# Shortest Path Problem

Dr Truong Dinh Huy

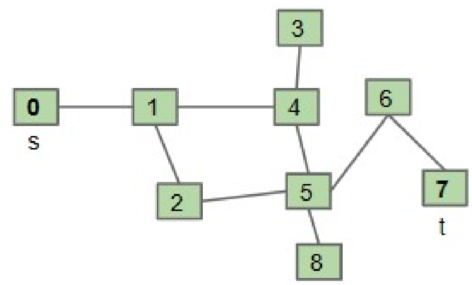
## Outline

Today we try to solve two famous problems:

- **s-t Path:** Is there a path between vertices s and t?
  - Traversal
- **Shortest s-t Path:** How to find the shortest path between vertices s and t?
  - Unweighted graph
  - Weighted graph

# s-t Path

- Given source vertex s and a target vertex t, is there a path between s and t?
- Requires us to traverse the graph somehow.
  - Start from s and apply traversal until we reach t




# Tree Traversal

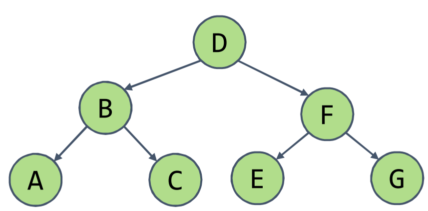
Level Order (Breadth First Traversals)

- Visit top-to-bottom, left-to-right (like reading in English): DBFACEG

Depth First Traversals

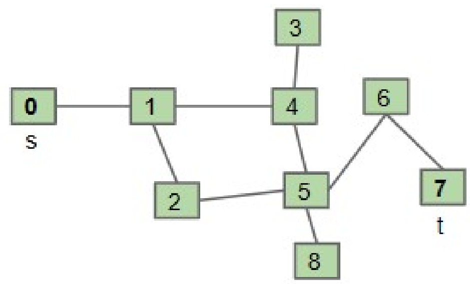
- 3 types: Preorder, Post-order , In-order (only for binary tree)
- Basic (rough) idea: Traverse "deep nodes" (e.g. A) before shallow ones (e.g. F).
- Note: Traversing a node is different than "visiting" a node

- Preorder: DBACFEG
- Postorder: ACBEGFD
- Inorder: ABCDEFG
- BFS: DBFACEG



# Graph Traversals

- We can consider a graph as a tree and use the same methods with a modification:
  - Mark some vertices visited. Why?

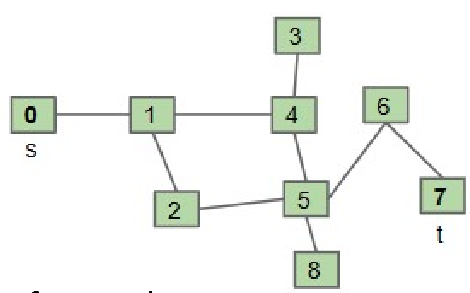



# DFS: Preorder

- Recursive DFS traversal

  dfs(v)
    **mark v as visited**
    for (each unvisited vertex u adjacent to v)
      dfs(u)

Ordering traversal: 012543678




- We don't use Post-order and In-order for graphs

## How to determine the route from s to t?

Create a following table and update the table when we traverse

dfs(v):

mark v.

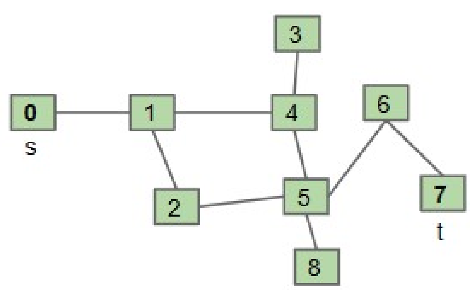
for each unmarked adjacent vertex u:

**set edgeTo[u] = v**

dfs(u)

| # | marked | edgeTo |
|---|---|---|
| 0 | F | - |
| 1 | F | - |
| 2 | F | - |
| 3 | F | - |
| 4 | F | - |
| 5 | F | - |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

## Breadth First Search




Order: 0 1 24 53 68 7

# BFS pseudo code

- Iterative BFS traversal

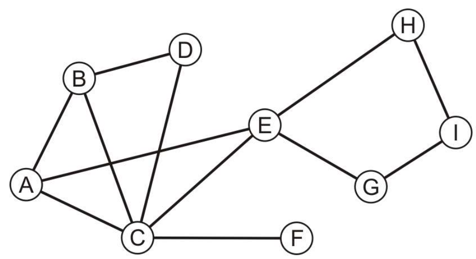
```
bfs(v)
  Queue q
  enqueue(q, v)
  mark v as visited
  while (!isEmpty(q)) {
    w = dequeue(q)
    for (each unvisited vertex u adjacent to w) {
            mark u as visited
            enqueue(q, u)
            set edgeTo[u] = v

    }
  }
```

| # | marked | edgeTo |
|---|---|---|
| 0 | F | - |
| 1 | F | - |
| 2 | F | - |
| 3 | F | - |
| 4 | F | - |
| 5 | F | - |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

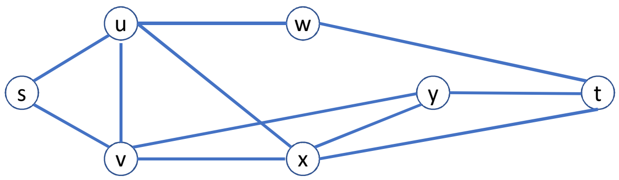
# Quiz

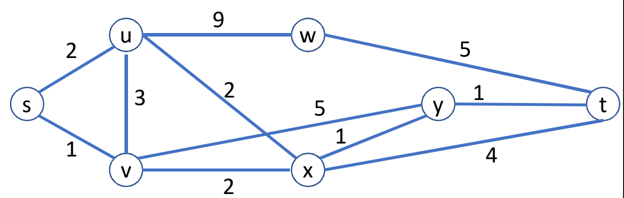
Find the route from A to G by using DFS, BFS. We assume that with the nodes at the same level, we should expand the successor according to the **alphabetical order**.



# Shortest path problem

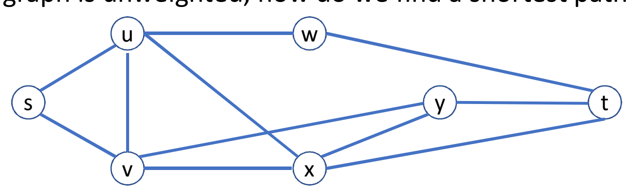
- Given: a graph G and vertices s and t
- Find: the path which has the shortest length from s to t.
- The length of a path is:
  - the number of edges on that path in unweighted graph
  - the sum of the edge weights on that path in weighted graph
    - Finding the path whose f(t) = $w_1$ + $w_2$ + … + $w_n$ is minimum with n is the number of edges from s to t




# Unweighted directed graph: shortest paths

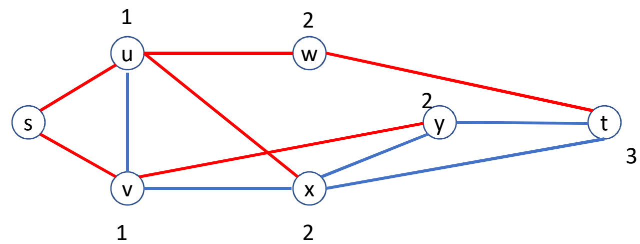
- If the graph is unweighted, how do we find a shortest path?




