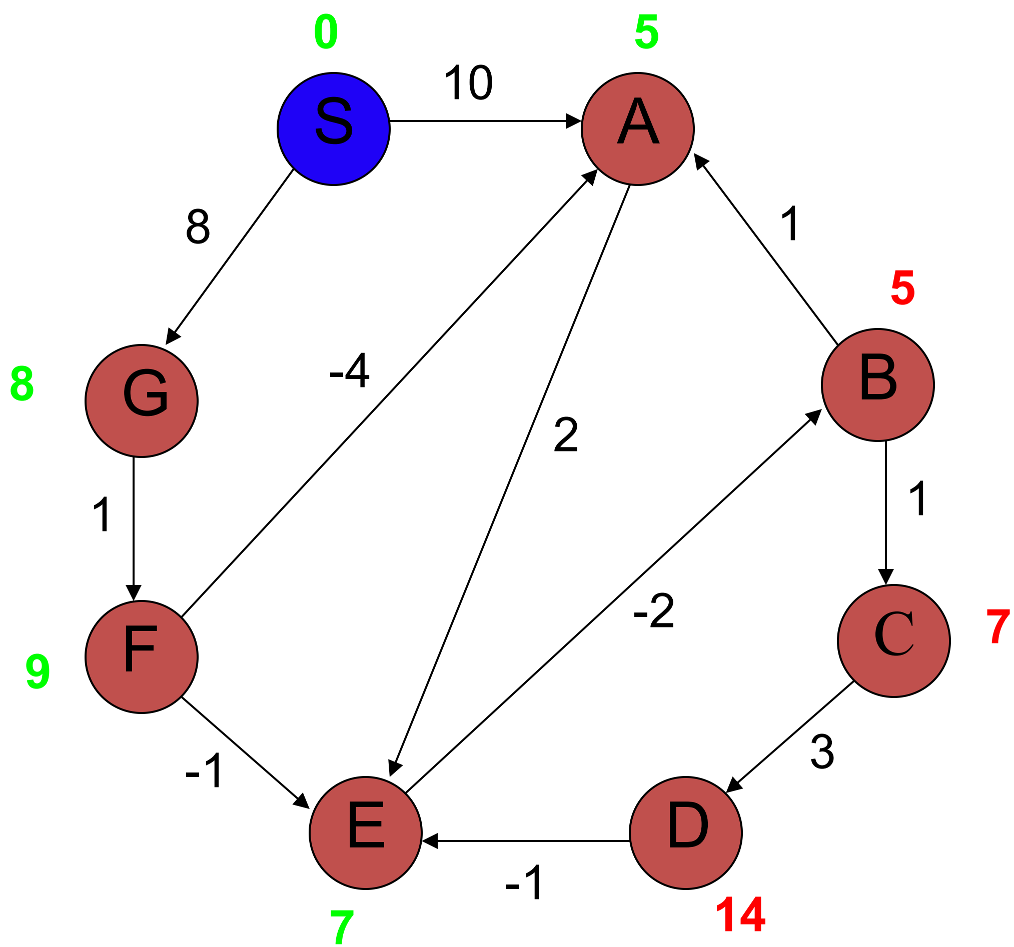
A has the correct distance and path

Bellman-Ford algorithm



Iteration: 4

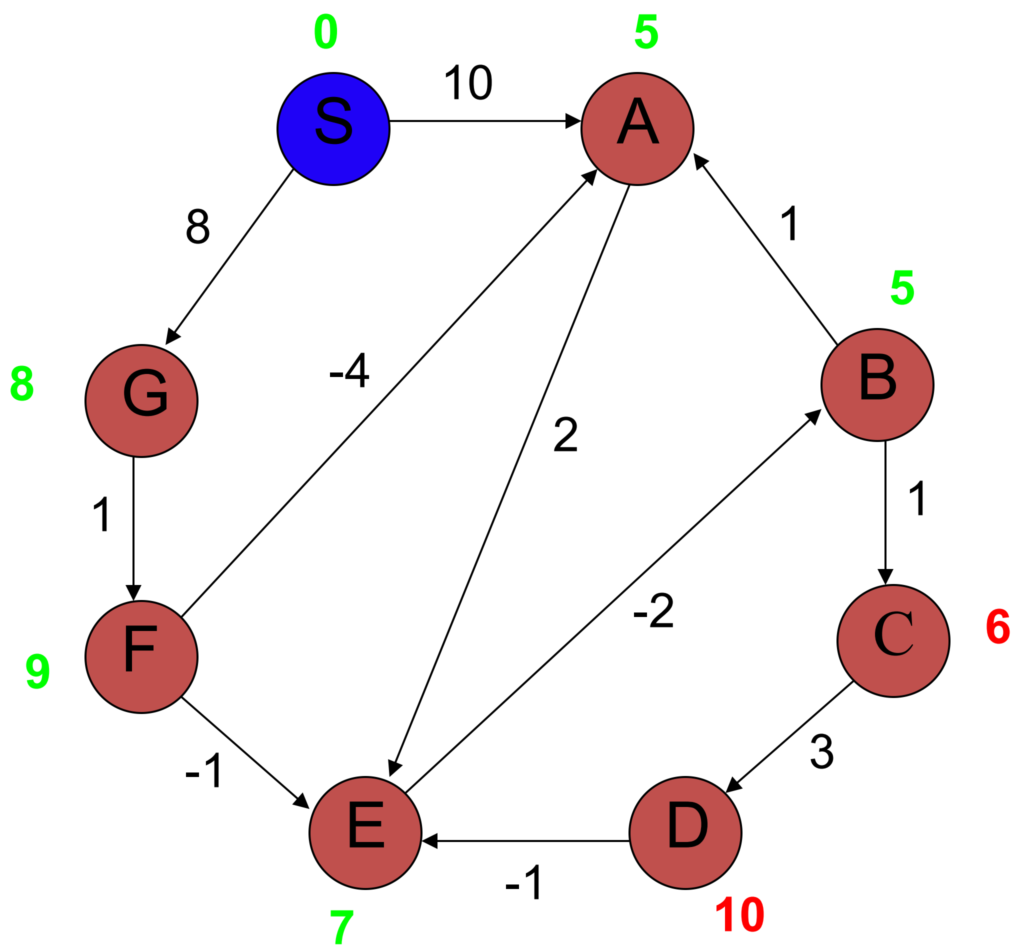
Bellman-Ford algorithm



Iteration: 5

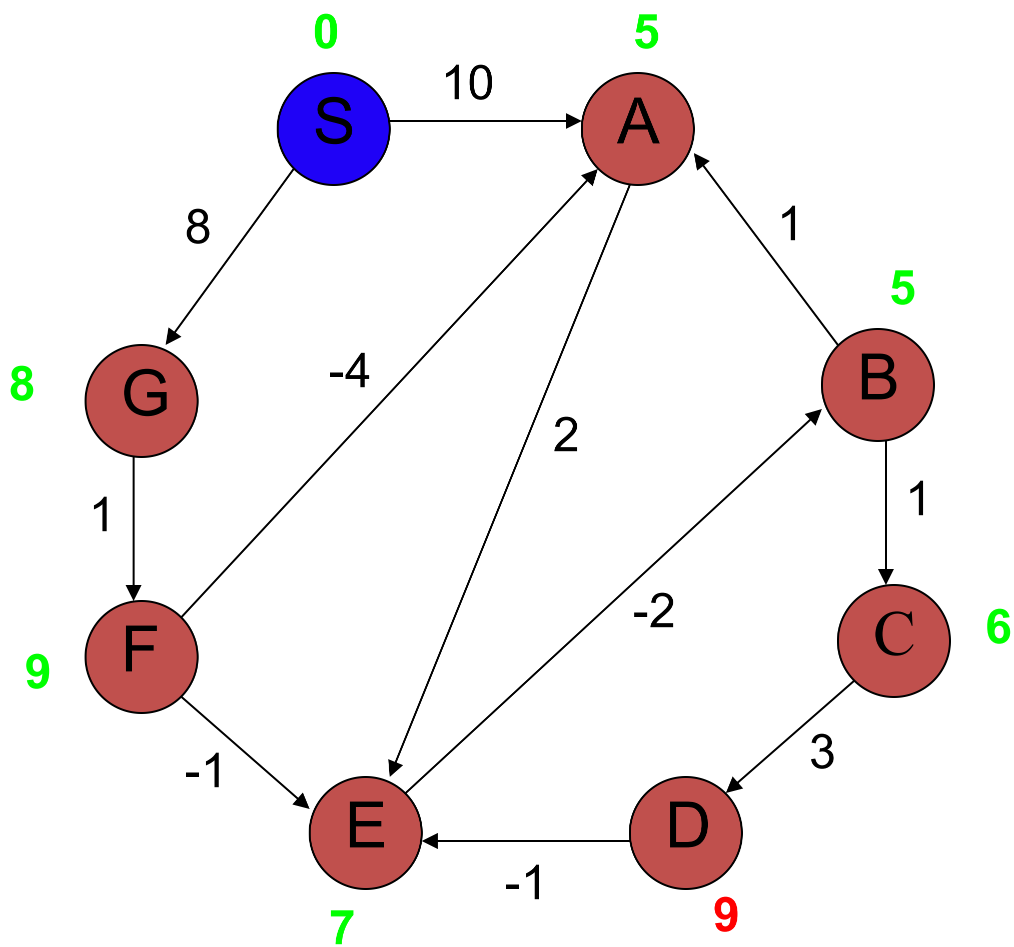
B has the correct distance and path

Bellman-Ford algorithm



Iteration: 6

Bellman-Ford algorithm



Iteration: 7

D (and all other nodes) have the correct distance and path

Correctness of Bellman-Ford

- Loop invariant: After iteration i, all vertices with shortest paths from s of length i edges or less have correct distances

Bellman-Ford(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V| - 1$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
        return $false$

Runtime of Bellman-Ford

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

O(|V| |E|)

Runtime of Bellman-Ford

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Can you modify the algorithm to run faster (in some circumstances)?

Summary

- Definition and applications
- Dijkstra’s algorithm
  - Single source shortest distance (non-negative weights)
- Bellman-Ford algorithm
  - For graphs with negative weights (but no cycles with negative weight)