Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Data Structures and Algorithms
★ Graph ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

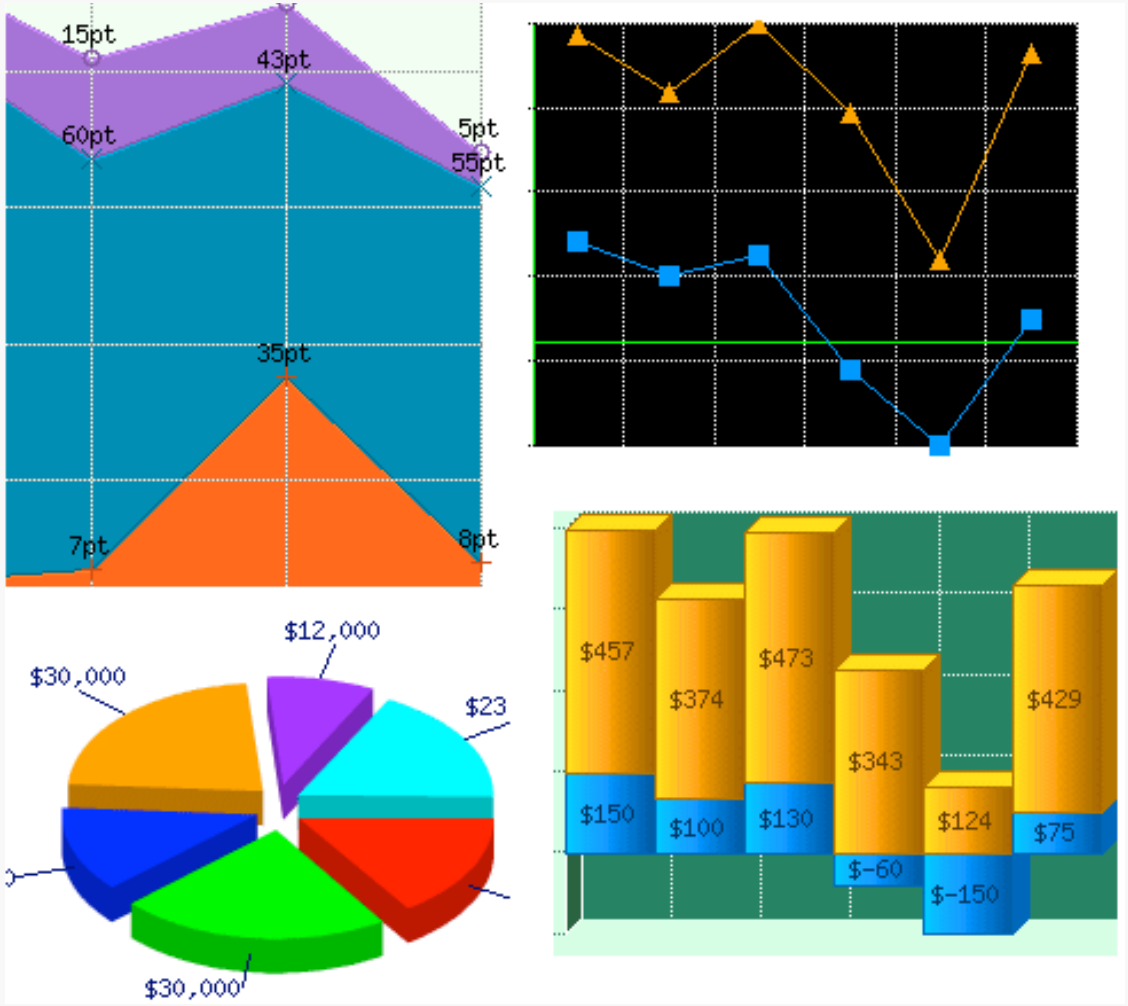
Final-Exam

10 LABS

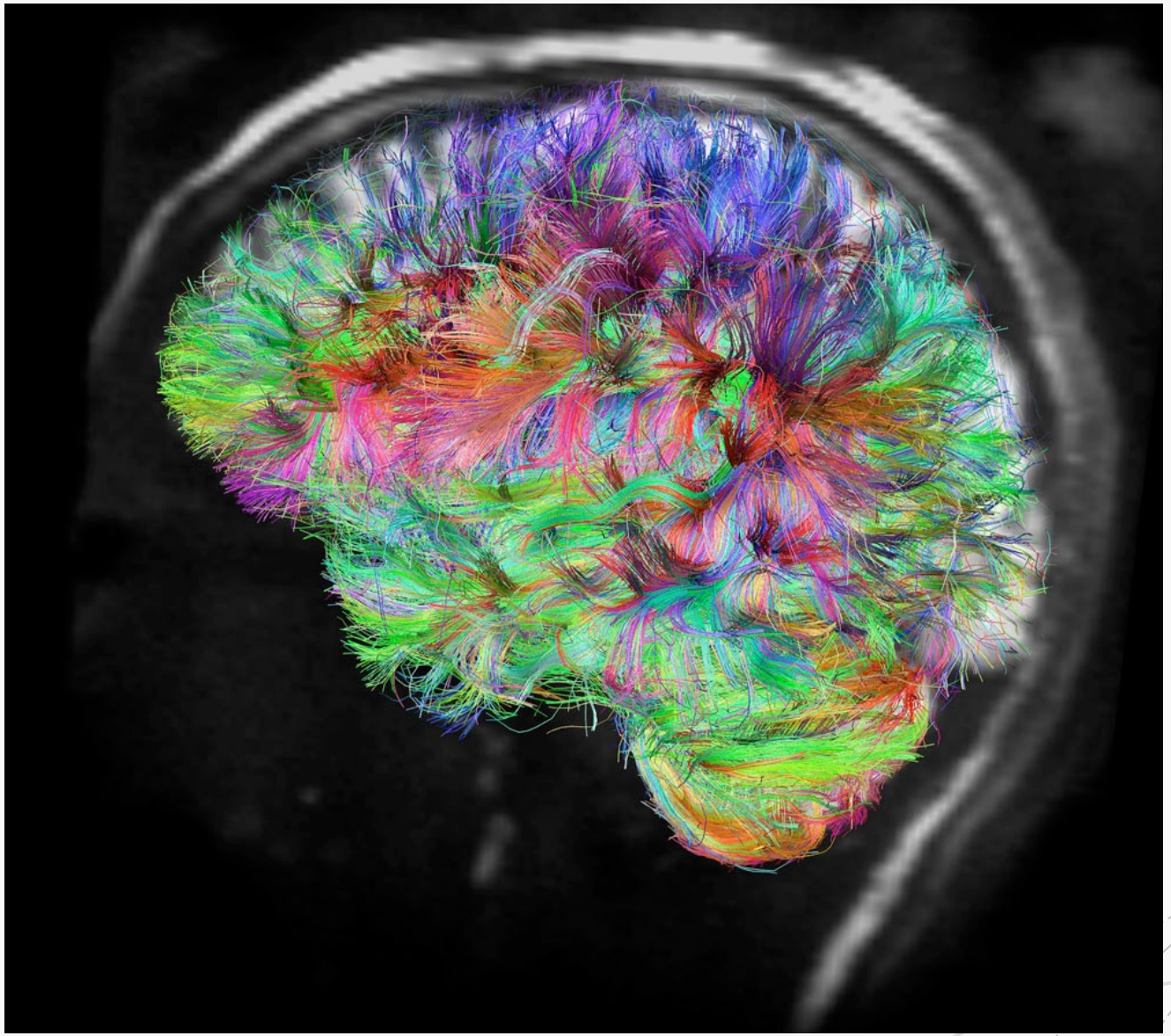
Contents

- Definition
- Implementation
- Depth First Search
- Breadth First Search
- Topological sorting
- Dijkstra Algorithm
- Connectivity Problem

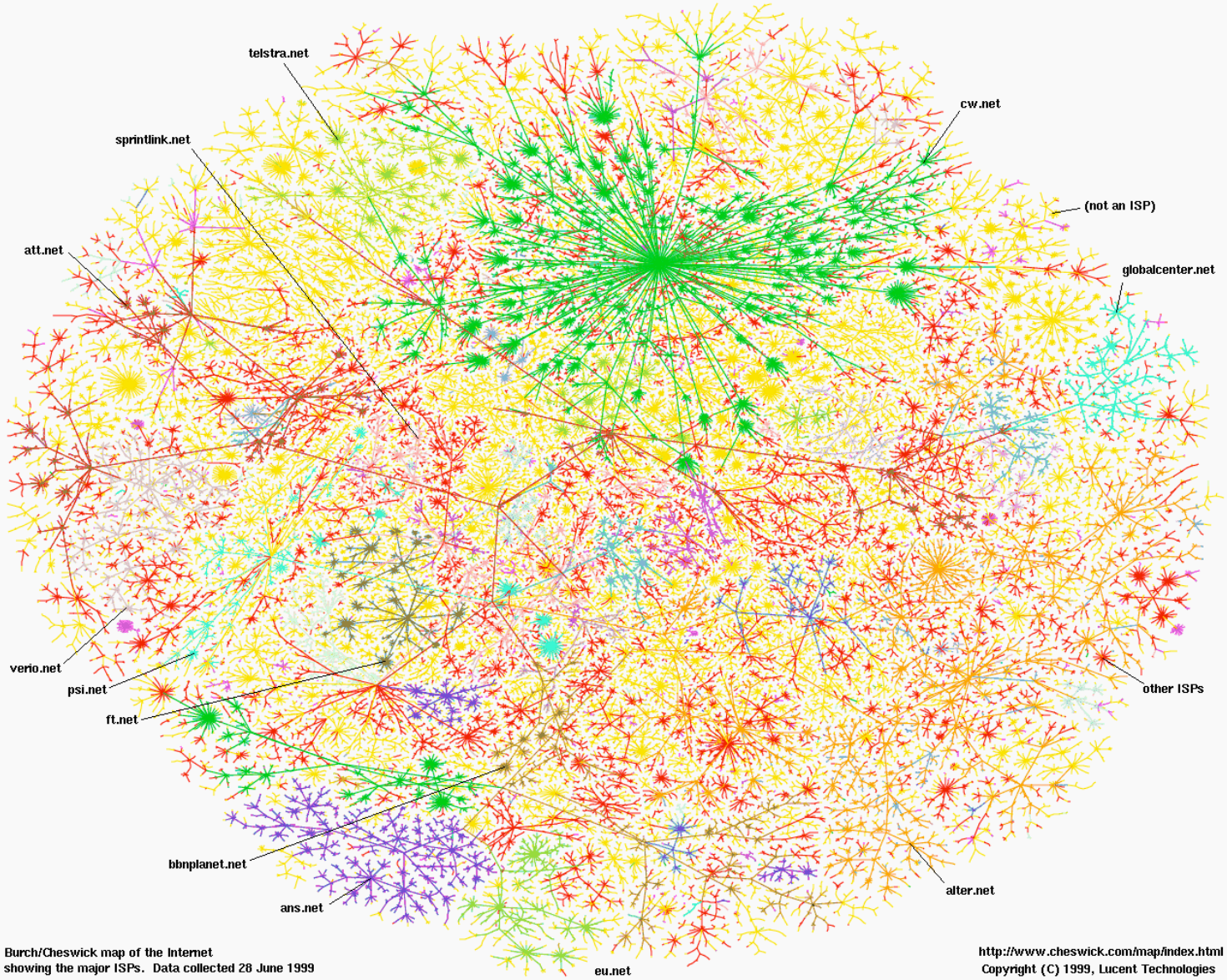
These ***aren't*** the graphs we're interested in



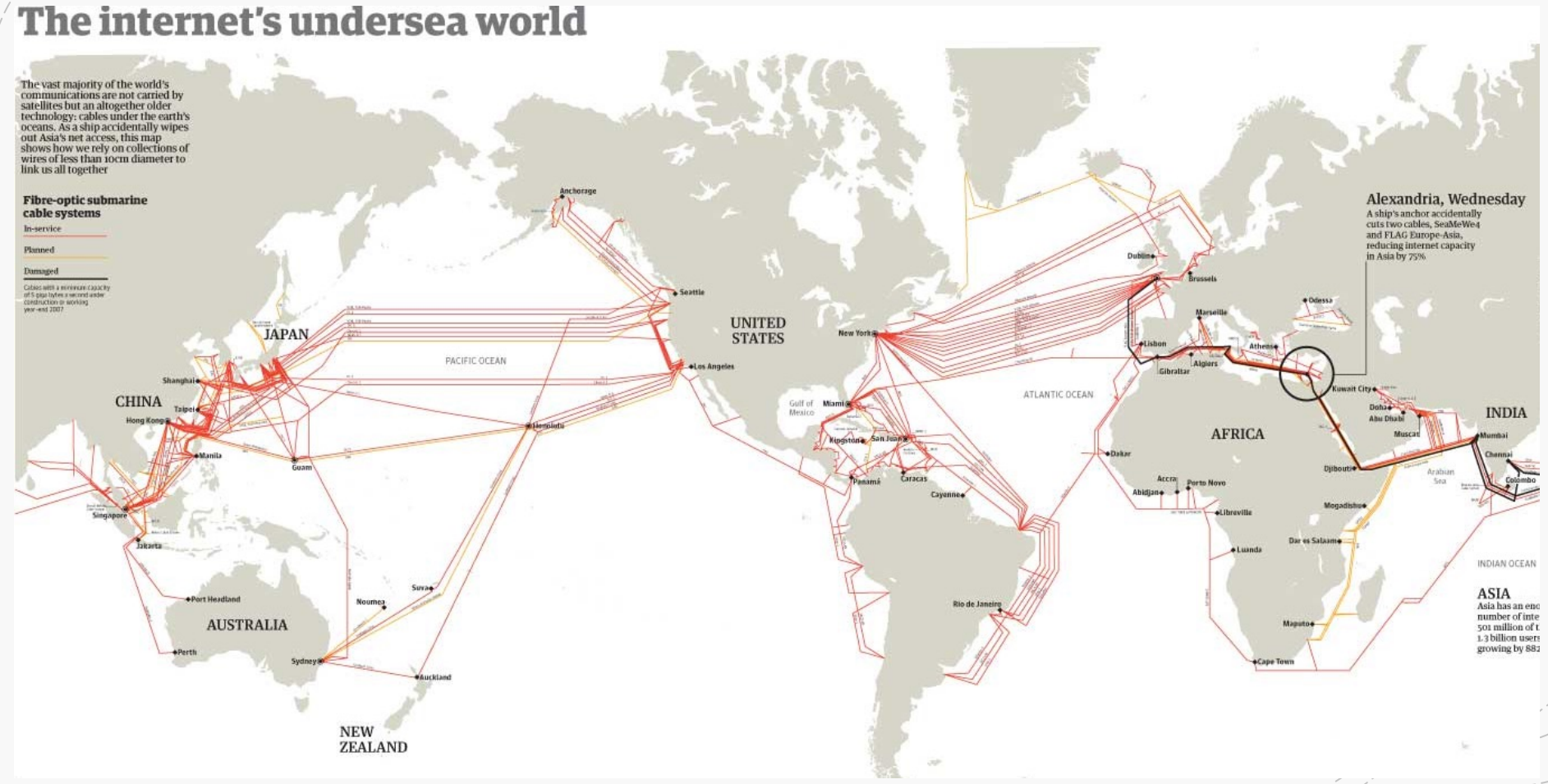
This is



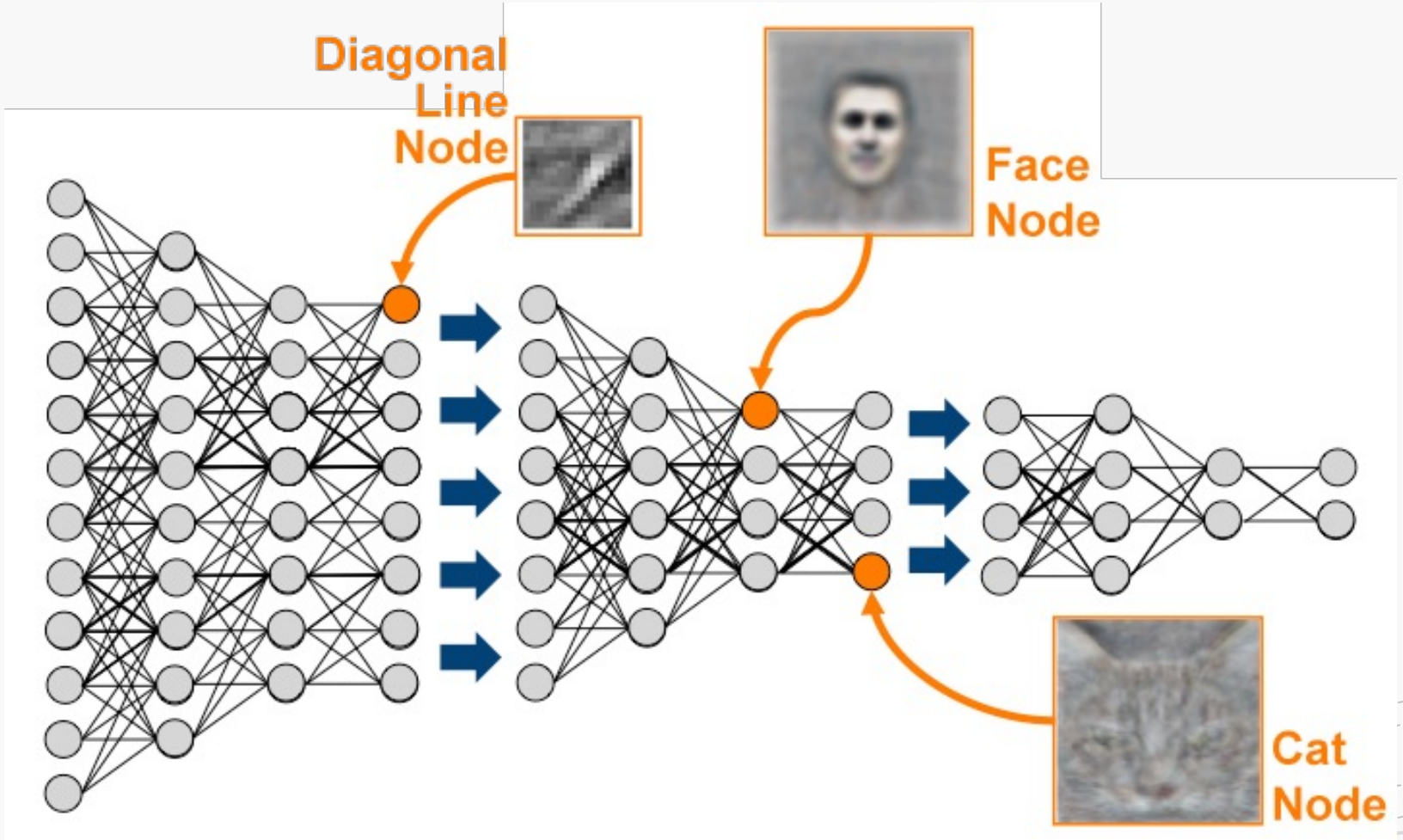
And so is this



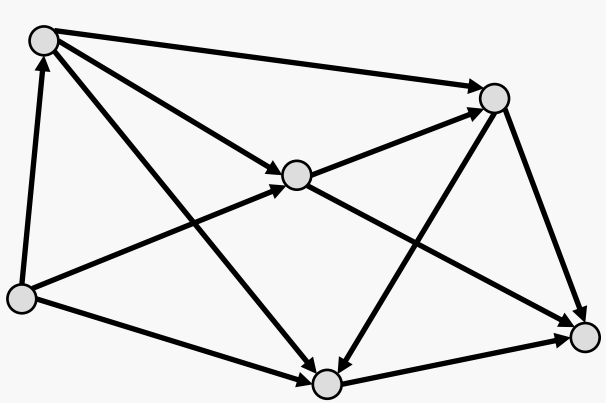
And this

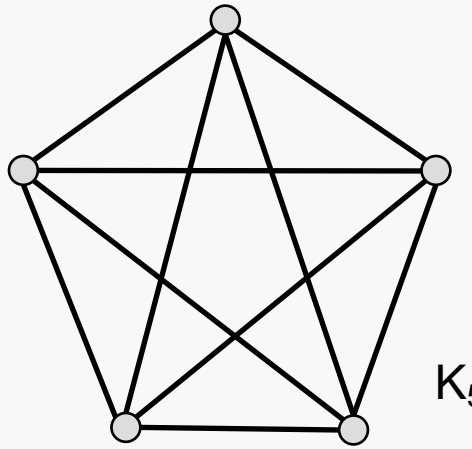

The internet's undersea world
The vast majority of the world's communications are not carried by satellites but an altogether older technology: cables under the earth's oceans. As a ship accidentally wipes out Asia's net access, this map shows how we rely on collections of wires of less than 10cm diameter to link us all together
Fibre-optic submarine cable systems
In-service
Planned
Damaged
Alexandria, Wednesday
A ship's anchor accidentally cuts two cables, SeaMeWe4 and FLAG Europe-Asia, reducing internet capacity in Asia by 75%
Anchorage
Seattle
Los Angeles
JAPAN
CHINA
Shanghai
Taipei
Hong Kong
Manila
Guam
Singapore
Jakarta
PACIFIC OCEAN
UNITED STATES
New York
Miami
Gulf of Mexico
Kingston
San Juan
Panama
Caracas
Cayenne
Rio de Janeiro
Dublin
Brussels
Marseille
Lisbon
Gibraltar
Algiers
Athens
Odessa
ATLANTIC OCEAN
AFRICA
Dakar
Accra
Abidjan
Porto Novo
Libreville
Luanda
Cape Town
Maputo
Kuwait City
Doha
Abu Dhabi
Muscat
Djibouti
Mogadishu
Dar es Salaam
INDIA
Mumbai
Chennai
Colombo
Arabian Sea
INDIAN OCEAN
ASIA
AUSTRALIA
Port Headland
Perth
Sydney
Noumea
Suva
Auckland
NEW ZEALAND