- What's the shortest path from s to s?
  - Well….we're already there.
- What's the shortest path from s to u or v?
  - Just go on the edge from s
- From s to t?
  - We need to find a path with minimum number of edges => we already know the algorithm the does something like this.

# Unweighted directed graph: shortest paths

- Use BFS to find shortest paths in this graph




```
bfsShortestPaths(graph G, vertex source)
   Queue Visited;
   enqueue(Visited,source)
   mark source as visited
   while(!isEmpty(Visited)){
      current = dequeue(Visited)
      for (each unvisited vertex NB adjacent to current)
       {
           mark NB as visited
           enqueue(Visited, NB)
           set edgeTo[NB] = current

       }
   }
```

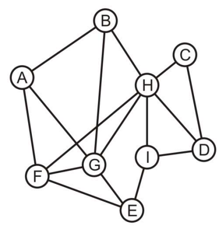
# Comments

- BFS didn't mention a target vertex, we can stop when we visit target vertex.
- It actually finds the shortest path from s to every vertex.

- Note: BFS can be applied to unweight directed graph to find the shortest path.

## Quiz

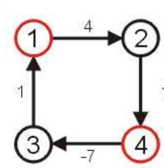
- Find the shortest path from A to other vertices




## Weighted Graphs

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

- Clearly, if we have negative vertices, 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...
  - First circle is -1, second circle is -2,….

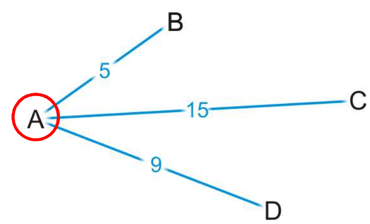



- BFS is not applied to weighted graphs
- Dijkstra’s algorithm is a good choice for this case.

## Strategy

Suppose you start at vertex A and A only have three neighbours B, C, and D (the following is a part of weighted graph)

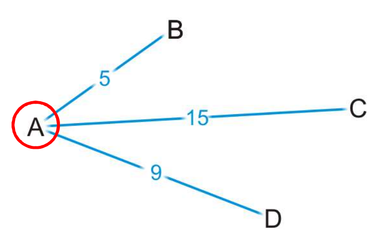
- You are aware of all vertices adjacent to it
- This information is either in an adjacency list or adjacency matrix




## Strategy

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

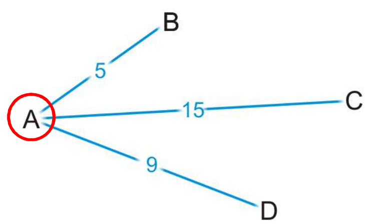
- Why or why not?




At this time:
f(B) = 5
f(C) = 15
f(D) = 9

## Strategy

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

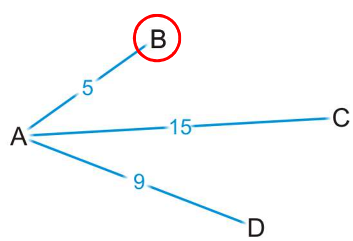



At this time:
f(B) = 5
f(C) = 15
f(D) = 9

## Strategy

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

- Let's see where we can go from B



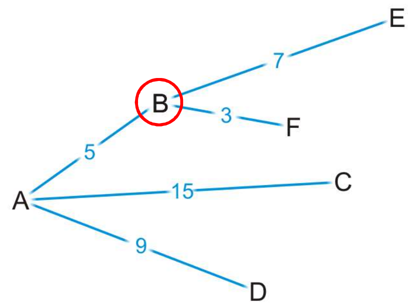

At this time:
f(B) = 5
f(C) = 15
f(D) = 9

## Strategy

By some simple arithmetic, we can determine that

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

Previous time:
f(B) = 5
f(C) = 15
f(D) = 9

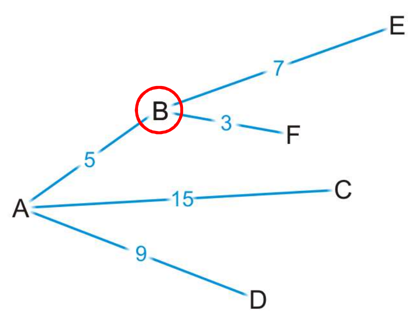



At this time:
f(B) = 5
f(C) = 15
f(D) = 9
f(E) = 12
f(F) = 8

## Strategy

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

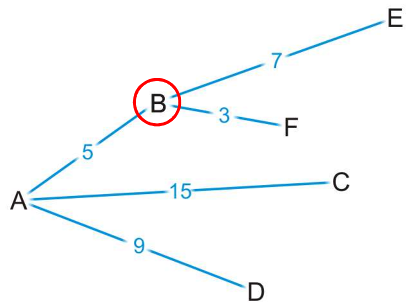
- Why or why not?




At this time:
f(B) = 5
f(C) = 15
f(D) = 9
f(E) = 12
f(F) = 8

# Strategy

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

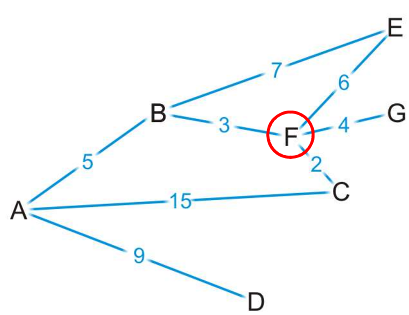



At this time:
f(B) = 5
f(C) = 15
f(D) = 9
f(E) = 12
f(F) = 8

# Strategy

Okay, let's visit vertex F

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

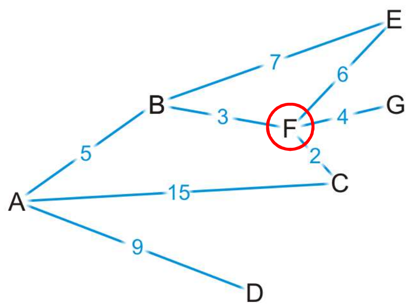



At this time:
f(B) = 5
f(C) = 15
f(D) = 9
f(E) = 12
f(F) = 8

# Strategy

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

- (A, B, F, E) of length 8 + 6 = 14. Do we need to remember this path? No
- (A, B, F, G) of length 8 + 4 = 12. => new node => update table
- (A, B, F, C) of length 8 + 2 = 10. => this cost is better => update table