This is a graph(ical model) that has learned to recognize cats

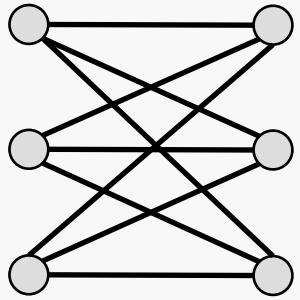


Some abstract graphs

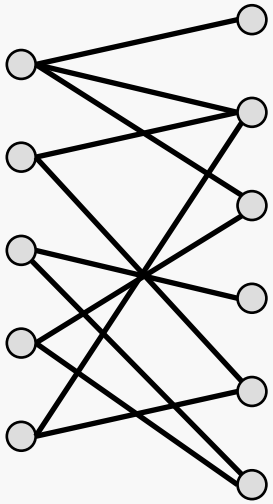


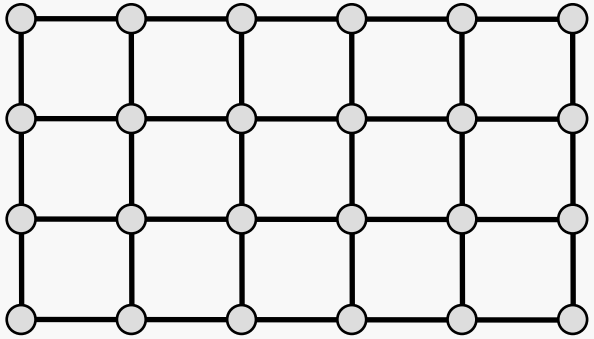


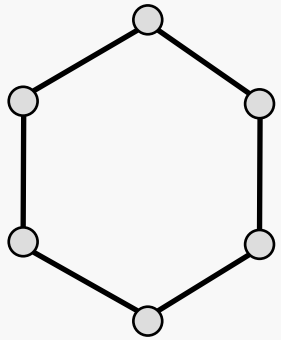
K_5



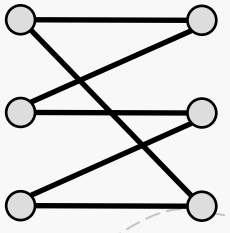
$K_{3,3}$





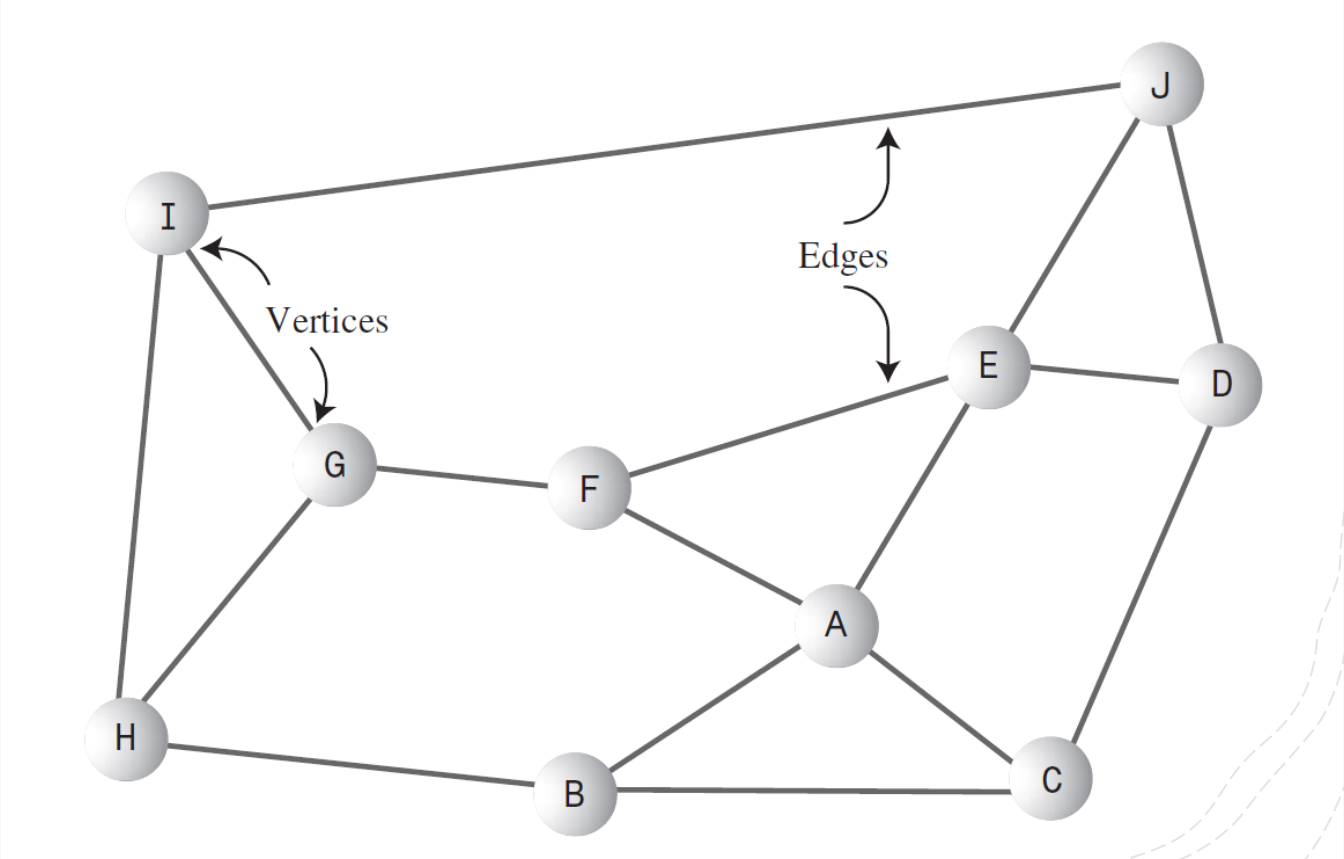


=



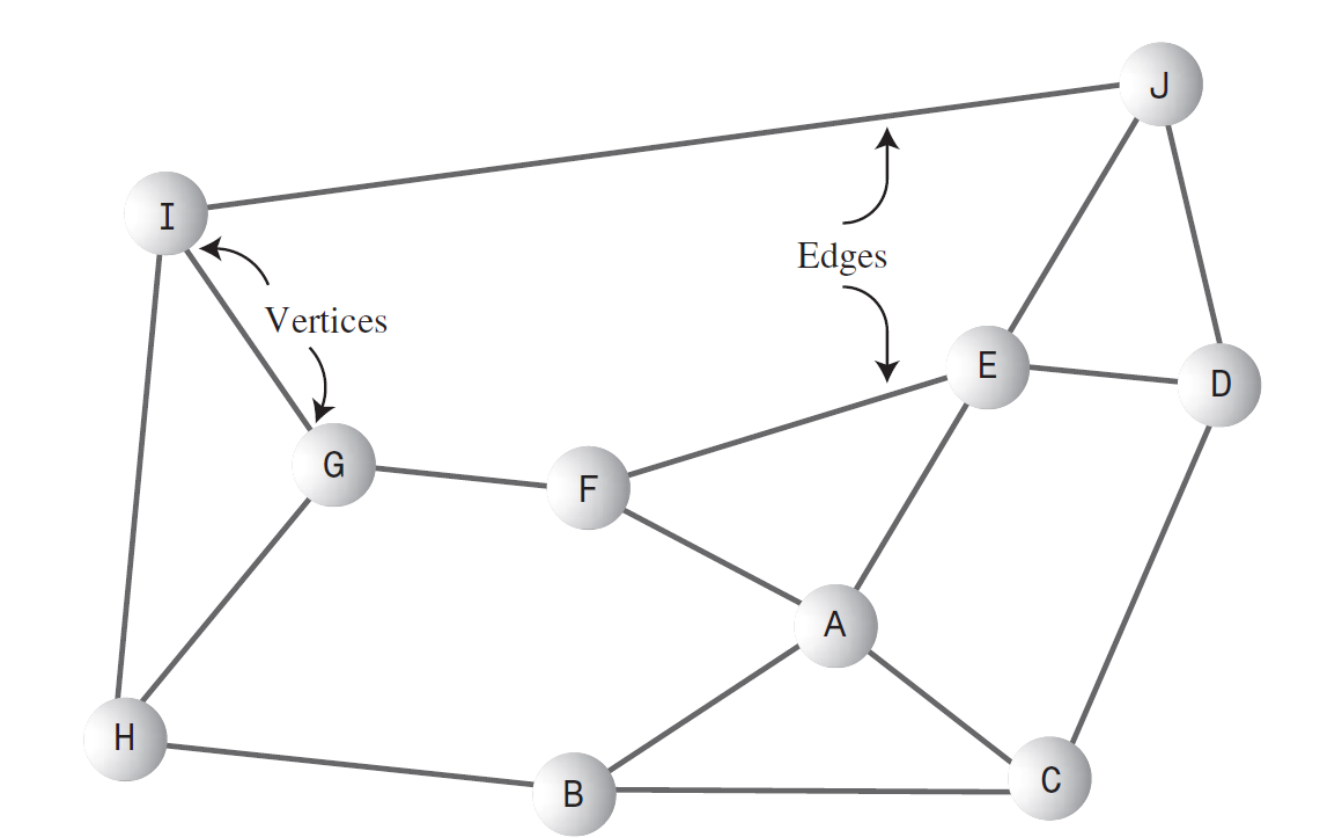
Graphs

- Graph is a data structure which represent relationships between entities
  - Vertices represent entities
  - Edges represent some kind of relationship



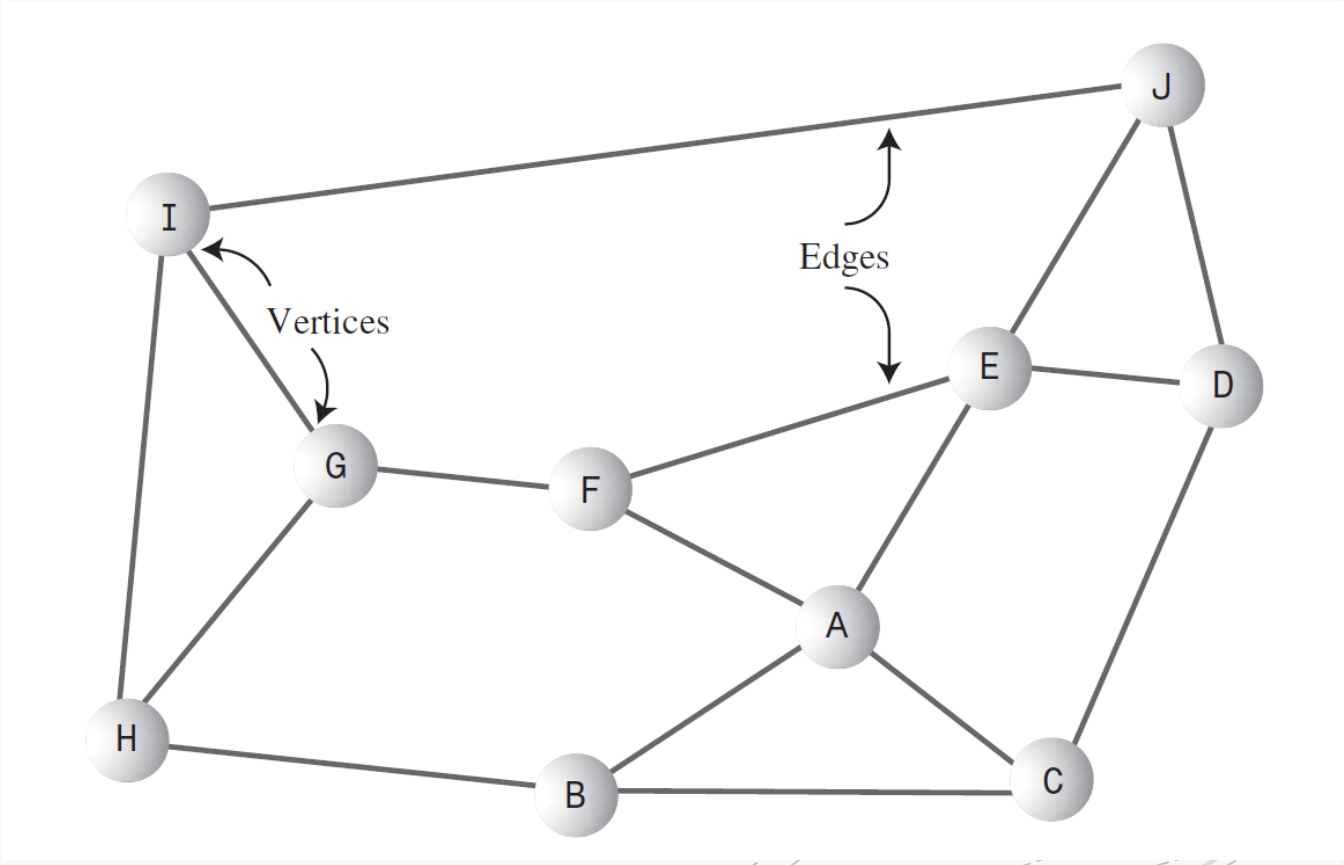
Adjacency

- Two vertices are ***adjacent*** to one another if they are connected by a single edge
- For example:
  - I and G are adjacent
  - A and C are adjacent
  - I and F are not adjacent
- Two adjacent nodes are considered neighbours



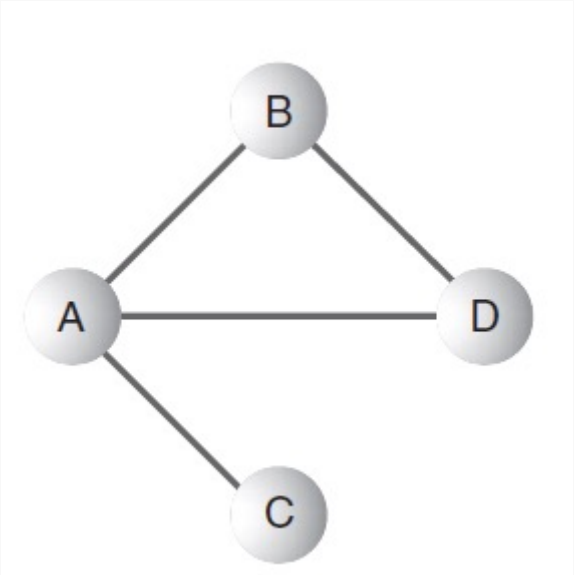
Path

- A path is a sequence of edges
- Paths in this graph include:
  - BAEJ
  - CAFG
  - HFEJDCAB
  - HIJDCBAFE
  - etc.



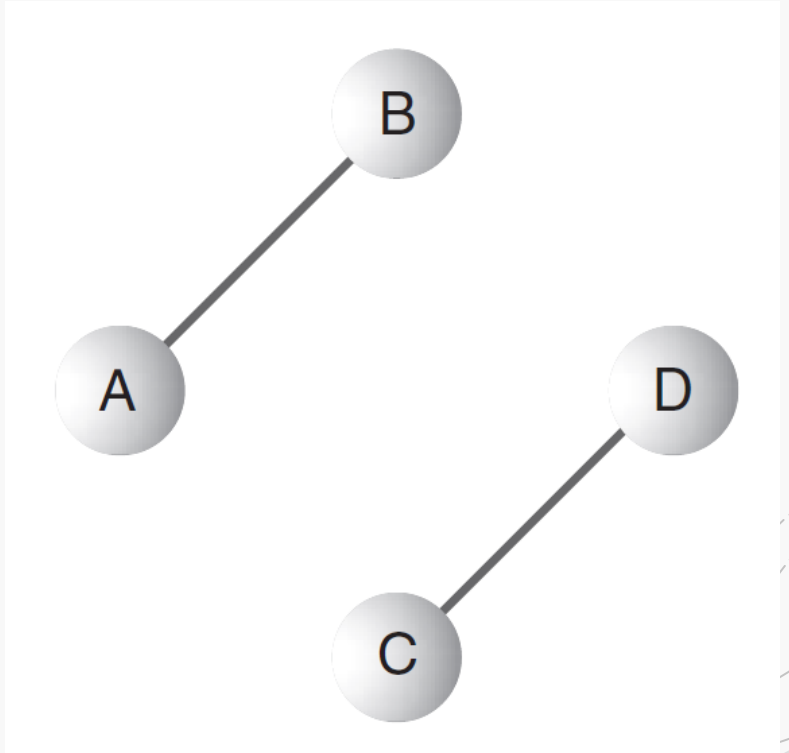
Connected Graphs

- A graph is connected if there is at least one path from every vertex to every other vertex



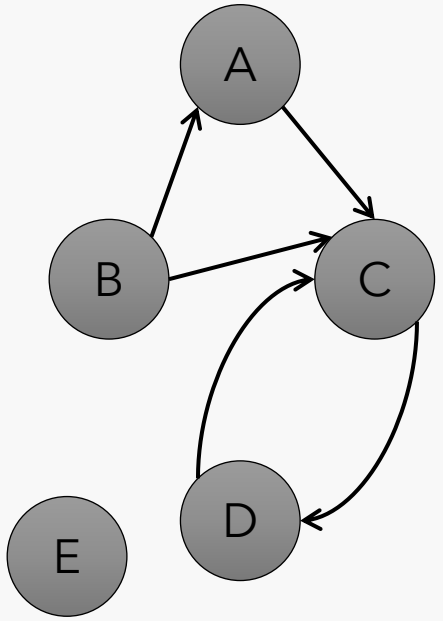
Unconnected Graph

- Consists of several connected components
- Connected components of this graph are:
  - AB
  - CD
- We'll be working with connected graphs



Directed Graphs

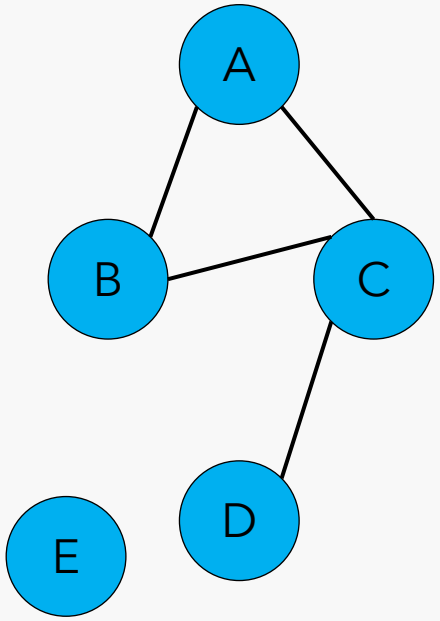
- A directed graph is a pair (V, E) where
  - V is a set
  - E is a set of ***ordered*** pairs (u, v) where $u,v \in V$
    - Often require $u \neq v$ (i.e. no self-loops)
- An element of V is called a vertex or node
- An element of E is called an edge or arc
- $|V|$ = size of V, often denoted n
- $|E|$ = size of E, often denoted m



$V = \{A, B, C, D, E\}$
$E = \{(A,C), (B,A), (B,C), (C,D), (D,C)\}$
$|V| = 5$
$|E| = 5$

Undirected Graphs

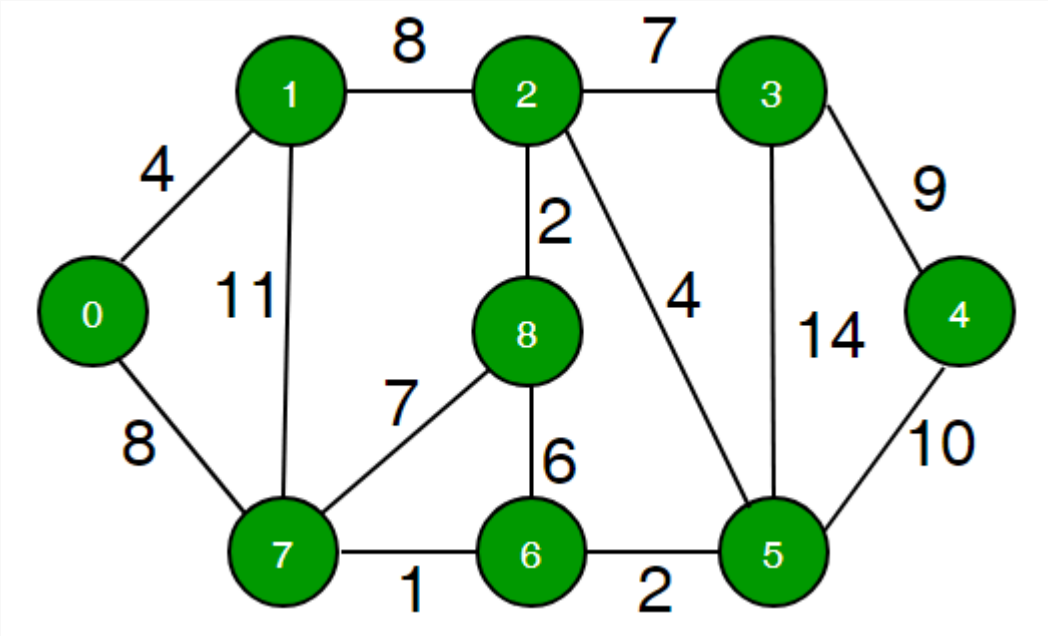
- **An undirected graph** is just like a directed graph!
  - … except that E is now a set of unordered pairs {u, v} where u, v ∈ V
- Every undirected graph can be easily converted to an equivalent directed graph via a simple transformation:
  - Replace every undirected edge with two directed edges in opposite directions
  - … but not vice versa



$V = \{A, B, C, D, E\}$
$E = \{\{A,C\}, \{B,A\}, \{B,C\}, \{C,D\}\}$
$|V| = 5$
$|E| = 4$

Weighted Graphs

- A graph where edges have weights, which quantifies the relationship
- For example
  - Assign path distances between cities
  - Or airline costs
- These graphs can be directed or undirected

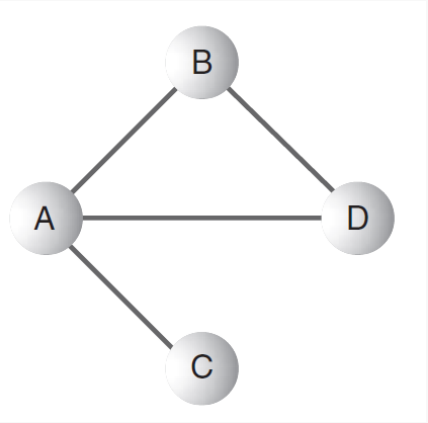


Vertices: Java Implementation

- We can represent a vertex as a Java class with:
  - Character data
  - A Boolean data member to check if it has been visited
- Now we need to specify edges.
- But in a graph, we don’t know how many there will be!
- We can do this using either an adjacency matrix or an adjacency list .
- We’ll look at both.

Graph Representation: Adjacency Matrix

- An adjacency matrix for a graph with ***n*** nodes, is size ***n*** *x* ***n***
- Position ***(i, j)***
  - has the value of ***1*** if there is an edge connecting node ***i*** with node ***j***
  - has the value of ***0*** otherwise



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

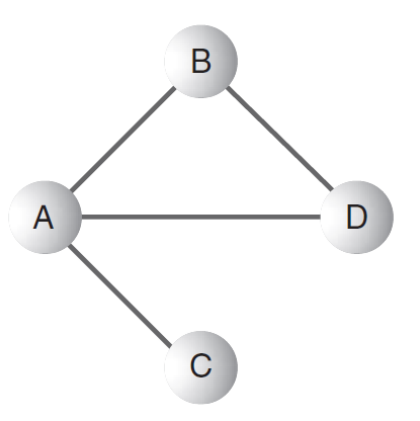
Practice

- What is the graph of this adjacency matrix?

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

Redundant?

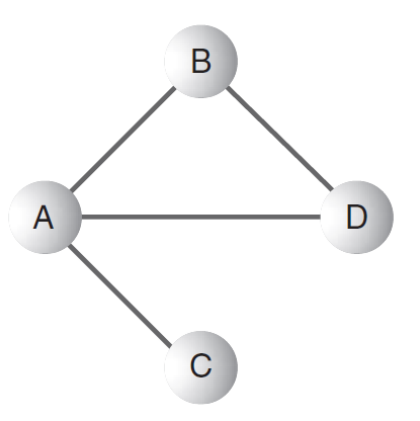
- Why store two pieces of information for the same edge?
  - i.e., (A, B) and (B, A)
- Unfortunately, there's no easy way around it
  - Because edges have no direction
  - No concept of 'parents' and 'children'



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Graph Representation: Adjacency List

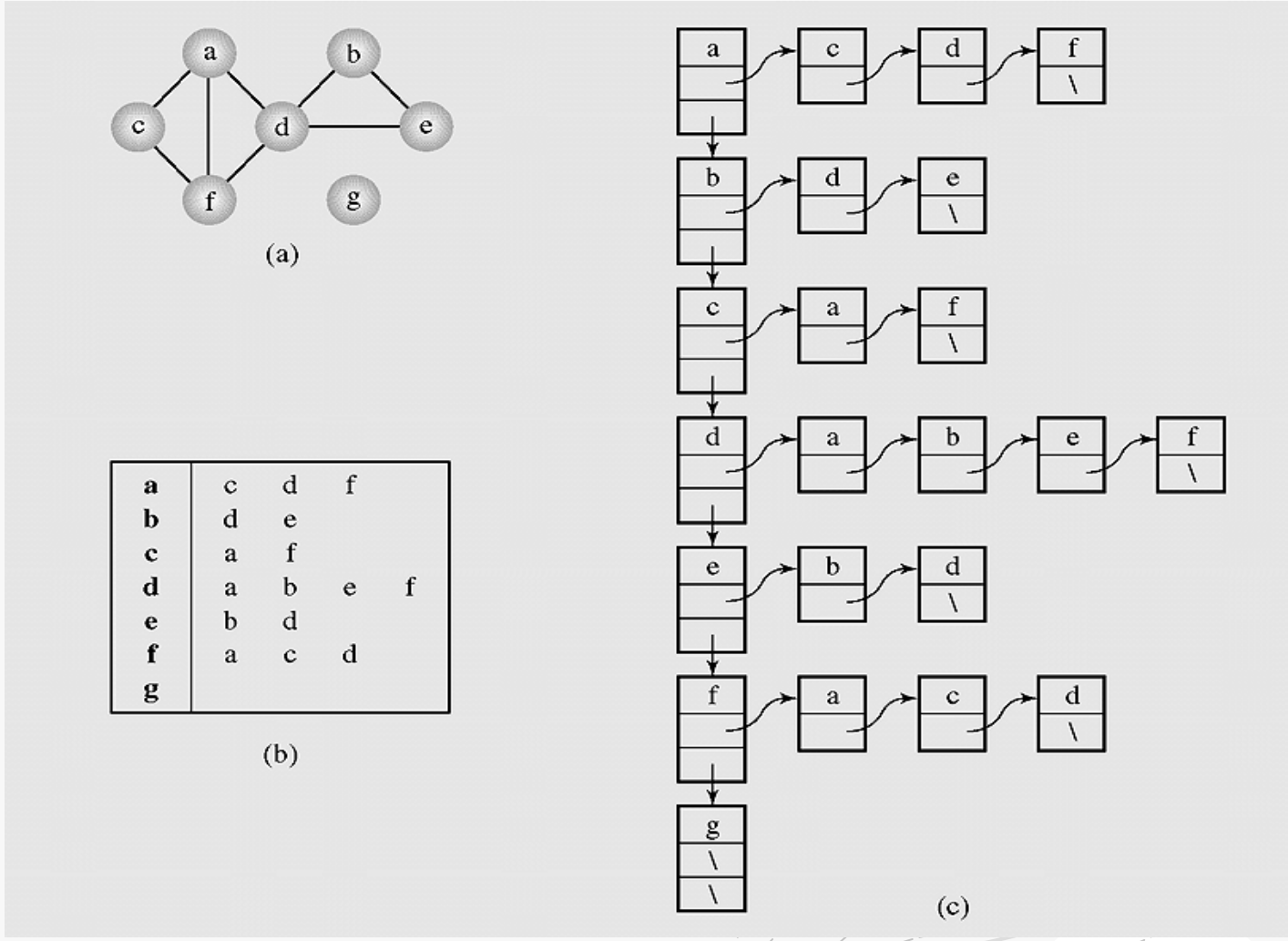
- An array of linked lists
  - Index by vertex, and obtain a linked list of neighbors
  - Here is the same graph, with its adjacency list:



Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

Graph Representation: Adjacency List

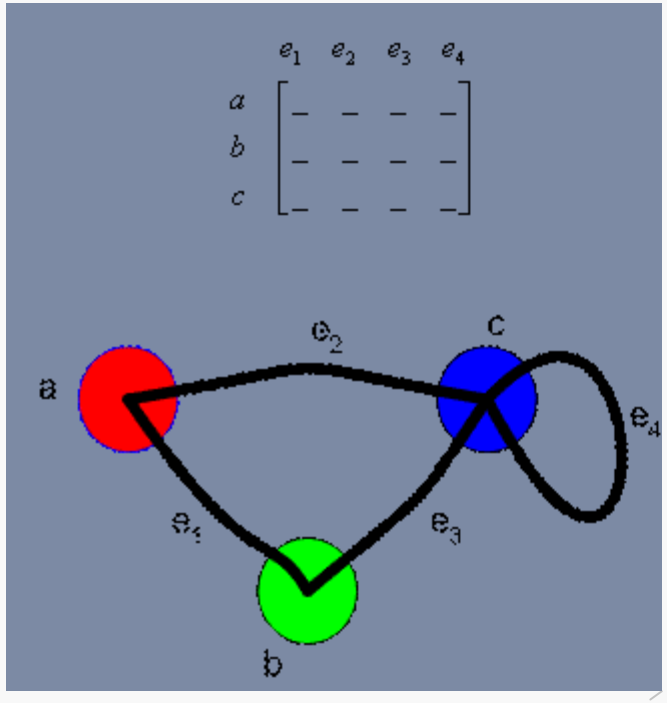
- Graph representations.
- A graph (a) can be represented as (b–c) in an adjacency list.



Graph Representation: Incident Matrix

- A vertex is said to be **incident to an edge** if the edge is connected to the vertex.

Graph represented by incident matrix

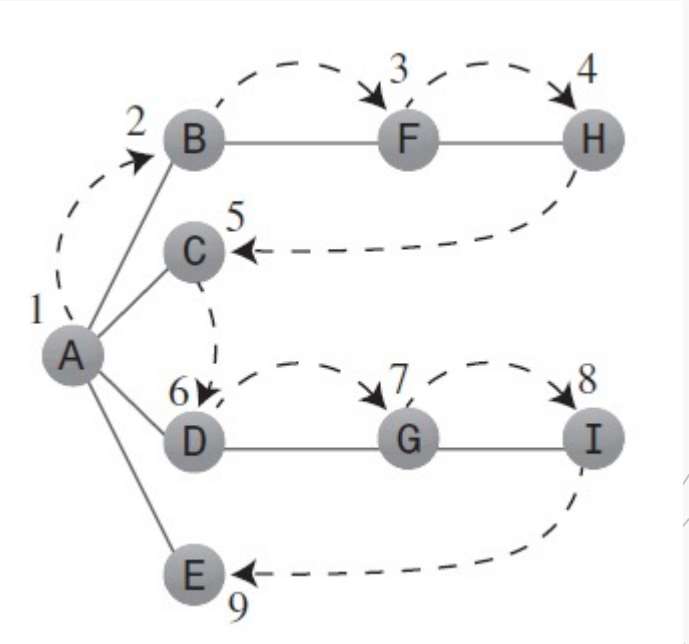


Application: Searches

- A fundamental operation for a graph is:
  - Starting from a particular vertex
  - Find all other vertices which can be reached by following paths
- Example application
  - How many towns in the VN can be reached by train from Da Lat?
- Two approaches
  - Depth first search (DFS)
  - Breadth first search (BFS)

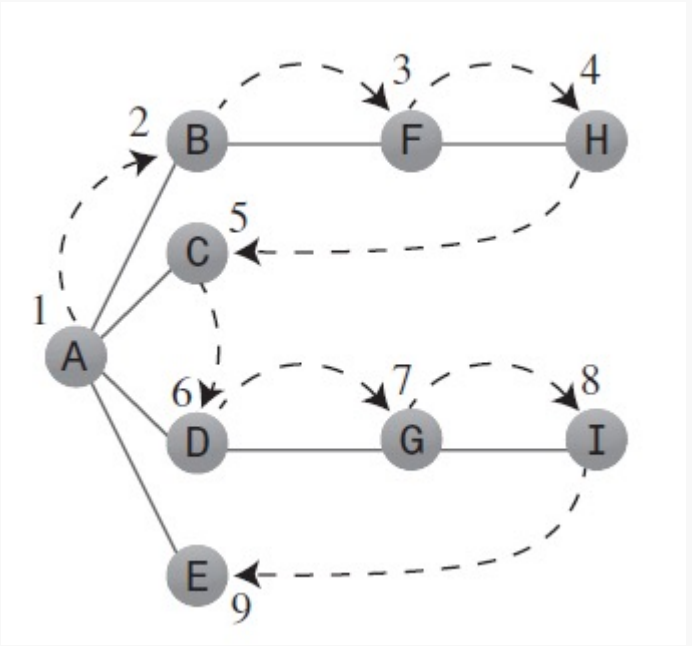
Depth First Search (DFS)

- Idea
  - Pick a starting point
  - Follow a path to unvisited vertices, as long as you can until you hit a dead end
  - When you hit a dead end, go back to a previous spot and hit unvisited vertices
  - Stop when every path is a dead end



Depth First Search (DFS)

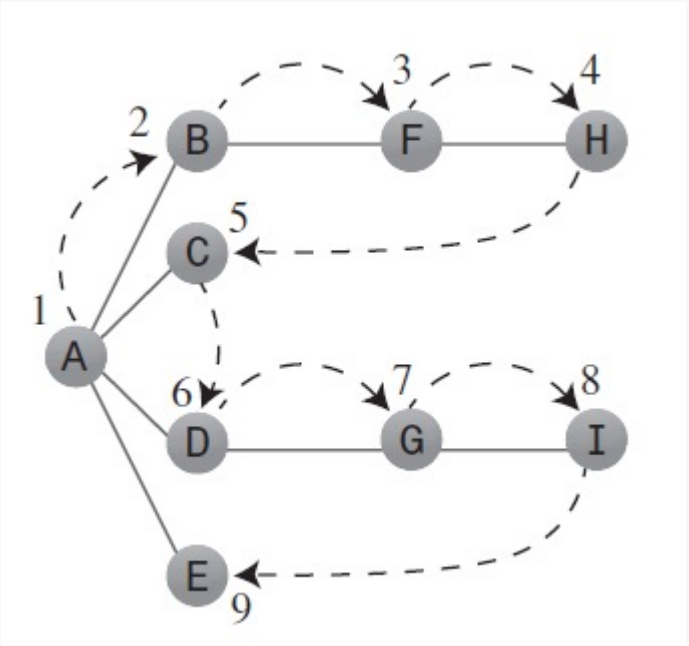
- Algorithm
  - Pick a vertex (call it A) as your starting point
  - Visit this vertex, and:
    - Push it onto a stack of visited vertices
    - Mark it as visited (so we don't visit it again)
  - Visit any neighbor of A that hasn't yet been visited
    - Repeat the process
  - When there are no more unvisited neighbors
    - Pop the vertex off the stack
  - Finished when the stack is empty



- Note: We get as far away from the starting point until we reach a dead end, then pop (can be applied to mazes)

Activity

- Stack contents during depth first search
- Two columns:
  - Activity step
  - Stack content