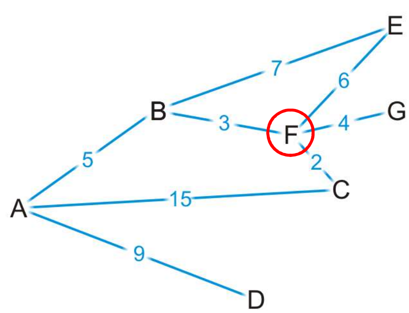



| Previous time: | At this time: |
|---|---|
| f(B) = 5 | f(B) = 5 |
| f(C) = 15 | f(C) = 10 (updated) |
| f(D) = 9 | f(D) = 9 |
| f(E) = 12 | f(E) = 12 (no updated) |
| f(F) = 8 | f(F) = 8 |
| | f(G) = 12 |

# 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



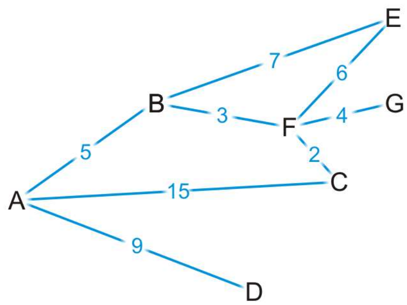

At this time:
f(B) = 5
f(C) = 10
f(D) = 9
f(E) = 12
f(F) = 8
f(G) = 12

# 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?

D



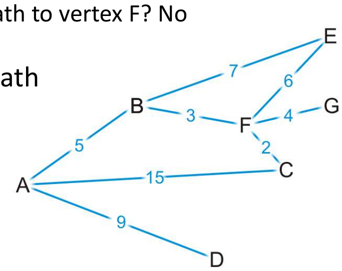

At this time:
f(B) = 5
f(C) = 10
f(D) = 9
f(E) = 12
f(F) = 8
f(G) = 12

# Dijkstra’s algorithm

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

- We require an array of distances, all initialized to infinity except for the source vertex, which is initialized to 0
- 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
  - Thus, we need an array of previous vertices, all initialized to null



# Dijkstra's algorithm

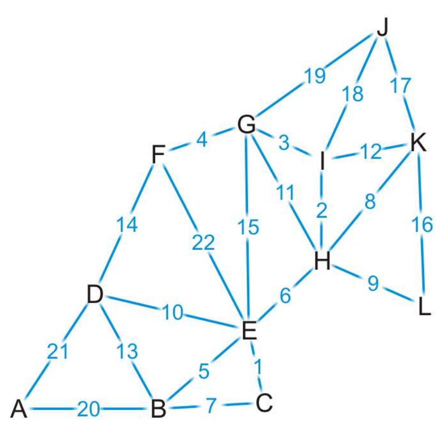
Thus, we will iterate $|V|$ times:

- Find that unvisited vertex $\boldsymbol{v}$ that has a minimum distance to it
- Mark it as having been visited
- Consider every adjacent vertex $\boldsymbol{w}$ that is unvisited:
  - Is the distance to $\boldsymbol{v}$ plus the weight of the edge $(\boldsymbol{v}, \boldsymbol{w})$ less than our currently known shortest distance to $\boldsymbol{w}$
  - If so, update the shortest distance to $\boldsymbol{w}$ and record $\boldsymbol{v}$ as the previous pointer

- Continue iterating until all vertices are visited or all remaining vertices have a distance to them of infinity

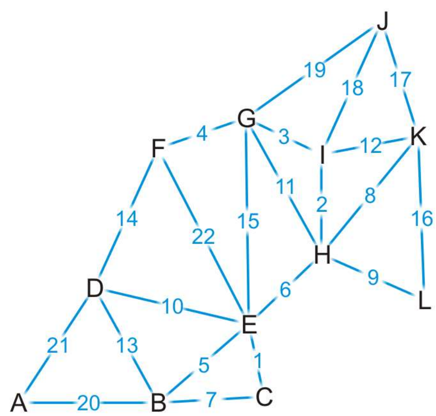
# Example

Let us give a weighted graph



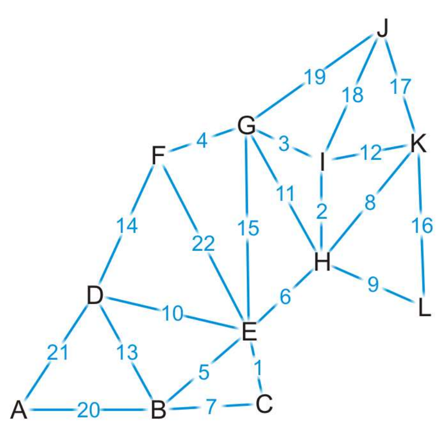
# Example

Find the shortest distance from K to every vertex.




# Example

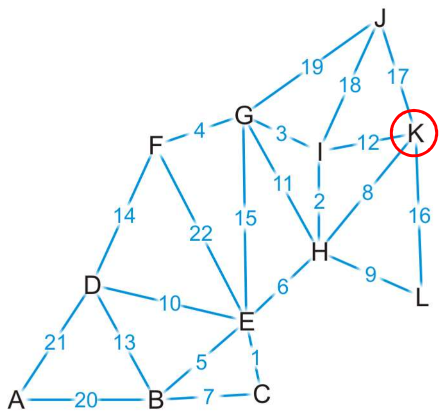
We set up our table




| Vertex | Visited | Distance (or f()) | 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

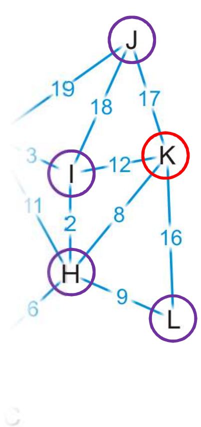
We visit vertex K




| 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

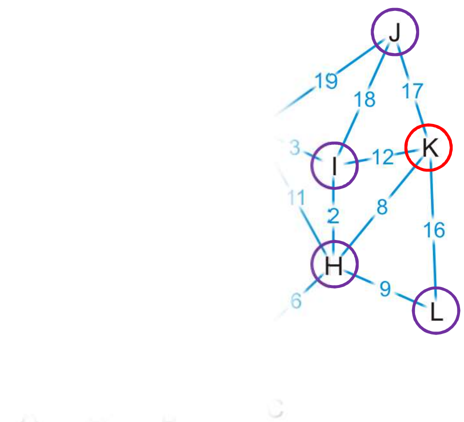
Vertex K has four neighbors:  H, I, J and L




| 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

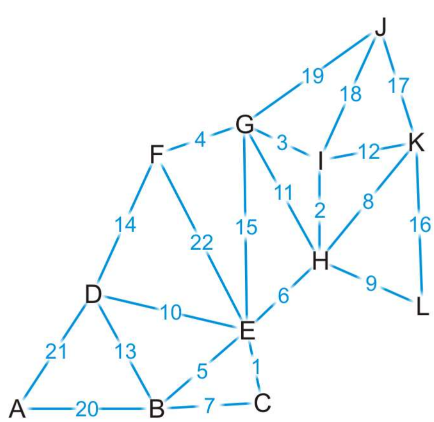



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | ∞ | Ø |
| F | F | ∞ | Ø |
| G | F | ∞ | Ø |
| **H** | F | **8** | **K** |
| **I** | F | **12** | **K** |
| **J** | F | **17** | **K** |
| **K** | T | **0** |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