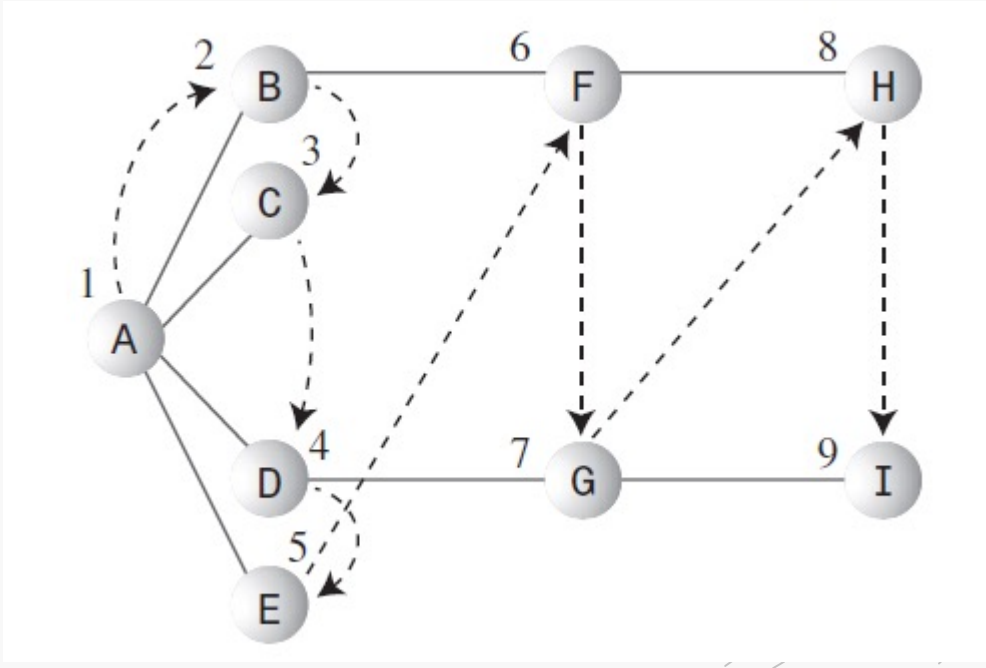


Depth First Search: Complexity

- Let |V| be the number of vertices in a graph
- And let |E| be the number of edges
- In the worst case, we visit every vertex and every edge:
  - O(|V| + |E|) time
- At first glance, this doesn't look too bad
  - But remember a graph can have lots of edges!
  - Worst case, every vertex is connected to every other:
    - $(n-1) + (n-2) + \ldots + 1 = O(n^2)$
  - So it can be expensive if the graph has many edges

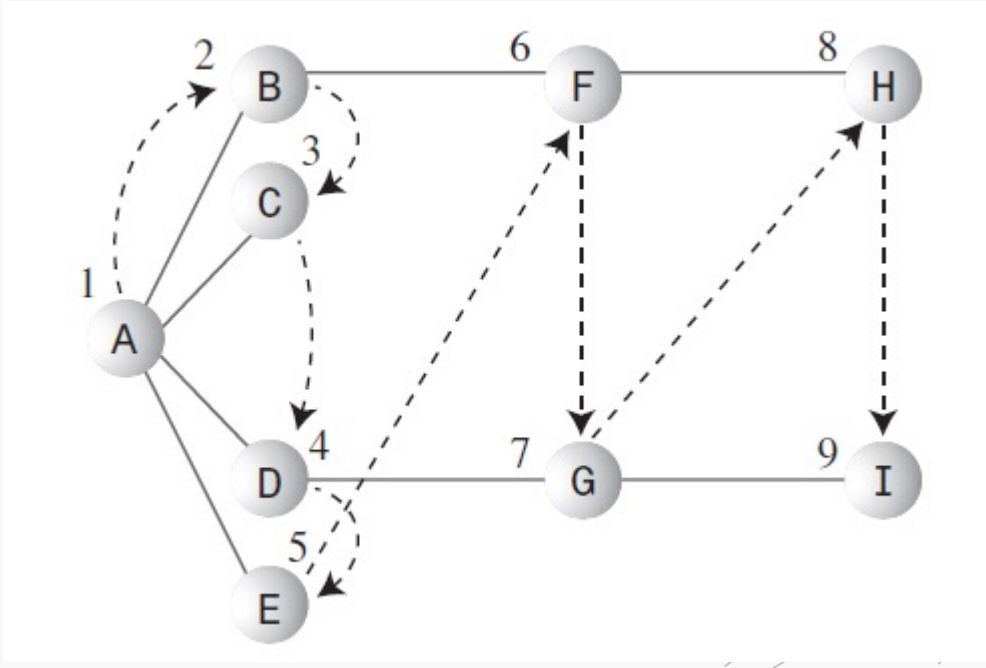
Breadth First Search (BFS)

- Same application as DFS; we want to find all vertices which we can get to from a starting point, call it A
- However this time, instead of going as far as possible until we find a dead end, like DFS
  - We visit all the closest vertices first
  - Then once all the closest vertices are visited, branch further out



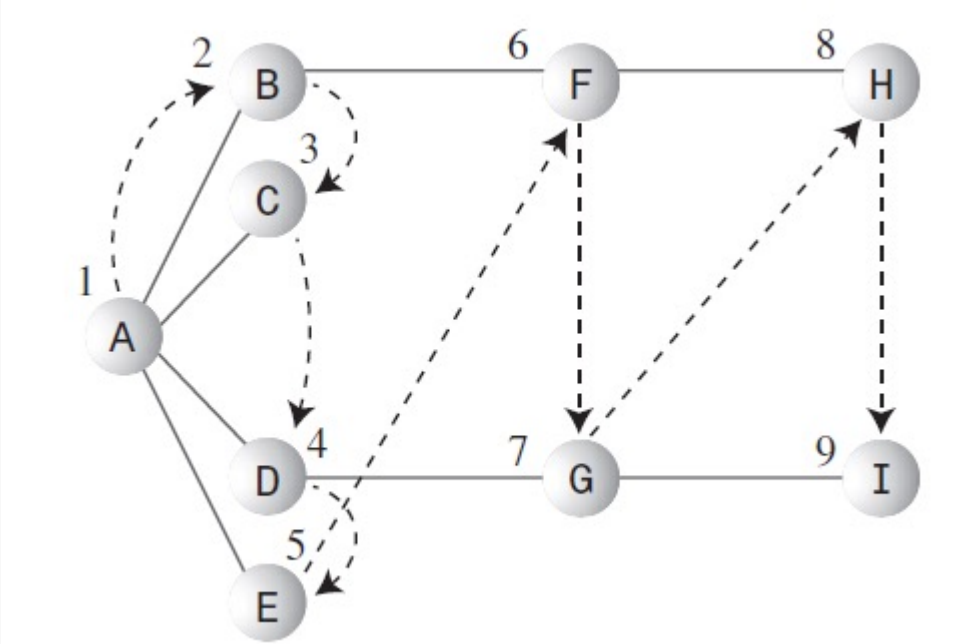
Breadth First Search (BFS)

- We’re going to use a queue instead of a stack!
- Algorithm
  - Start at a vertex, call it current
  - If there is an unvisited neighbor, mark it, and insert it into the queue
  - If there is not:
    - If the queue is empty, we are done
    - Otherwise: Remove a vertex from the queue and set current to that vertex, and repeat the process



Activity

- Queue contents during depth first search
- Two columns:
  - Activity step
  - Queue content

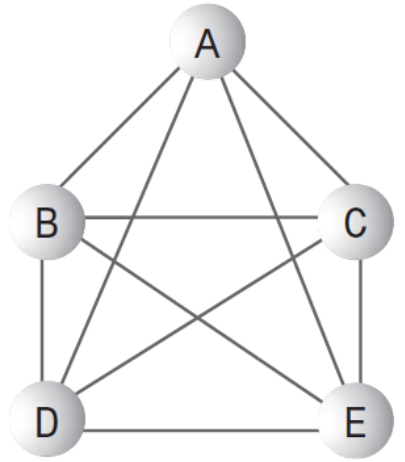


Breadth First Search: Complexity

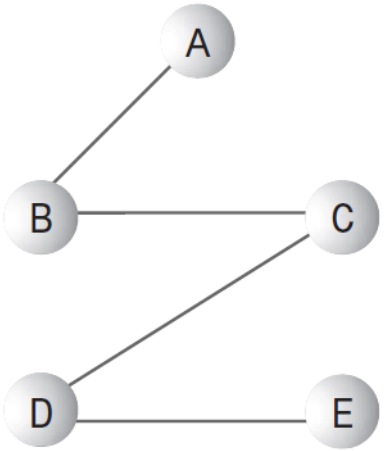
- Let |V| be the number of vertices in a graph
- And let |E| be the number of edges
- In the worst case, we visit every vertex and every edge:
  - O(|V| + |E|) time
  - Same as DFS
- Again, if the graph has lots of edges, you approach quadratic run time which is the worst case

Minimum Spanning Trees (MSTs)

- On that note of large numbers of edges slowing down our precious search algorithms:
- Let's look at MSTs, which can help ease this problem
- It would be nice to take a graph and reduce the number of edges to the minimum number required to span all vertices:



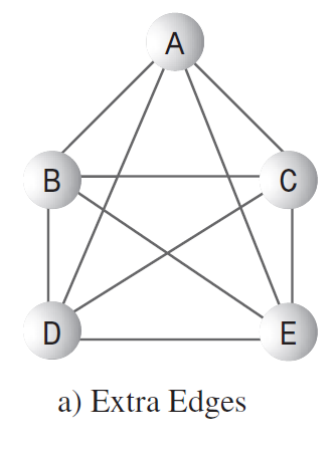
a) Extra Edges



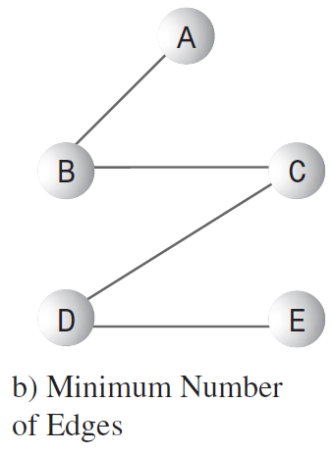
b) Minimum Number of Edges

We've done it already...

- Actually, if you execute DFS you've already computed the MST!



a) Extra Edges



b) Minimum Number of Edges

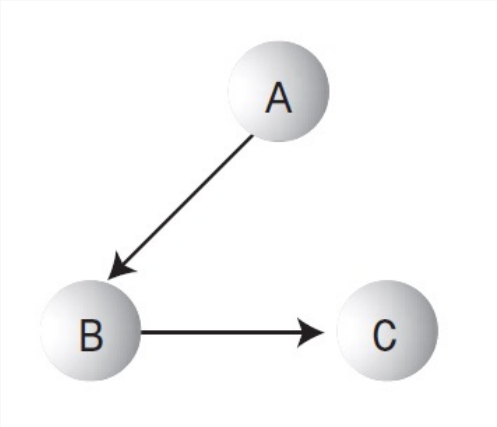
- Think about it: you follow a path for as long as you can, then backtrack (visit every vertex at most once)
- ➔ Just have to save edges as you go

Homework

- Code and compile:
  - (page 631) LISTING 13.1 The **dfs.java** Program
  - (page 639) LISTING 13.2 The **bfs.java** Program

Directed Graphs

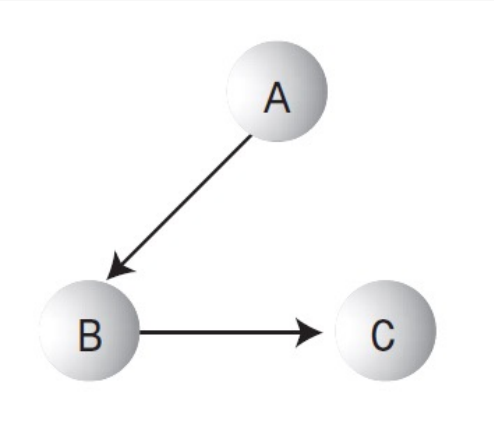
- A directed graph is a graph where the edges have direction, signified by arrows

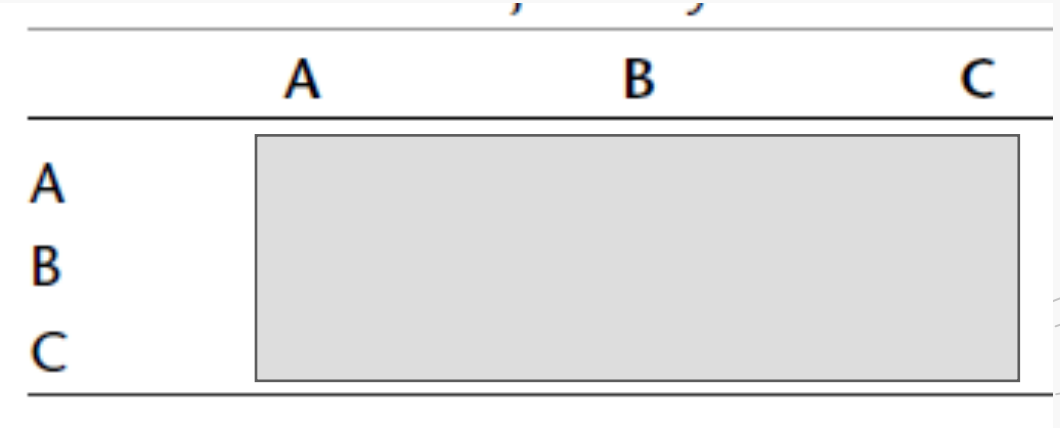


- This will simplify the adjacency matrix a bit…

Adjacency Matrix

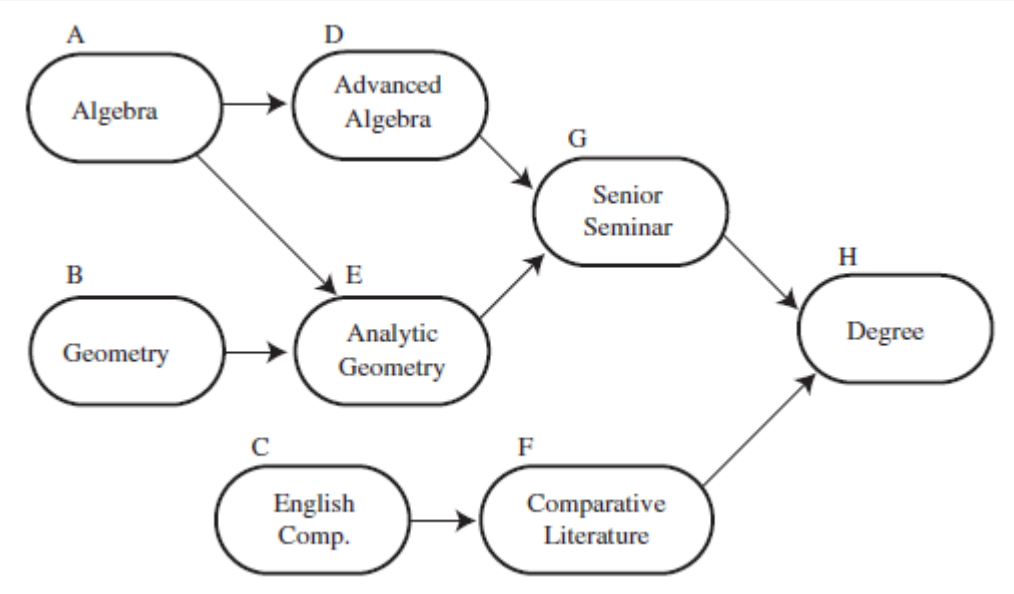
- The adjacency matrix for this graph does not contain redundant entries
  - Because now each edge has a source and a sink
  - So entry ***(i, j)*** is only set to ***1*** if there is an edge going from ***i*** to ***j***
    - ***0*** otherwise





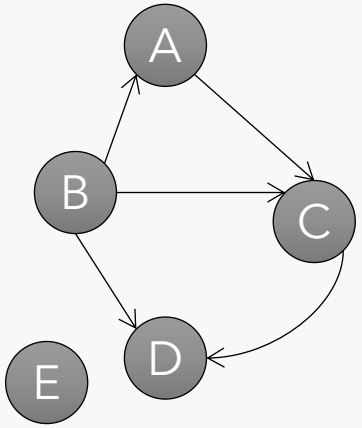
Topological Sorting

- Another operation that can be modeled with graphs
- Example:
  - Arrange the courses in the order you need to take
  - To get your Degree

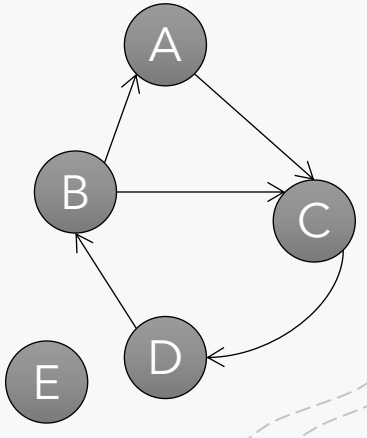


Topological Sorting

- Only works with ***DAGs (Directed Acyclic Graphs)***
  - That is if the graph has a cycle, this will not work
- Idea: Sort all vertices such that if a vertex's successor always appears after it in a list (application: course prerequisites)

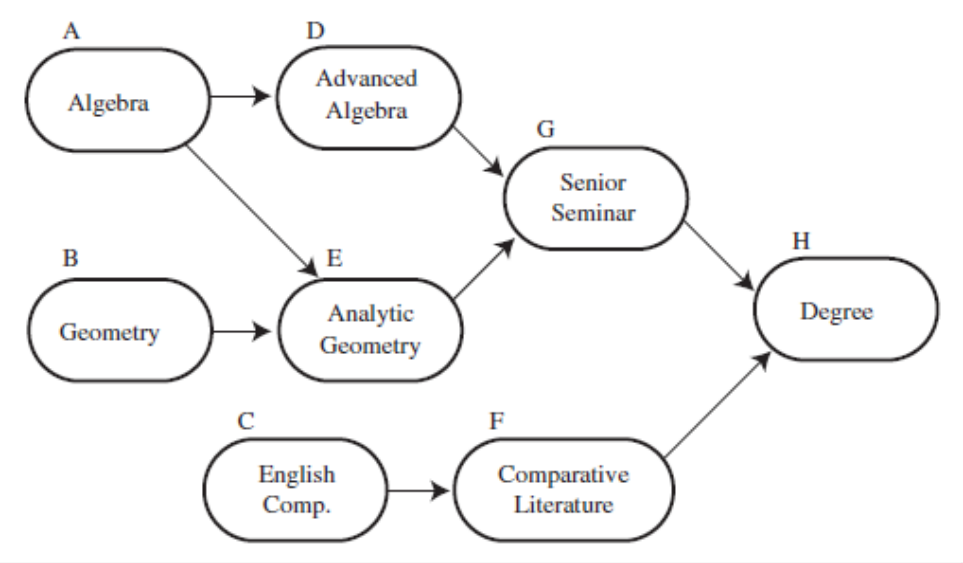


DAG



Not a DAG

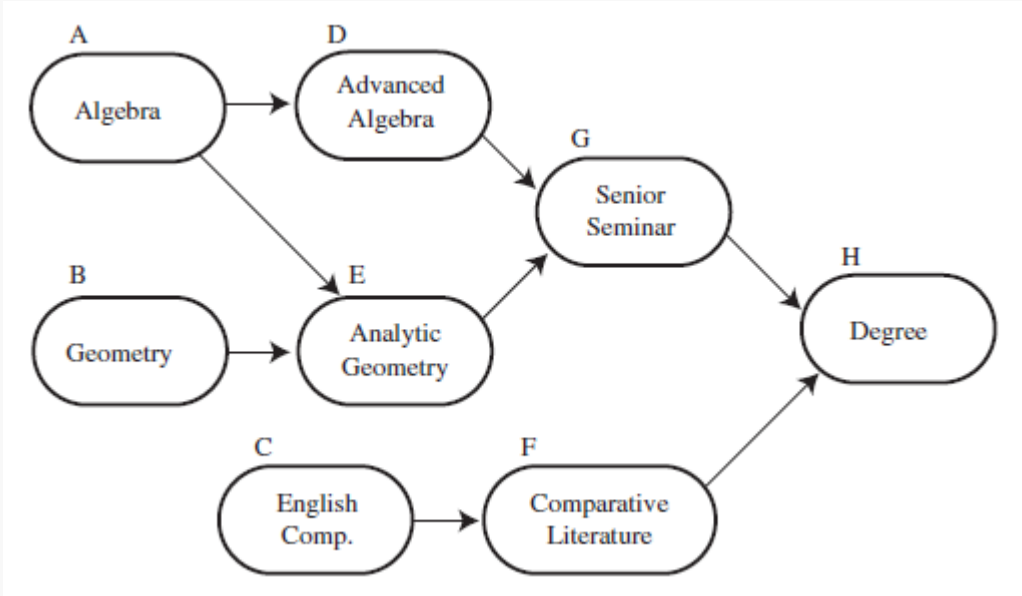
Topological Sorting



- Algorithm
  - Find a vertex that has no successors
  - Delete this vertex from the graph, and insert its label into a list
  - Repeat until every vertex is gone
- Works because if a vertex has no successors, it must be the last one (or tied for last) in the topological ordering
  - As soon as it's removed, at least one of the remaining vertices must not have successors
    - Because there are no cycles!
  - Graph does NOT have to be fully connected for this to work!

Topological Sorting

- The work is done in the **while** loop, which continues until the number of vertices is reduced to 0.
- Here are the steps involved:
  1. Call **noSuccessors()** to find any vertex with no successors.
  2. If such a vertex is found, put the vertex label at the end of **sortedArray[]** and delete the vertex from graph.
  3. If an appropriate vertex isn't found, the graph must have a cycle.



Java code - topo

```
public void topo() // topological sort
{
     int orig_nVerts = nVerts; // remember how many verts

     while(nVerts > 0) // while vertices remain,
     {
          // get a vertex with no successors, or -1
          int currentVertex = noSuccessors();

          if(currentVertex == -1) // must be a cycle
          {
               System.out.println("ERROR: Graph has cycles");
               return;
          }
          // insert vertex label in sorted array (start at end)
          sortedArray[nVerts-1] = vertexList[currentVertex].label;

          deleteVertex(currentVertex); // delete vertex
     } // end while

     // vertices all gone; display sortedArray
     System.out.print("Topologically sorted order: ");

     for(int j=0; j<orig_nVerts; j++)
          System.out.print( sortedArray[j] );
     System.out.println("");
} // end topo
```

Java code – noSuccessor()

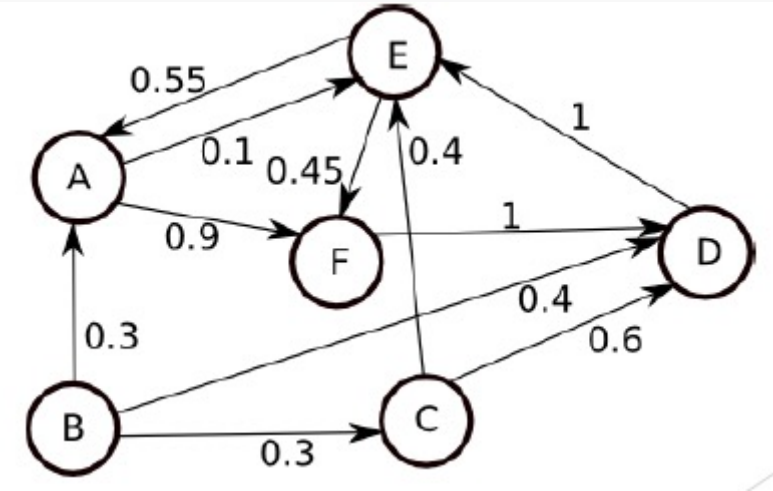
```
public int noSuccessors() // returns vert with no successors
{ // (or -1 if no such verts)
    boolean isEdge; // edge from row to column in adjMat

    for(int row=0; row<nVerts; row++) // for each vertex,
    {
        isEdge = false; // check edges
        for(int col=0; col<nVerts; col++)
        {
            if( adjMat[row][col] > 0 ) // if edge to
            { // another,
                isEdge = true;
                break; // this vertex
            } // has a successor
        } // try another

        if( !isEdge ) // if no edges,
                return row; // has no successors
    }
    return -1; // no such vertex
} // end noSuccessors()
```

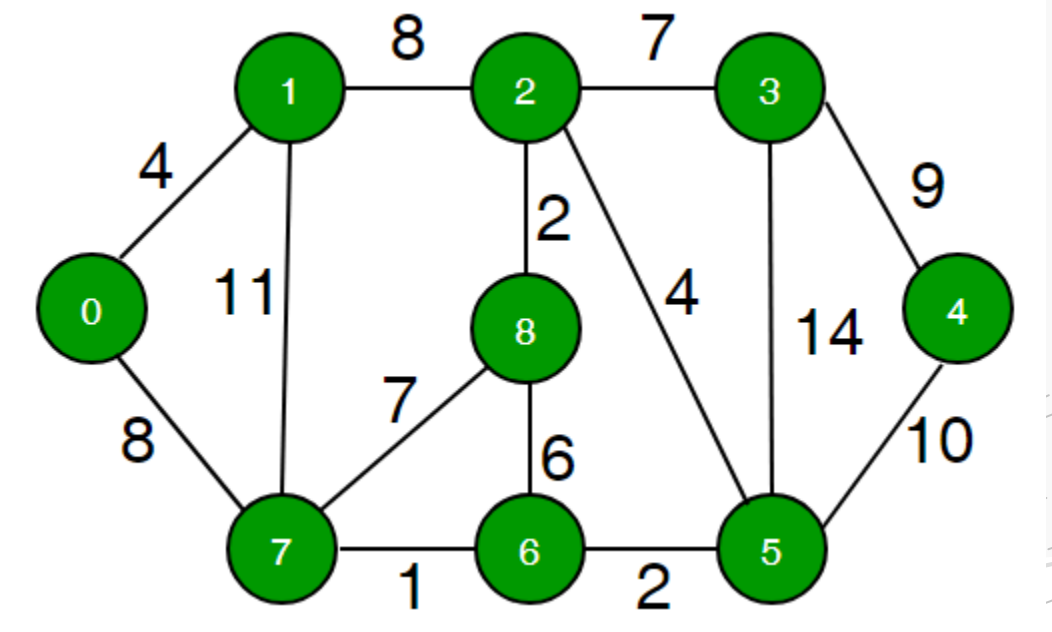
Java code

- LISTING 13.4 The **topo.java** Program

Weighted Graphs

- Once again, a graph where edges have weights, which quantifies the relationship
- These graphs can be directed or undirected

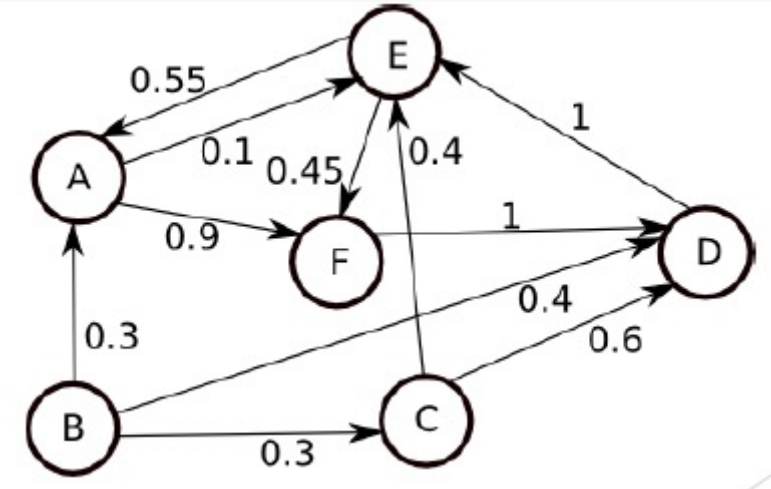




Weighted Graph: Adjacency List

- The adjacency list for a weighted graph contains ***edge weights***
  - Instead of ***0*** and ***1***
- If there is no edge connecting vertices ***i*** and ***j***, a weight of ***INFINITY*** is used (**not** 0!)
  - Because '0' can also be a weight
  - Also, most applications of weighted graphs are to find minimum spanning trees or shortest path (we'll look at this)
- Also remember if the graph is undirected, redundant information should be stored

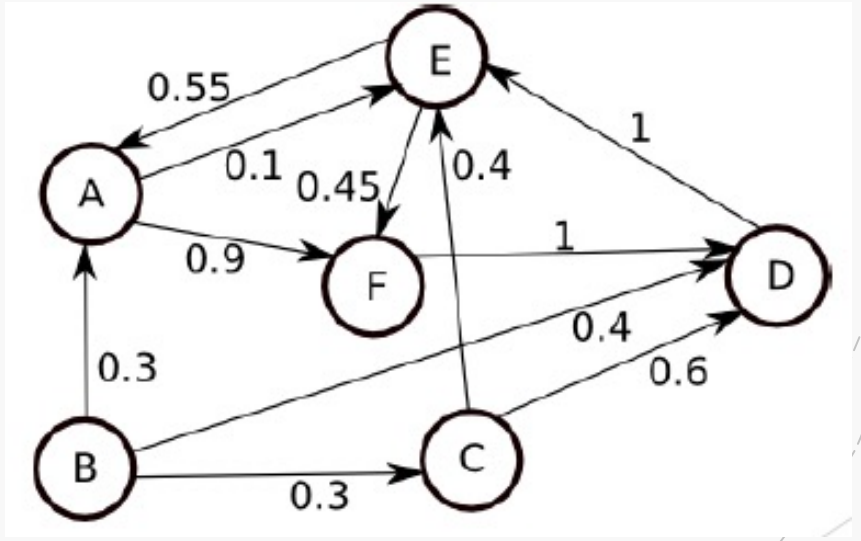
Weighted Graph: Adjacency List



	A	B	C	D	E	F
A	Inf	Inf	Inf	Inf	0.1	0.9
B	0.3	Inf	0.3	0.4	Inf	Inf
C	Inf	Inf	Inf	0.6	0.4	Inf
D	Inf	Inf	Inf	Inf	1	Inf
E	0.55	Inf	Inf	Inf	Inf	0.45
F	Inf	Inf	Inf	1	Inf	Inf

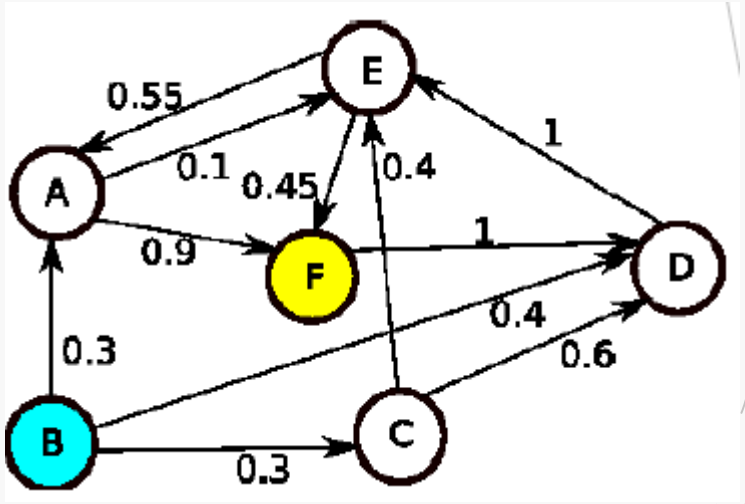
Dijkstra's Algorithm

- Given a weighted graph, find the shortest path (in terms of edge weights) between two vertices in the graph
- Numerous applications
  - Cheapest airline fare between departure and arrival cities
  - Shortest driving distance in terms of mileage



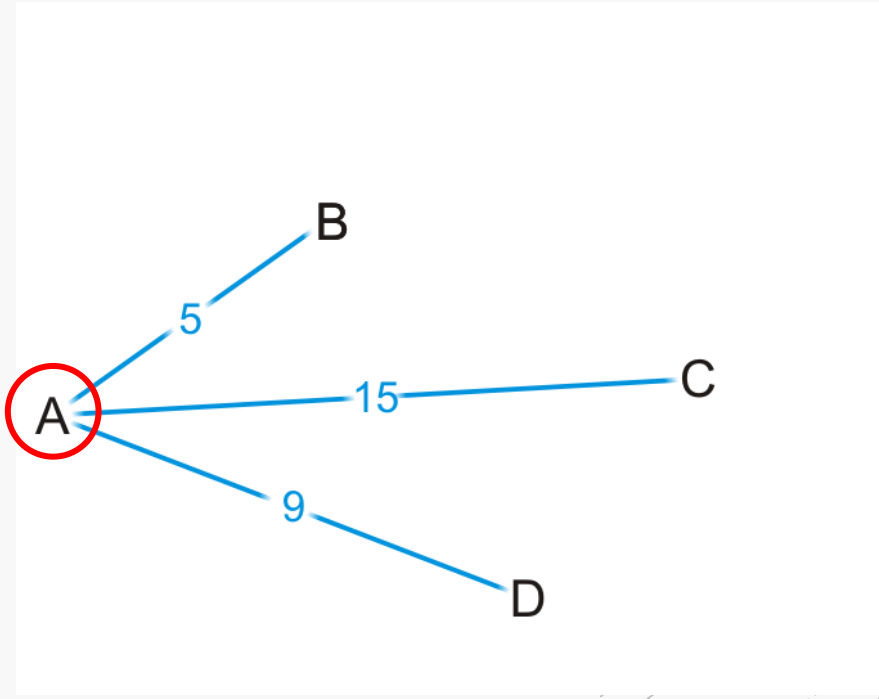
Dijkstra's Algorithm

- Suppose in the graph below, we wanted the shortest path from B to F
- **Idea**: Maintain a table of the current shortest paths from B to all other vertices (and the route it takes)
  - When finished, the table will hold the shortest path from B to all other vertices



Dijkstra's Algorithm

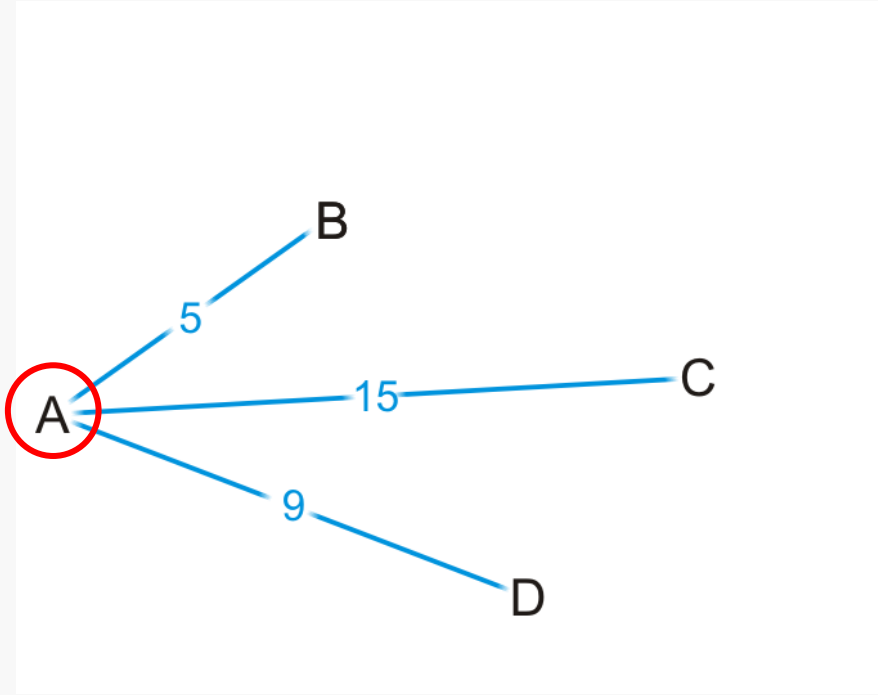
- Dijkstra's algorithm solves the single-source shortest path problem
- Assumption: all the weights are positive
- Suppose you are at vertex A
  - You are aware of all vertices adjacent to it
  - This information is either in an adjacency list or adjacency matrix



Strategy

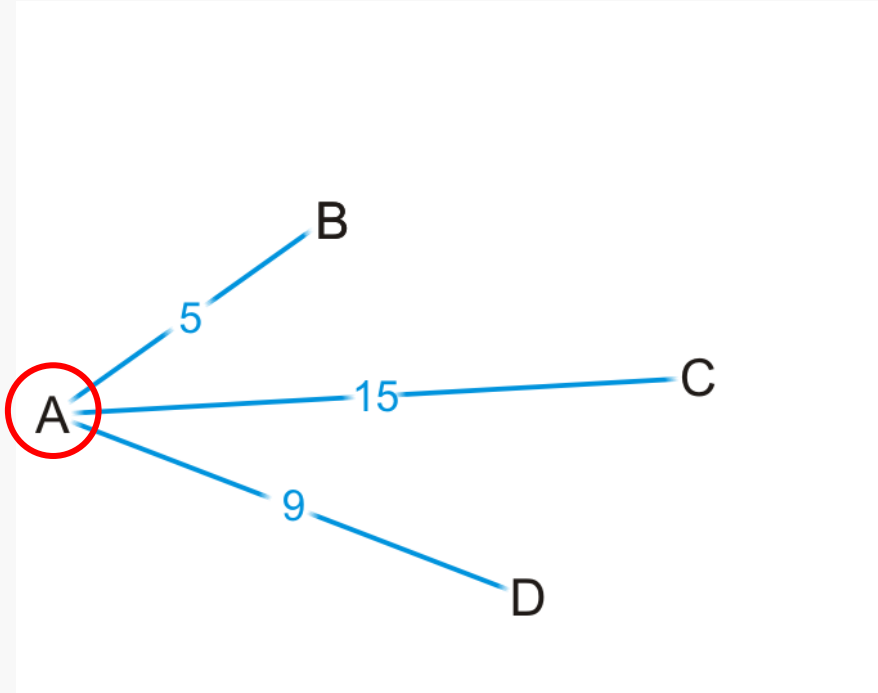
Is 5 the shortest distance to B via the edge (A, B)?

- Why or why not?