# Example

We're finished with vertex K

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



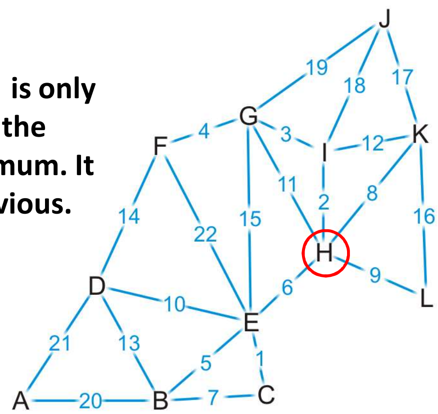

| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | ∞ | Ø |
| F | F | ∞ | Ø |
| G | F | ∞ | Ø |
| H | F | 8 | K |
| I | F | 12 | K |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# Example

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

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

**Visited status of vertex A  is only turn to T if we guarantee the distance (or f (A)) is minimum. It is very different with previous.**




| 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 |

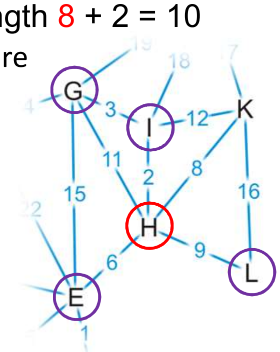
# Example

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

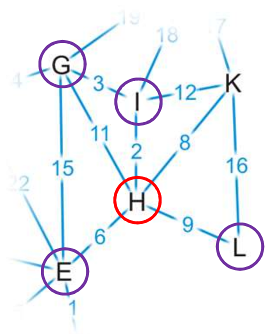
- Which of these are shorter than any known path?




| 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

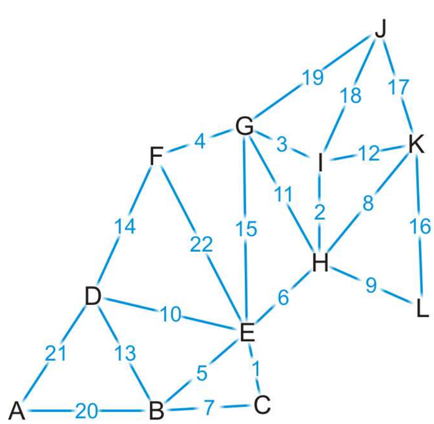



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | **14** | **H** |
| F | F | ∞ | Ø |
| G | F | **19** | **H** |
| **H** | **T** | **8** | **K** |
| I | F | **10** | **H** |
| J | F | 17 | K |
| K | T | 0 | Ø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

# Example

We are finished with vertex H

- Which vertex do we visit next?

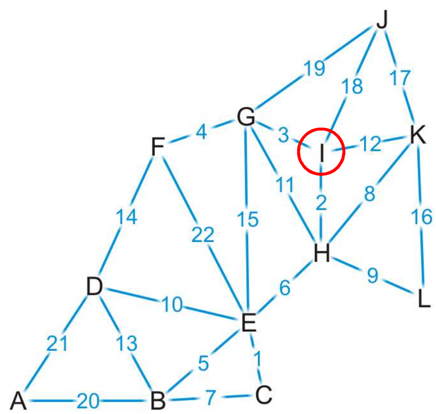



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | 14 | H |
| F | F | ∞ | Ø |
| G | F | 19 | H |
| H | T | 8 | K |
| I | F | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# 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

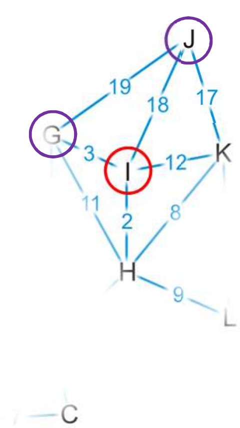



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | 14 | H |
| F | F | ∞ | Ø |
| G | F | 19 | H |
| H | T | 8 | K |
| **I** | **T** | **10** | **H** |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# Example

Consider these paths:

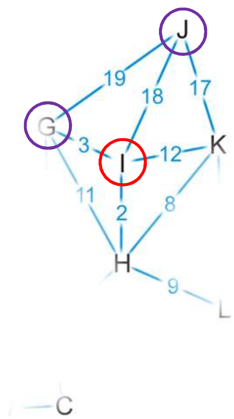
(K, H, I, G) of length 10 + 3 = 13 (K, H, I, J) of length 10 + 18 = 28




| 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

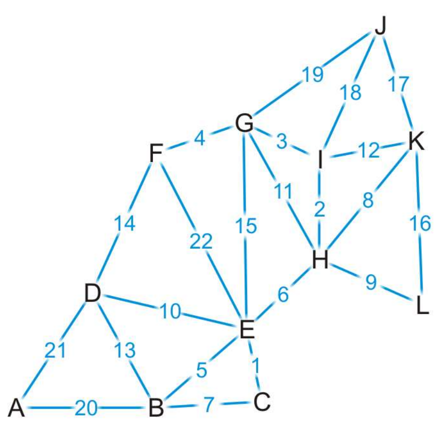
We have discovered a shorter path to vertex G, but (K, J) is still the shortest known path to vertex J




| 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?

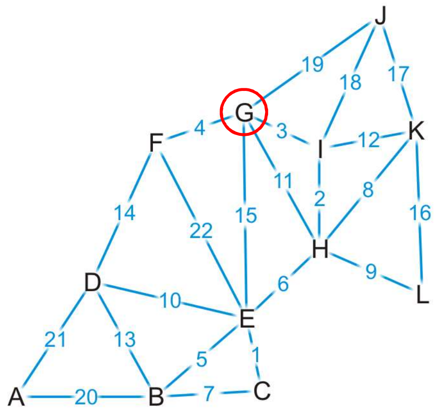



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | 14 | H |
| F | F | ∞ | Ø |
| G | F | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# 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 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 |

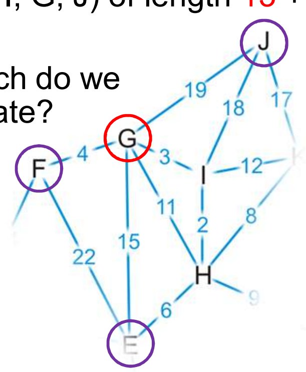
# Example

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

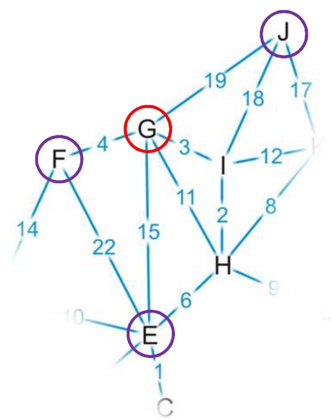
- Which do we update?