Strategy

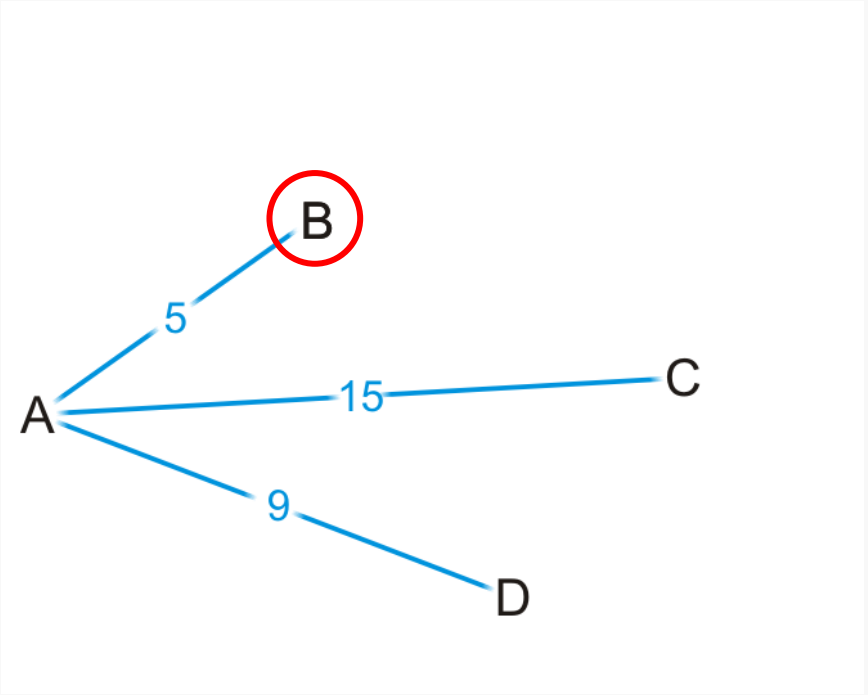
Are you guaranteed that the shortest path to C is (A, C), or that (A, D) is the shortest path to vertex D?



Strategy

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

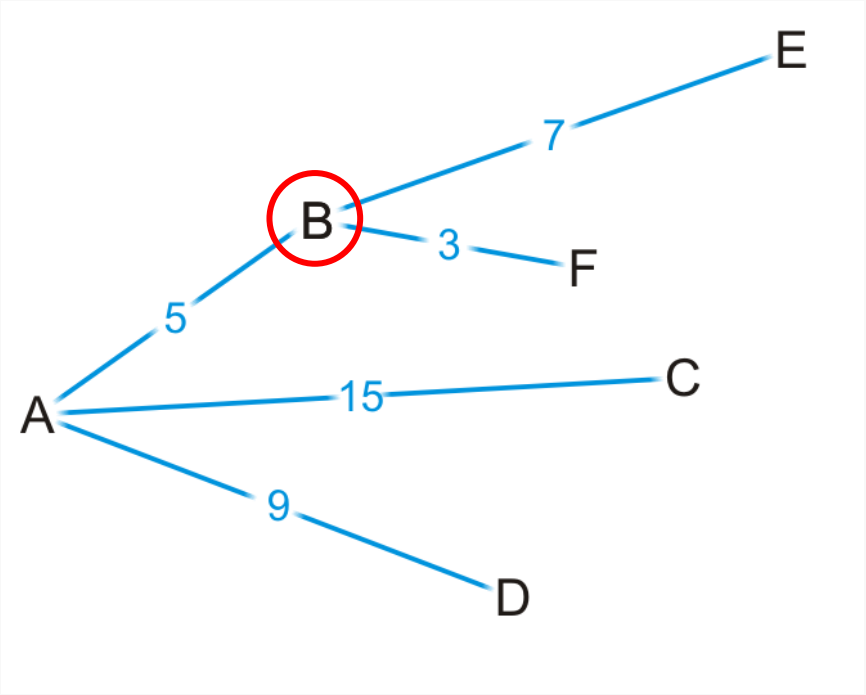
- Let's see where we can go from B



Strategy

By some simple arithmetic, we can determine that

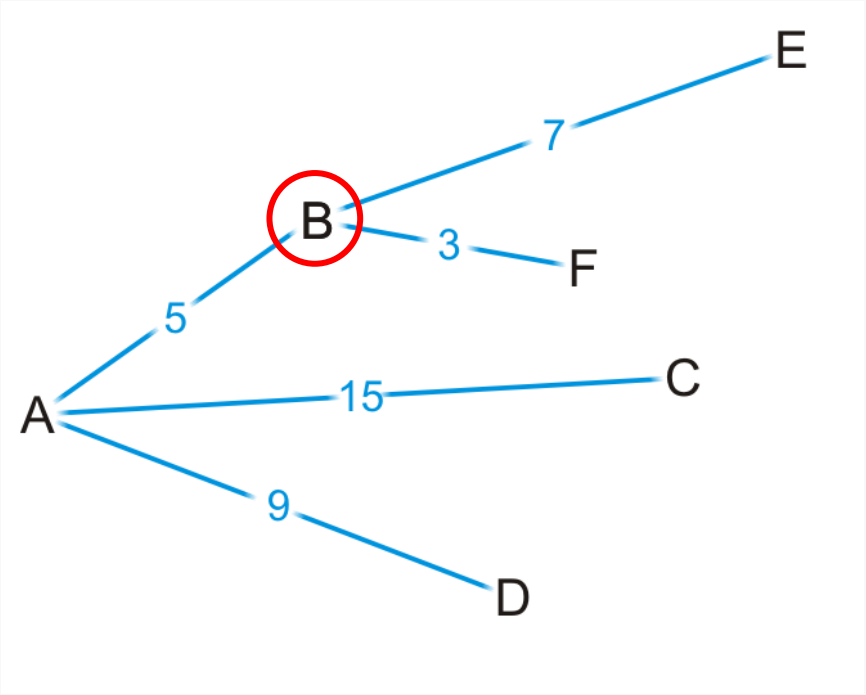
- There is a path (A, B, E) of length 5 + 7 = 12
- There is a path (A, B, F) of length 5 + 3 = 8



Strategy

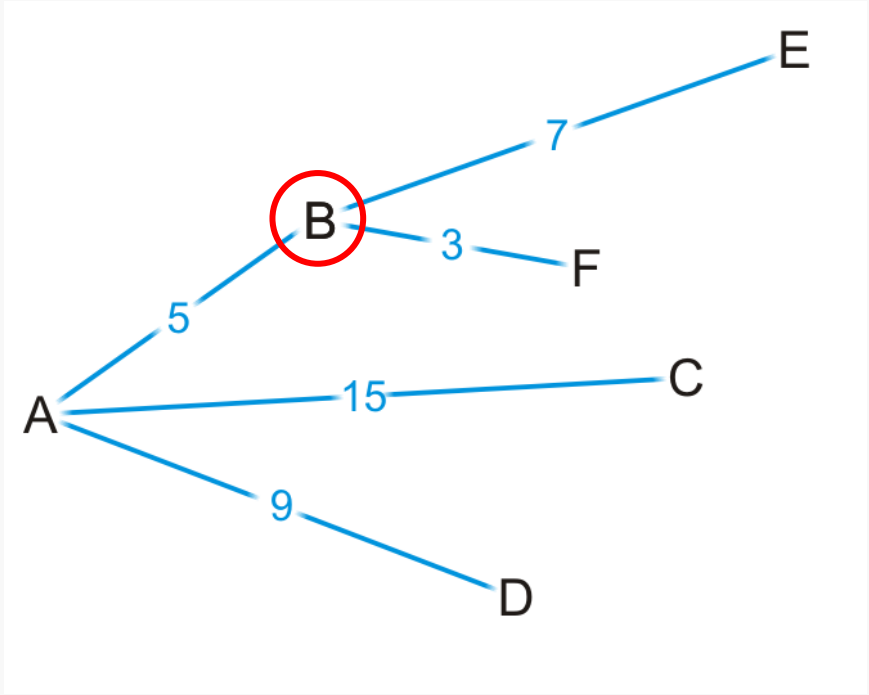
Is (A, B, F) is the shortest path from vertex A to F?

- Why or why not?



Strategy

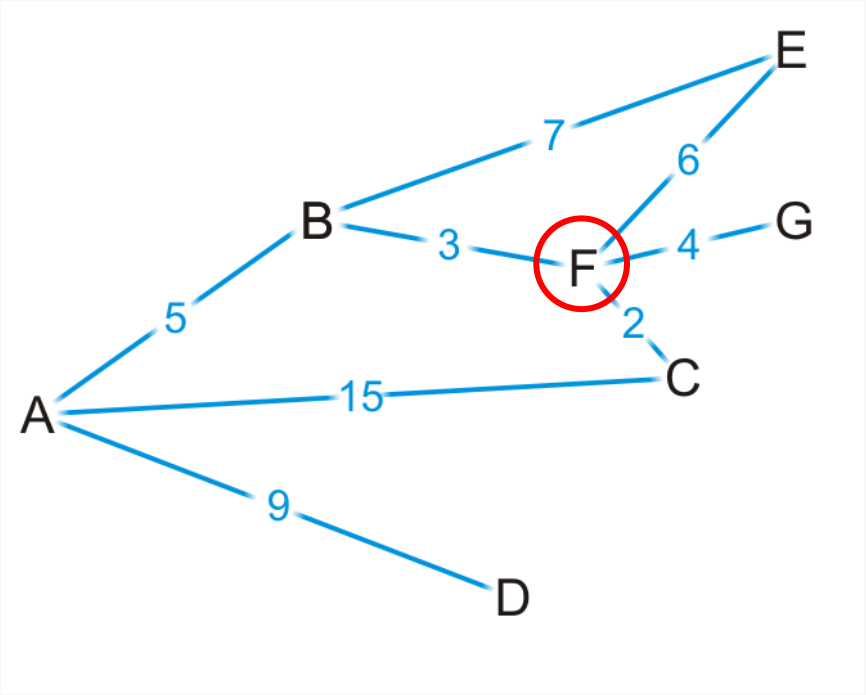
Are we guaranteed that any other path we are currently aware of is also going to be the shortest path?



Strategy

Okay, let's visit vertex F

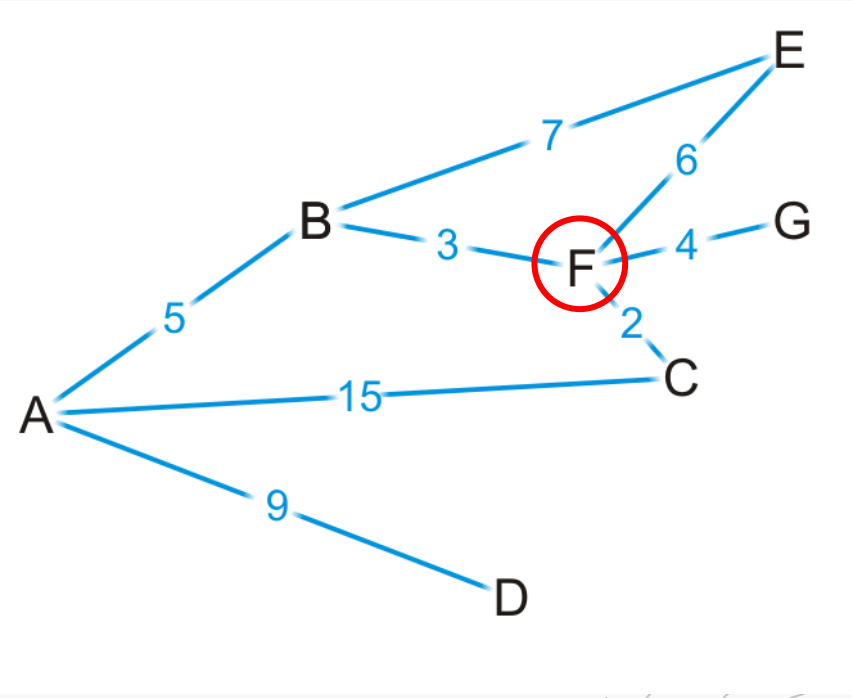
- We know the shortest path is (A, B, F) and it's of length 8



Strategy

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

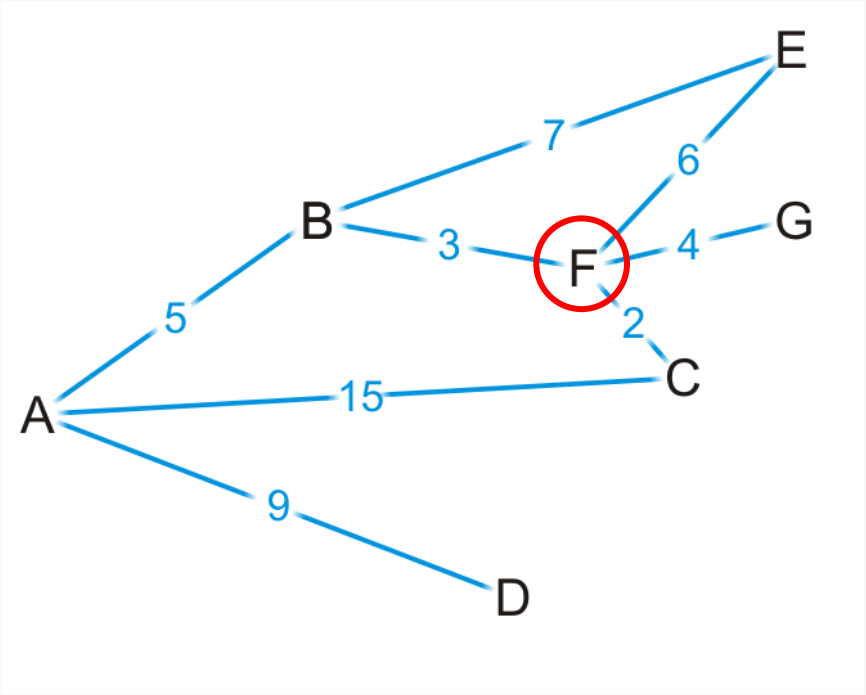
- (A, B, F, E) of length 8 + 6 = 14
- (A, B, F, G) of length 8 + 4 = 12
- (A, B, F, C) of length 8 + 2 = 10



Strategy

By observation:

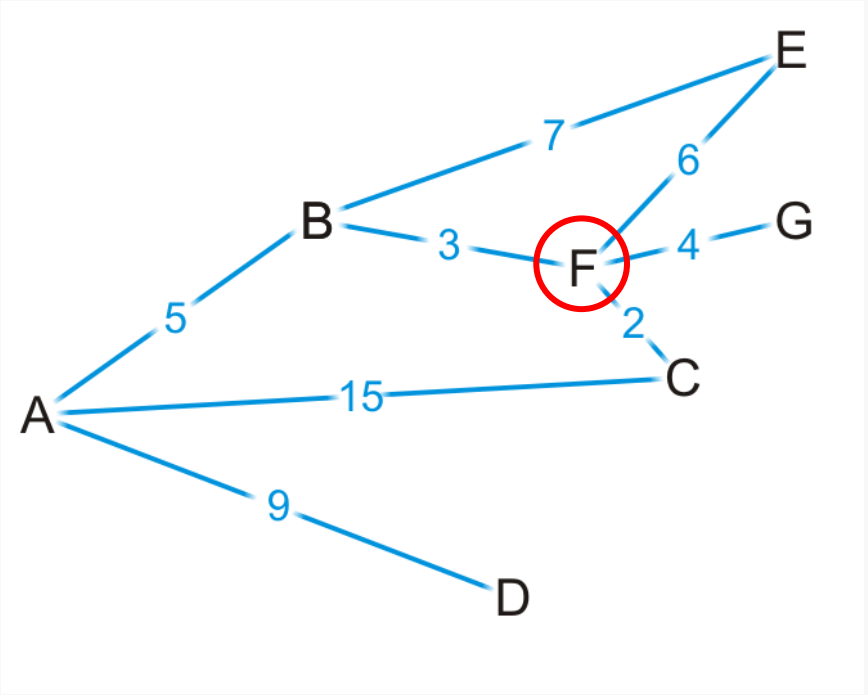
- The path (A, B, F, E) is longer than (A, B, E)
- The path (A, B, F, C) is shorter than the path (A, C)



Strategy

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

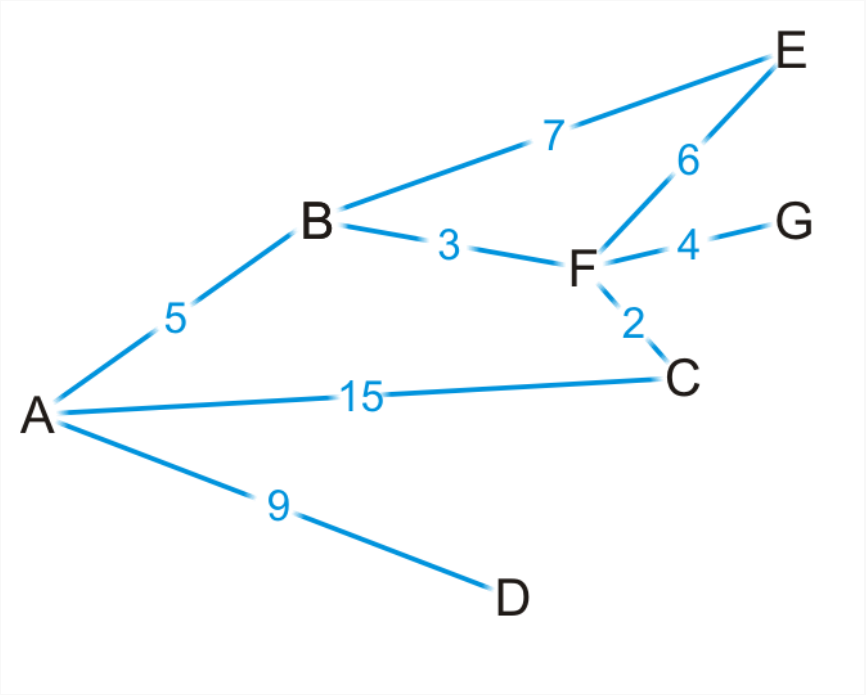
- Vertex B: (A, B) of length 5
- Vertex F: (A, B, F) of length 8



Strategy

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

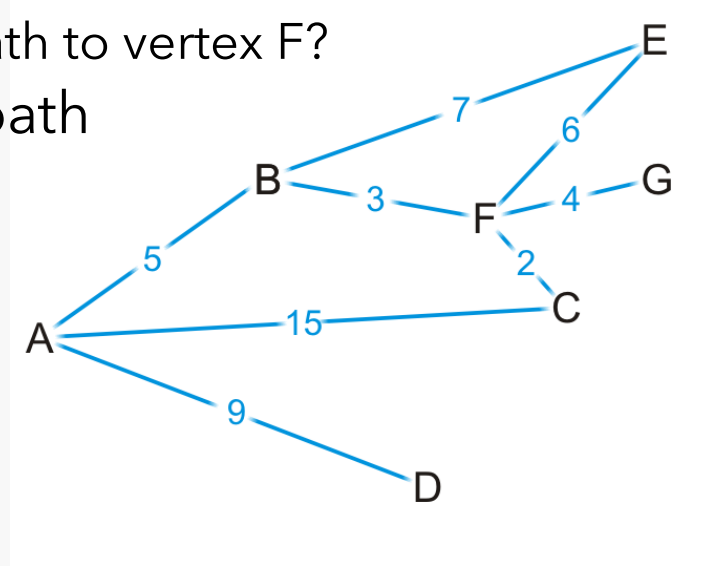
- Which remaining vertex are we currently guaranteed to have the shortest distance to?