| 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

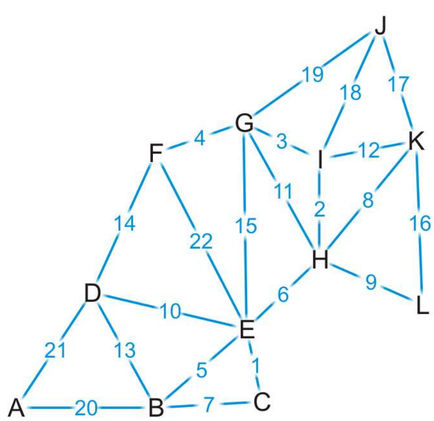
We have now found a path to vertex F




| 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?



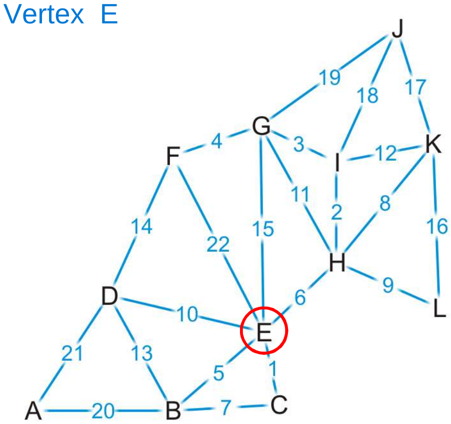

| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| E | F | 14 | H |
| F | F | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# 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 E



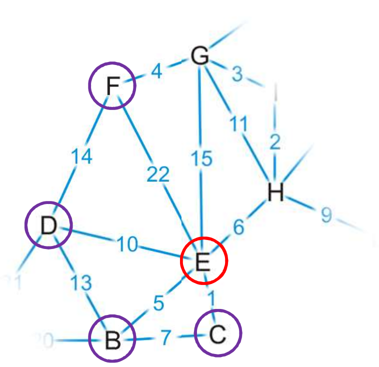

| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | ∞ | Ø |
| C | F | ∞ | Ø |
| D | F | ∞ | Ø |
| **E** | **T** | **14** | **H** |
| F | F | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# 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 E




| 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:

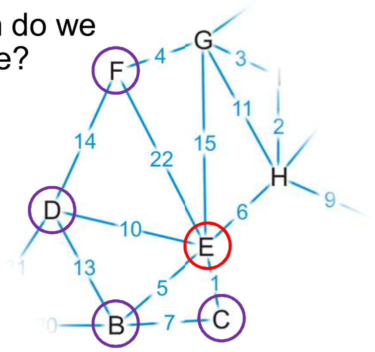
(K, H, E, B) of length 14 + 5 = 19

(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

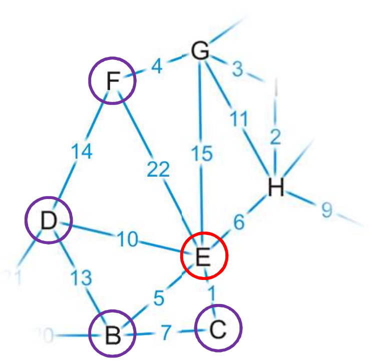
- Which do we update?




| 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

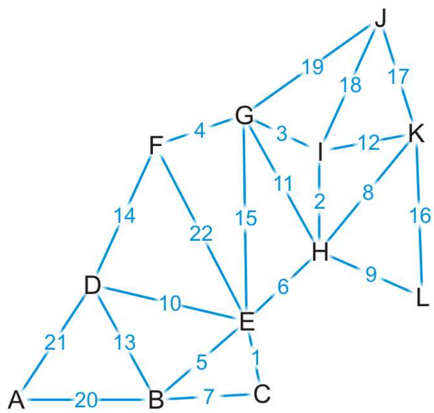
We’ve discovered paths to vertices B, C, D




| 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?

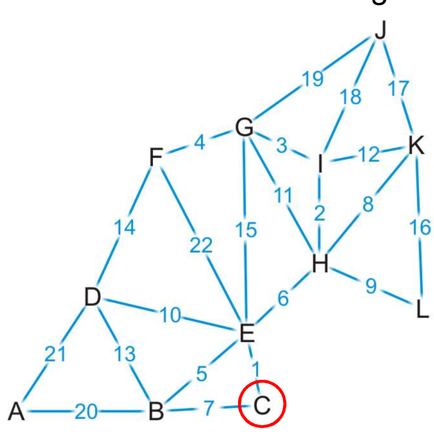



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | 19 | E |
| C | F | 15 | E |
| D | F | 24 | E |
| E | T | 14 | H |
| F | F | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# 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

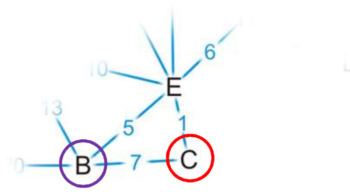



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | 19 | E |
| **C** | **T** | **15** | **E** |
| D | F | 24 | E |
| E | T | 14 | H |
| F | F | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# Example

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

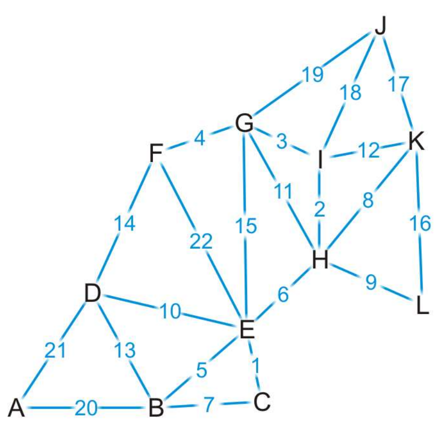
- We have already discovered a shorter path through vertex E




| 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?