Dijkstra's algorithm

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

- 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?
- 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



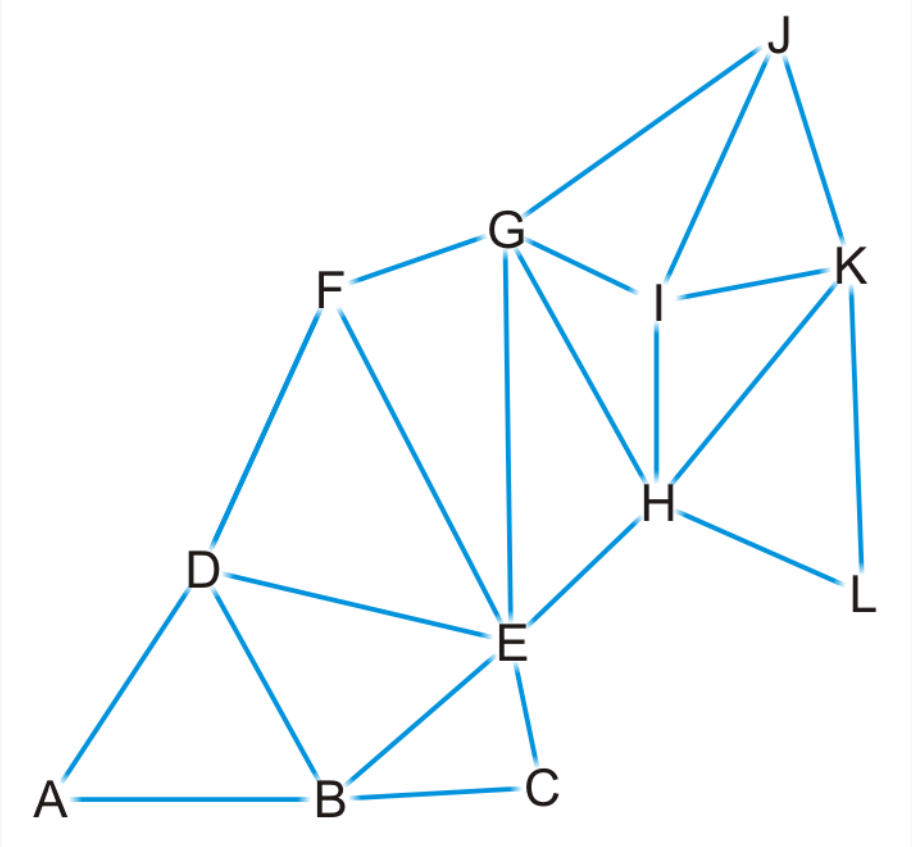
Dijkstra's algorithm

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

- Find that unvisited vertex 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

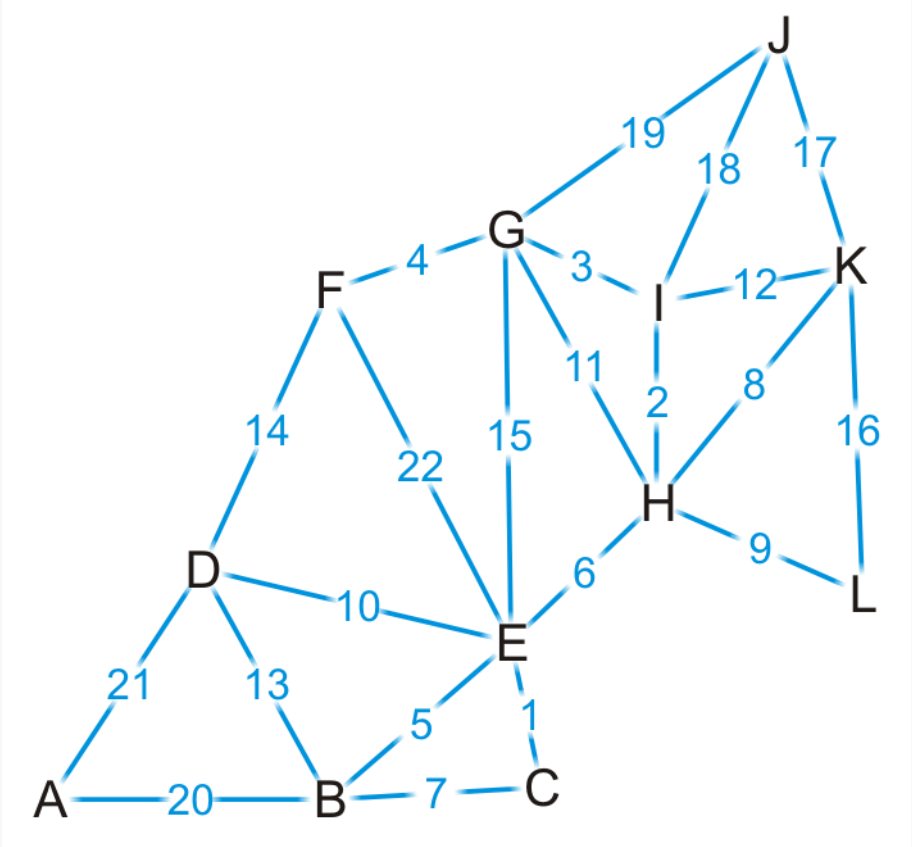
Example

Here is our abstract representation



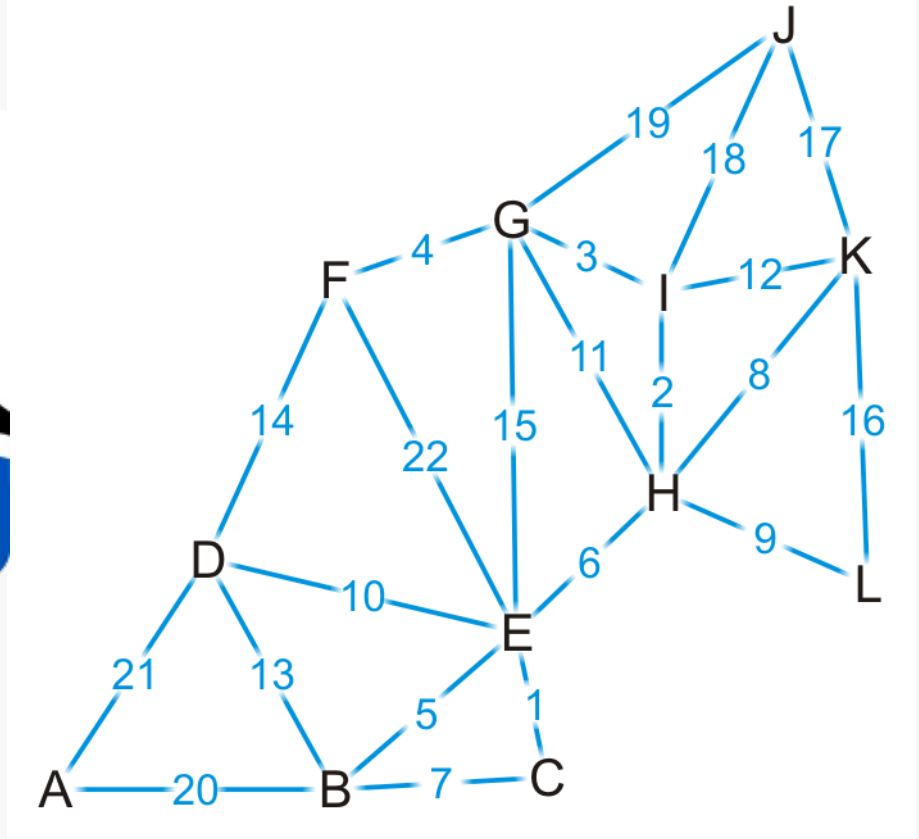
Example

Let us give a weight to each of the edges



Example

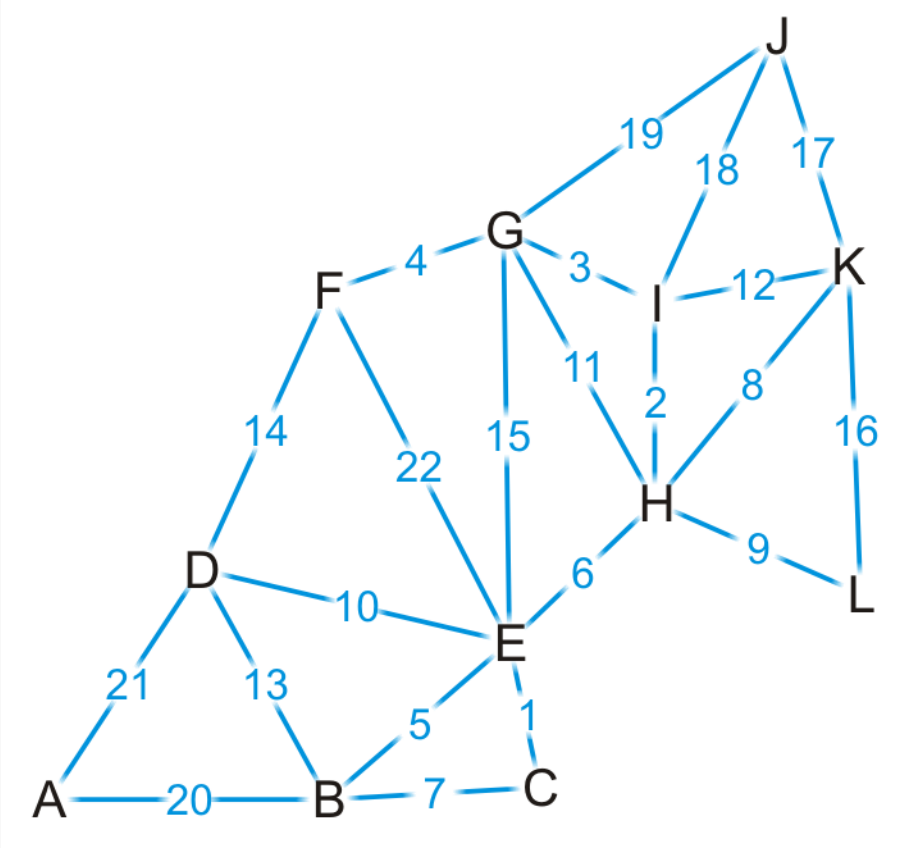
Find the shortest distance from Kamchatka (K) to every other region



Example

We set up our table

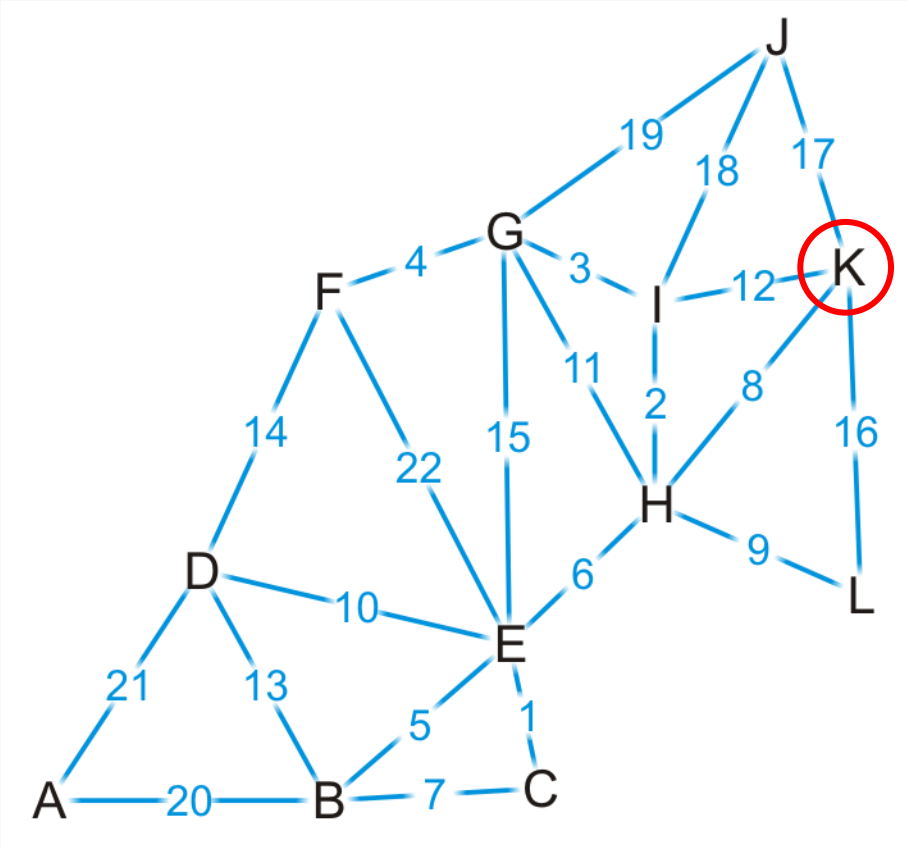
- Which unvisited vertex has the minimum distance to it?



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

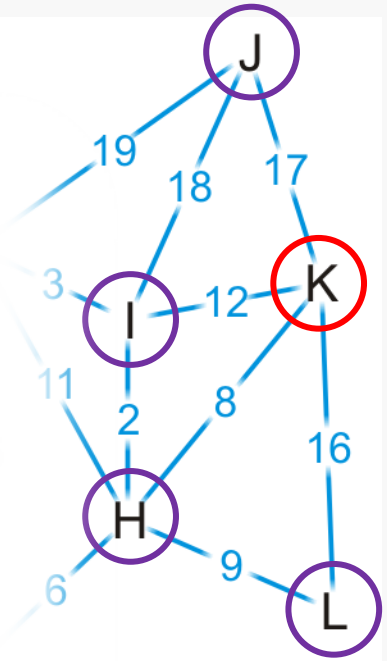
We visit vertex K



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

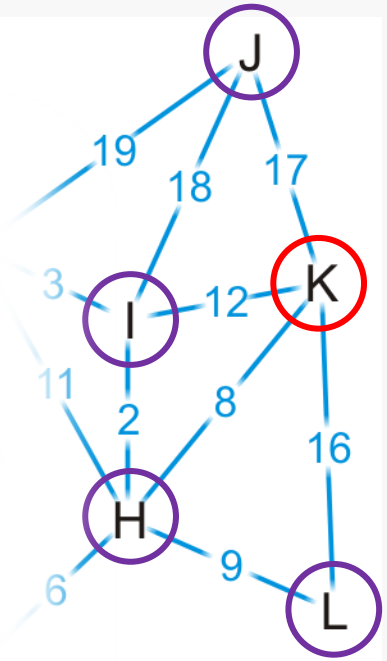
Vertex K has four neighbors: H, I, J and L



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

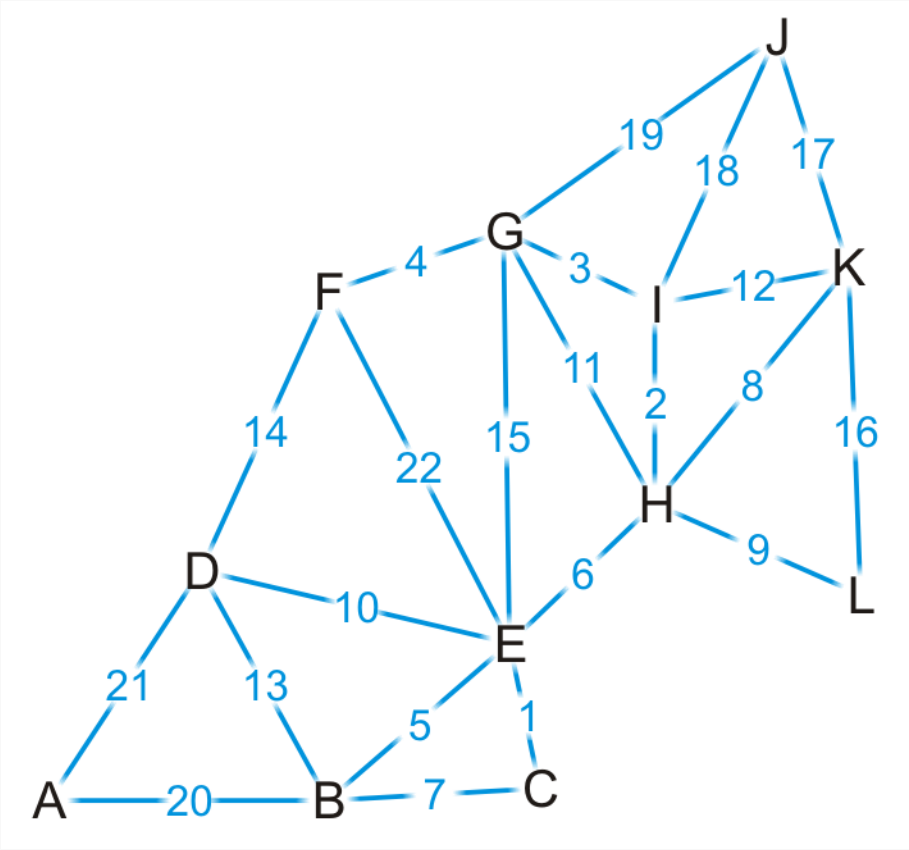


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We're finished with vertex K

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

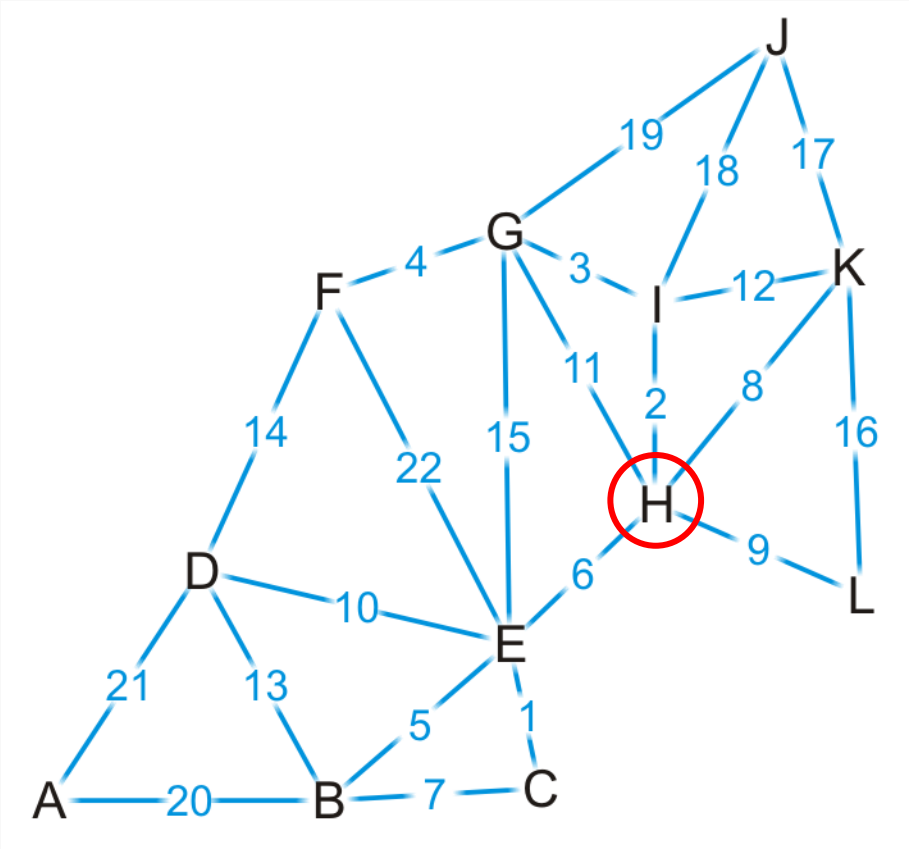


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

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:

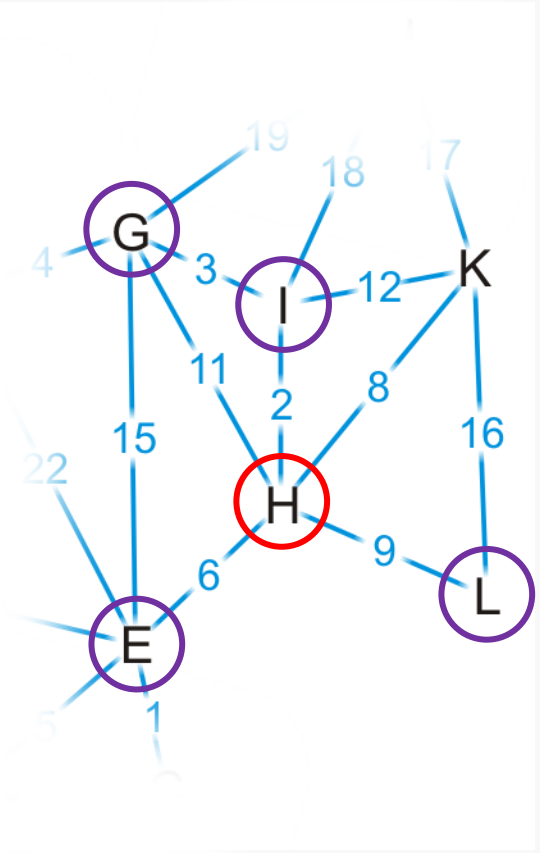
(K, H, E) of length 8 + 6 = 14

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

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

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

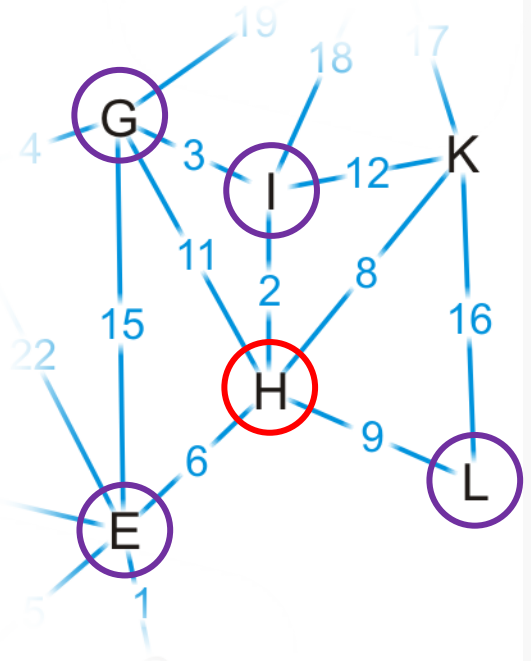
- Which of these are shorter than any known path?