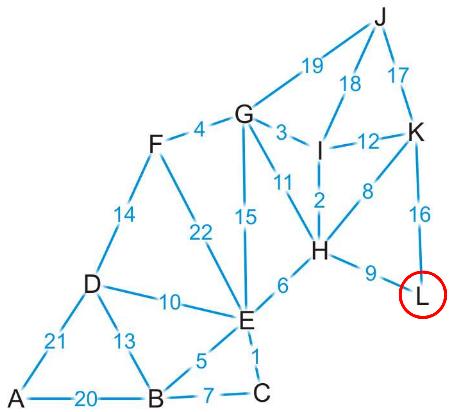



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | 19 | E |
| C | T | 15 | E |
| D | F | 24 | E |
| E | T | 14 | H |
| F | F | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | F | 16 | K |

# Example

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

- Vertex L has no unvisited neighbors

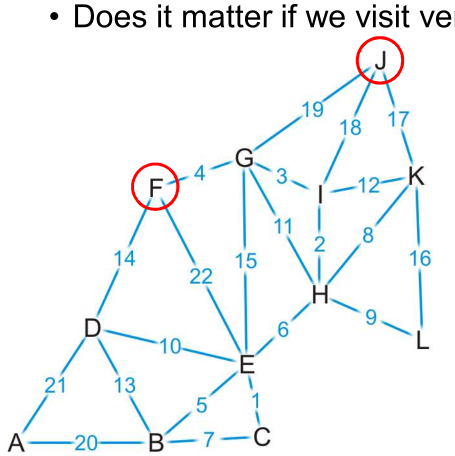



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | 19 | E |
| C | T | 15 | E |
| D | F | 24 | E |
| E | T | 14 | H |
| F | F | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| **L** | **T** | **16** | **K** |

# Example

Where to next?

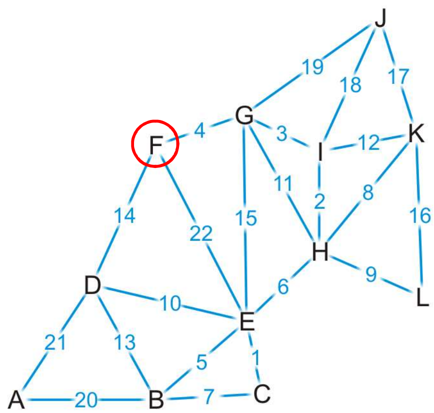
- Does it matter if we visit vertex F first or vertex J first?




| 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

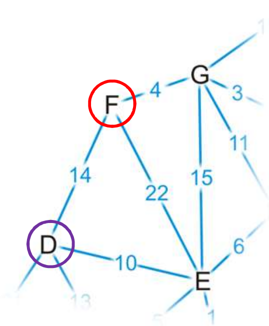
Let's visit vertex F first
- It has one unvisited neighbor, vertex D




| 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

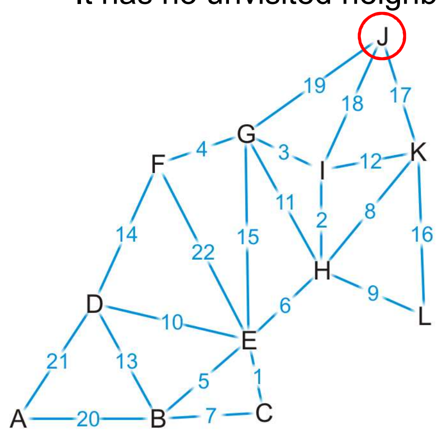



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | 19 | E |
| C | T | 15 | E |
| D | F | 24 | E |
| E | T | 14 | H |
| **F** | **T** | **17** | **G** |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | F | 17 | K |
| K | T | 0 | Ø |
| L | T | 16 | K |

# Example

Now we visit vertex J

- It has no unvisited neighbors



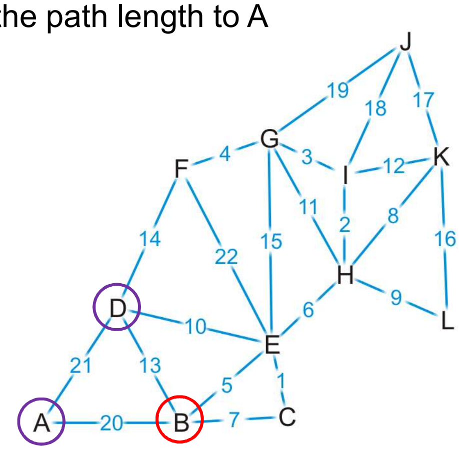

| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | ∞ | Ø |
| B | F | 19 | E |
| C | T | 15 | E |
| D | F | 24 | E |
| E | T | 14 | H |
| F | T | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| **J** | **T** | **17** | **K** |
| K | T | 0 | Ø |
| L | T | 16 | K |

# 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

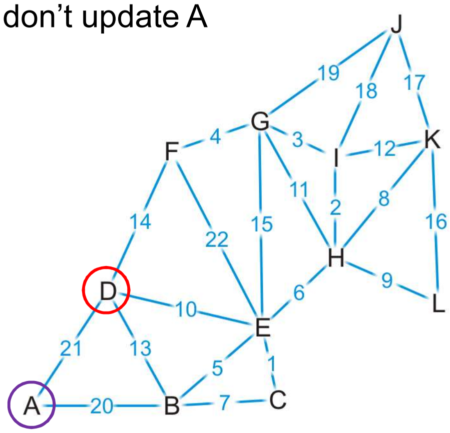



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | **39** | **B** |
| **B** | **T** | **19** | **E** |
| C | T | 15 | E |
| D | F | 24 | E |
| E | T | 14 | H |
| F | T | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | T | 17 | K |
| K | T | 0 | Ø |
| L | T |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

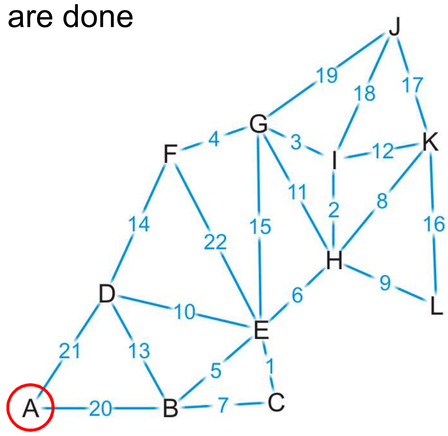



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | F | 39 | B |
| B | T | 19 | E |
| C | T | 15 | E |
| **D** | **T** | **24** | **E** |
| E | T | 14 | H |
| F | T | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | T | 17 | K |
| K | T | 0 | Ø |
| L | T | 16 | K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

# Example

Finally, we visit vertex A

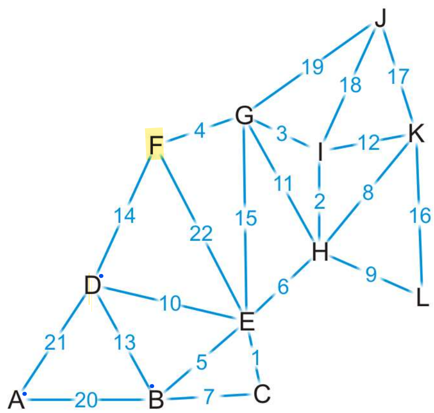
- It has no unvisited neighbors and there are no unvisited vertices left
- We are done




| 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

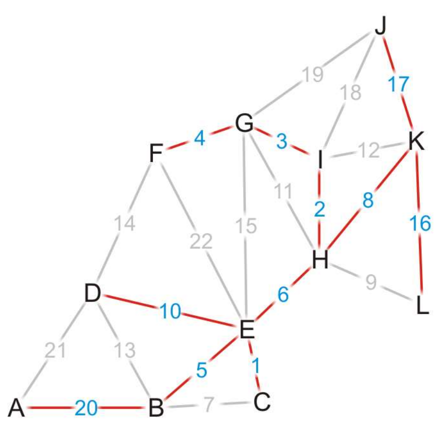
Thus, we have found the shortest path from vertex K to each of the other vertices




| 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

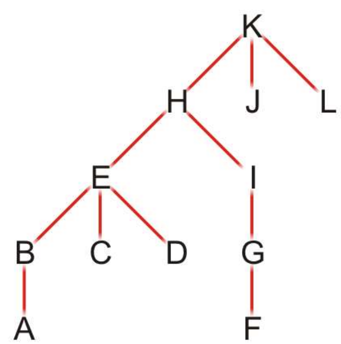



| Vertex | Visited | Distance | Previous |
|---|---|---|---|
| A | T | 39 | B |
| B | T | 19 | E |
| C | T | 15 | E |
| D | T | 24 | E |
| E | T | 14 | H |
| F | T | 17 | G |
| G | T | 13 | I |
| H | T | 8 | K |
| I | T | 10 | H |
| J | T | 17 | K |
| K | T | 0 | Ø |
| L | T | 16 | K |

# Example

Note that this table defines a rooted parental tree

- 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 |

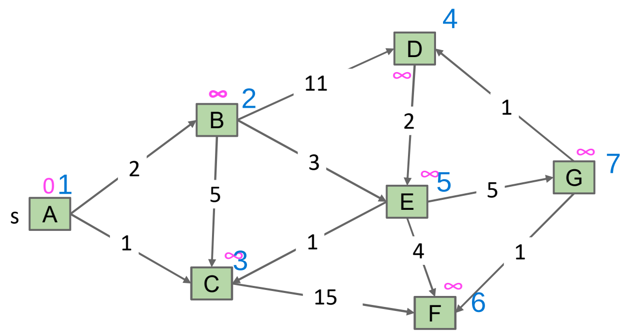
# Comments on Dijkstra's algorithm

Questions:

- What if at some points, all unvisited vertices have a distance $\infty$?
  - 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 $v_k$ (or visited status is turned on **T**)
- Does the algorithm change if we have a directed graph?
  - No

# Quiz

- Find the shortest path from A to other vertices using Dijkstra’s algorithm




7
2 2 2 3 1
3 3 5 4 11 5 3
1 6 15
1 5 2
3 3 1 6 4 7 5
0
2 4 1 6 1

# 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?
  - Assume we are using a binary heap
  - We will have to update the heap structure—this requires additional work

# 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

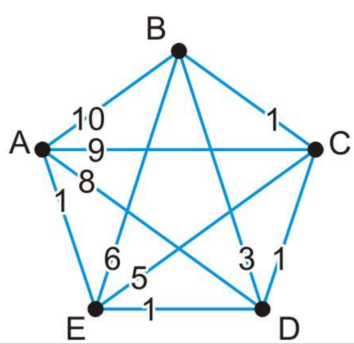
We iterate $|V|$ times, each time finding the *closest* vertex to the source
- Place the distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Thus, the work required for this is $O(|V| \ln(|V|))$

Is this all the work that is necessary?
- Recall that each edge visited may result in a new edge being pushed to the very top of the heap
- 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



## 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

## 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 $O(|E| + |V| \ln(|V|))$ time

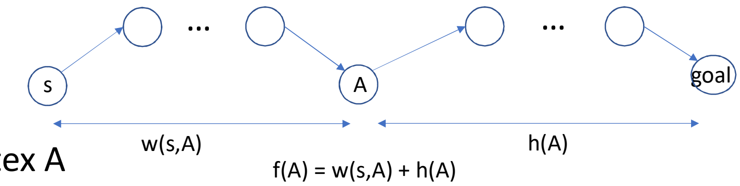
# Disadvantages of Dijkstra's algorithm

- Single target:
  - Is this a good algorithm for a navigation application? Dijkstra will explore every place within nearly two thousand miles of Denver before it locates NYC.
  - We have only a ***single target*** in mind, so we need a different algorithm. How can we do better?


# How can we do Better?

- Explore eastwards first?

# A* Idea




- Re-define cost function at a vertex A
  - f(A) =  W(s, A) + h(A, dest)

Where:

- W(s, A): the sum of weights from source s to A (as Dijkstra's Algorithms)
- h(A, dest): the **estimated weights (or cost)** from A to **destination**
  - h(A) is a heuristic function collected from experience or from other sources.
  - We need to know this information before finding the shortest path => informed search

# A* Algorithm

Insert all vertices into Priority Queue (PQ), storing the vertices in order of [W(s, A) + h(A, dest)].

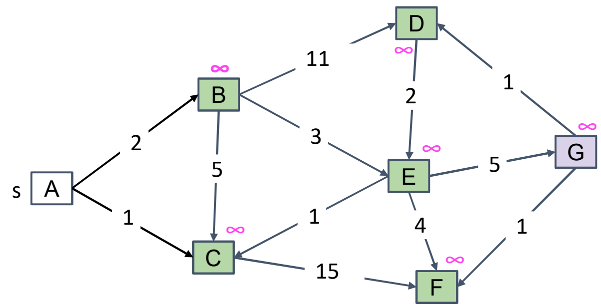
Repeat: Visit and remove best vertex A (or **f(A) is minimum**) from PQ, and relax all edges pointing from A.

=>We choose to visit a vertex A **if f(A) is minimum => We take care both: cost from source to A and cost from A to destination.**

# Example

- Given the directed graph with the following heuristic function. Find the shortest path from A to G

```
Node    h(v, G)
A       18
B       3
C       15
D       1
E       1
F       ∞
G       0
```




```
PQ: [(A: 18), (B: ∞), (C: ∞), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]
```

Of course, we are at A and remove (A: 18)

# Example

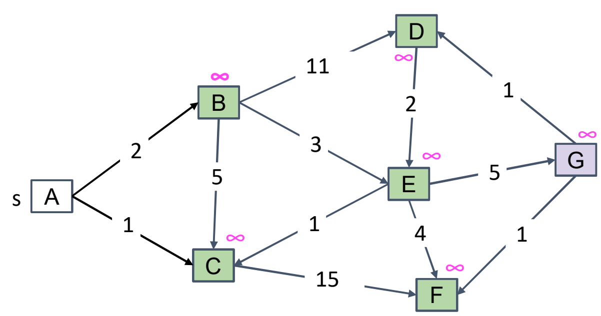
```
Node    h(v, G)
A       18
B       3
C       15
D       1
E       1
F       ∞
G       0
```

```
PQ: [(B: ∞), (C: ∞), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]
```

Explore A's neighbors:

1. B => f(B) = 2 + 3 (update)
2. C => f(C) = 1 + 15 (update)

| Vertex | f= w+h | Path |
|---|---|---|
| A | **18** | Ø |
| B | **2+3** | AB |
| C | **1+15** | AC |
| D | ∞ | Ø |
| E | ∞ | Ø |
| F | ∞ | Ø |
| G | ∞ | Ø |




```
PQ: [(B: 5), (C: 16), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]
```

PQ: [(B: 5), (C: 16), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Visit B, remove (B:5) in PQ

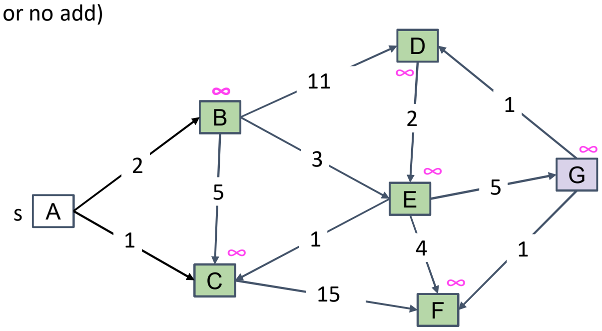
PQ: [(C: 16), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

| Node | h(v, G) |
|---|---|
| A | 18 |
| B | 3 |
| C | 15 |
| D | 1 |
| E | 1 |
| F | ∞ |
| G | 0 |

Explore B's neighbors: 7 + 15

1. C => (A,B,C) => f(C) = 2+5+15= 5+15 (no updated or no add)
2. D =>(A,B,D) => f(D) = 2+11+1 = 13 +1 (updated)
3. E => (A,B,E) => f(E) = 2+3+1 = 5 + 1 (updated)

| Vertex | f=w+h | Path |
|---|---|---|
| A | 18 | Ø |
| B | 2+3 | AB |
| C | 1+15 | AC |
| D | 13+1 | ABD |
| E | 5+1 | ABE |
| F | ∞ | Ø |
| G | ∞ | Ø |




PQ: [(E: 6), (D: 15), (C: 16), (F: ∞), (G: ∞)]

PQ: [(E: 6), (D: 15), (C: 16), (F: ∞), (G: ∞)]

Visit E and remove (E: 6)

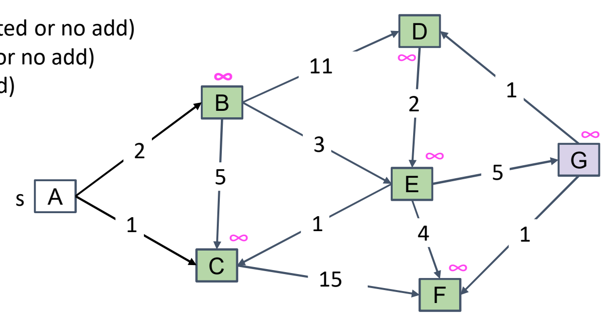
PQ: [(D: 15), (C: 16), (F: ∞), (G: ∞)]

| Node | h(v, G) |
|---|---|
| A | 18 |
| B | 3 |
| C | 15 |
| D | 1 |
| E | 1 |
| F | ∞ |
| G | 0 |

Explore E's neighbors:

1. C => (A,B,E,C) => f(C) = 2+3+1+15= 6+15 (no updated or no add)
2. F =>(A,B,E,F) => f(D) = 2+3+4+ ∞= ∞ (no updated or no add)
3. G => (A,B,E,G) => f(G) = 2+3+5+0 = 10 + 0 (updated)

| Vertex | f=w+h | Path |
|---|---|---|
| A | 18 | Ø |
| B | 2+3 | AB |
| C | 1+15 | AC |
| D | 13+1 | ABD |
| E | 5+1 | ABE |
| F | ∞ | Ø |
| G | 10+0 | ABEG |

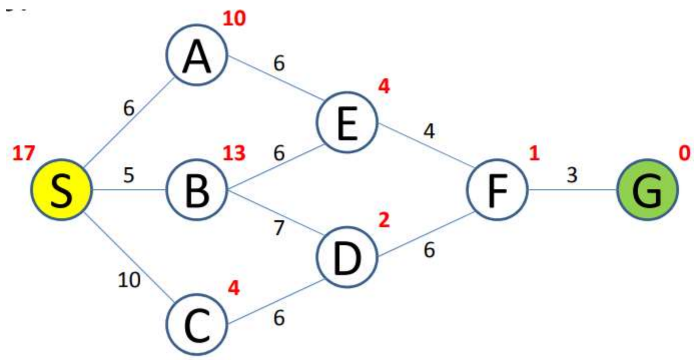



PQ: [(G: 10), (D: 15), (C: 16), (F: ∞)]

Next vertex to be dequeued is our target, so we're done! (We only stop if G is top of PQ)

# Example 2

- Given the graph, finding the shortest path from S to G with admissible heuristic function of each node to G is red number (ex. h(A,G)=10)




# Comments

- Not every vertex got visited.
- Result is not a shortest paths tree for vertex A (path to D is suboptimal!), but that's OK because we only care about path to G.
- Does A* always give us the optimal solution? Depending on heuristic function
  - heuristic: "using experience to learn and improve"
  - Doesn't have to be perfect!
  - If you choose a **bad heuristic function**, **A* can give the wrong answer**

- A* only returns a optimal solution if **h(A) is a Admissible Heuristics**
- If we don't care h(A) or h(A) = 0 => f(A) = distance(s, A) => A* is just the Dijkstra's Algorithm

## Admissible Heuristics

Admissible heuristics h must always be optimistic:

- Let w(*u*, *v*) represent the actual shortest distance from *u* to *v*
- A heuristic h(*u*, *v*) is *admissible* if h(*u*, *v*) ≤ w(*u*, *v*)
- For example, about distance of two points on a map, Straight-line distance is admissible heuristic.

How to choose a good admissible heuristic? This is an artificial intelligence topic, and is beyond the scope of our course

In the exam, admissiable heuristic function will be given.

## Graph Problem summary

| Problem | Problem Description | Solution | Efficiency |
|---|---|---|---|
| paths | Find a path from s to every reachable vertex. | DFS | O(V+E) time<br>Θ(V) space |
| shortest paths | Find the shortest path from s to every reachable vertex. | BFS | O(V+E) time<br>Θ(V) space |
| shortest weighted paths | Find the shortest path, considering weights, from s to **every reachable vertex**. | Dijkstra | O(E ln V) time<br>Θ(V) space |
| shortest weighted path | Find the shortest path, consider weights, from s to **some target vertices** | A*: Same as Dijkstra's but with h(v, goal) added to priority of each vertex. | Time depends on heuristic.<br>Θ(V) space |

## Summary

- Path Finding
  - DFS, BFS
- Shortest path of graphs
  - Unweighted graphs
    - BFS
  - Weighted graphs
    - Dijkstra's Algorithm (uninformed search)
    - A* Algorithm (informed search)
- Next week: minimum spanning tree
- Please do your homework