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

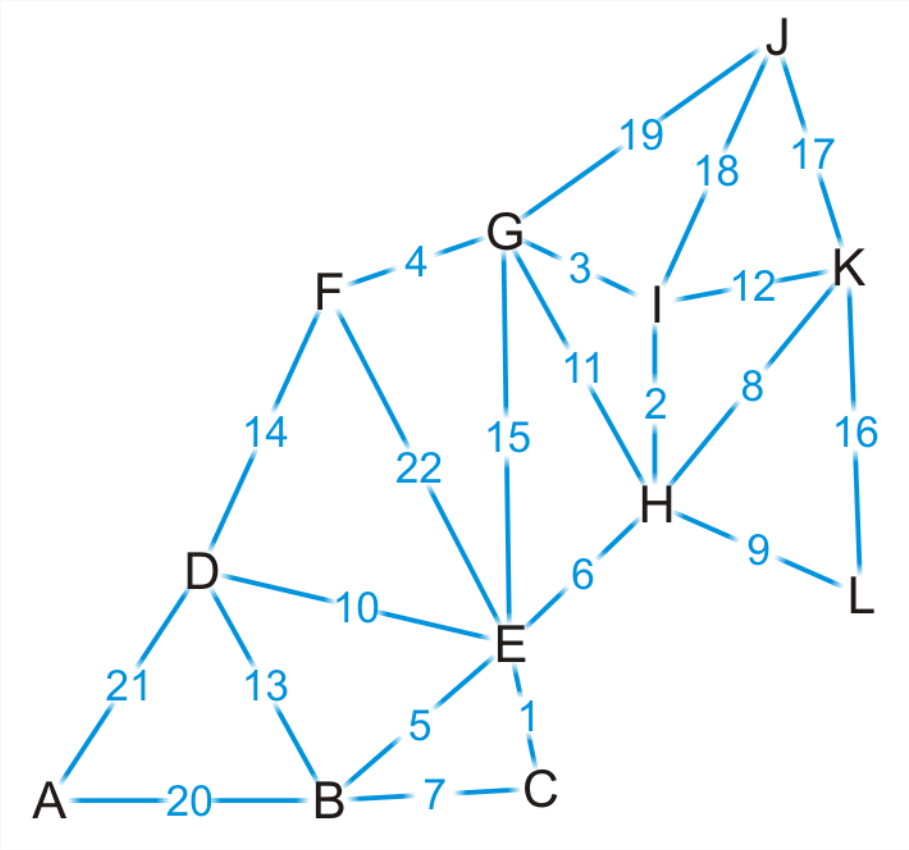


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	**14**	**H**
F	F	∞	Ø
G	F	**19**	**H**
H	**T**	**8**	**K**
I	F	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

We are finished with vertex H

- Which vertex do we visit next?

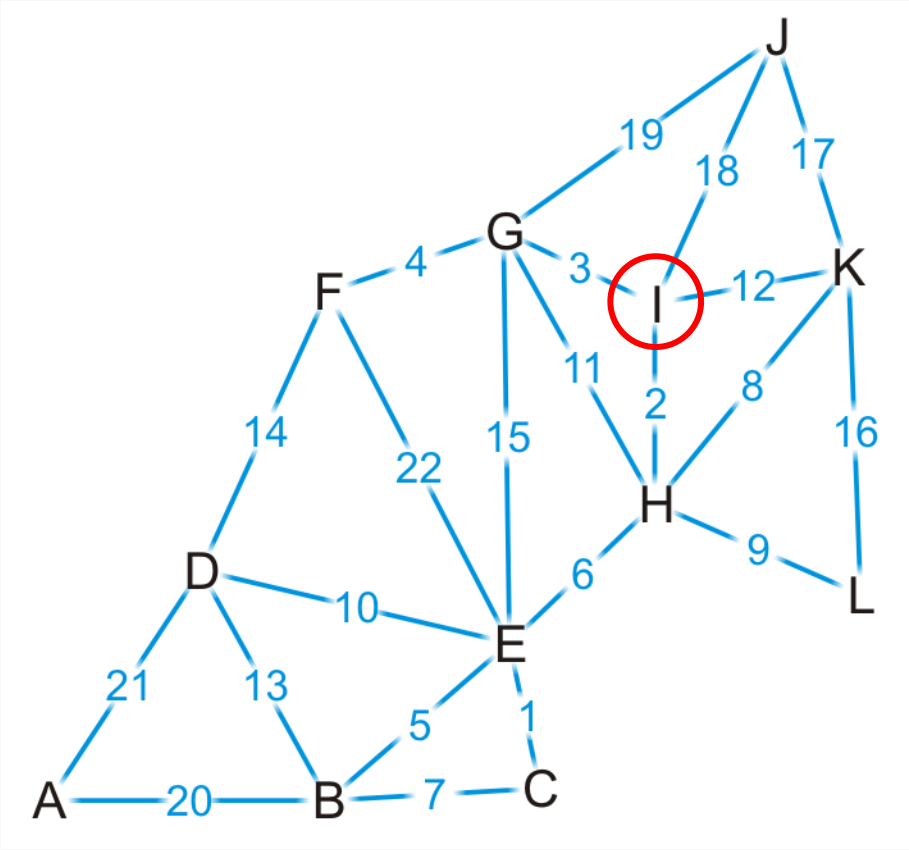


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	F	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

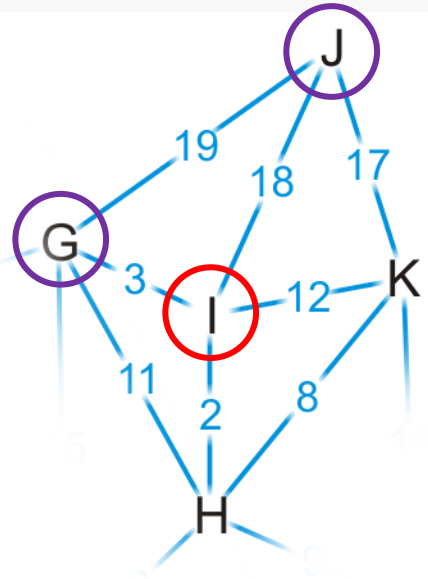


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

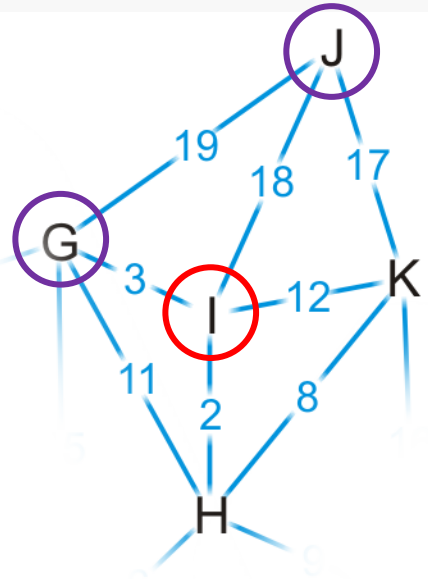
(K, H, I, G) of length 10 + 3 = 13 (K, H, I, J) of length 10 + 18 = 28



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

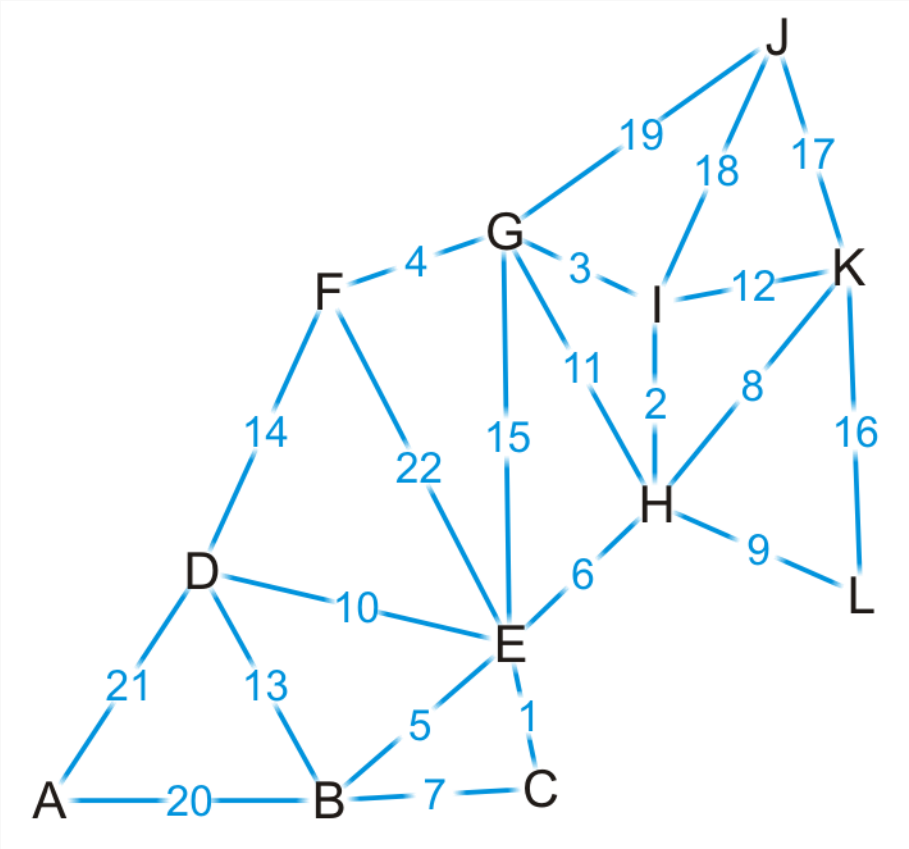
We have discovered a shorter path to vertex G, but (K, J) is still the shortest known path to vertex J



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?

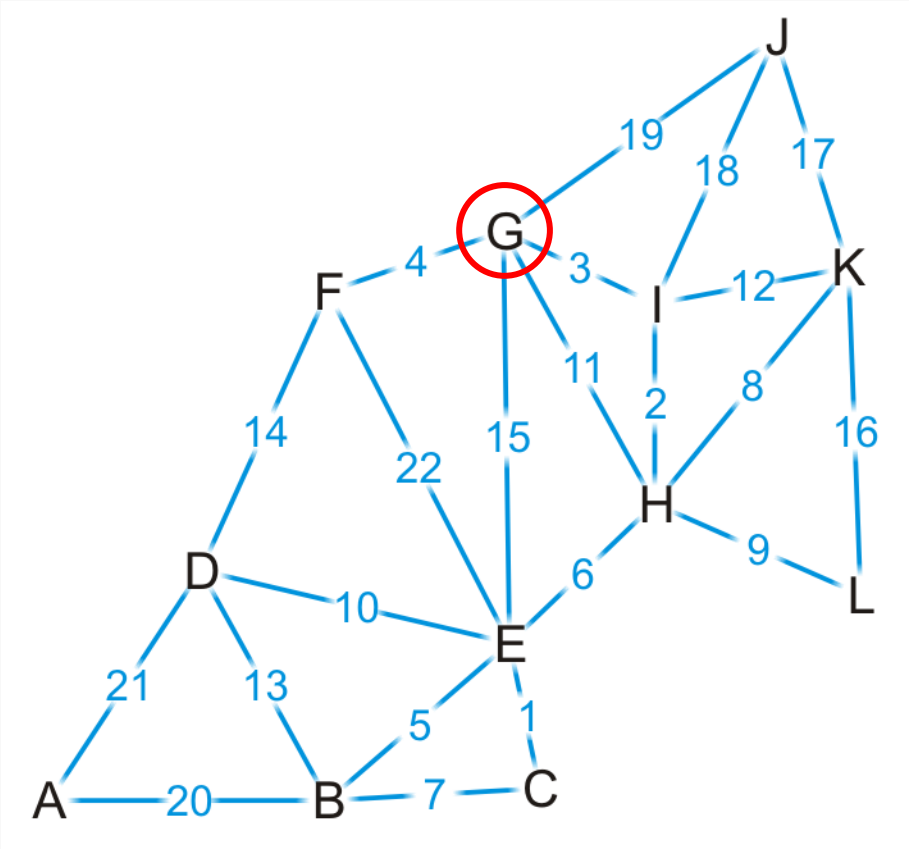


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

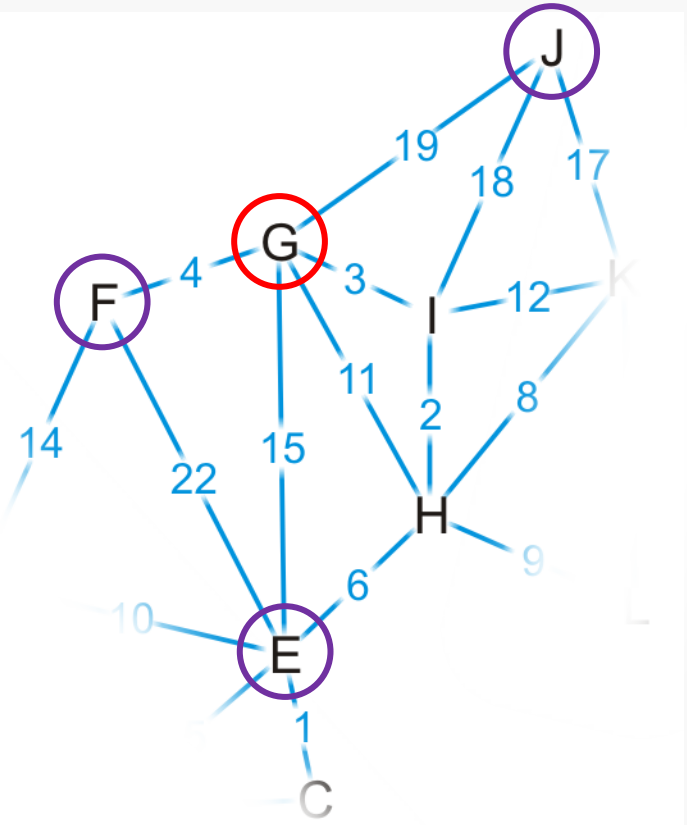
Consider these paths:

(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

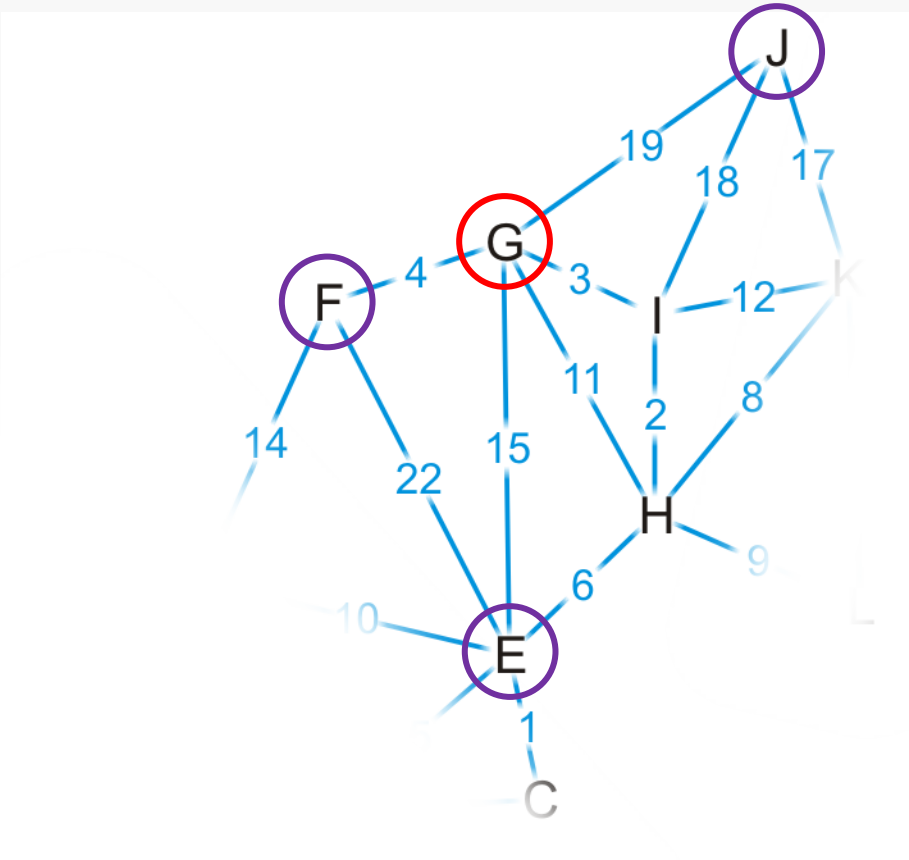
- Which do we update?



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

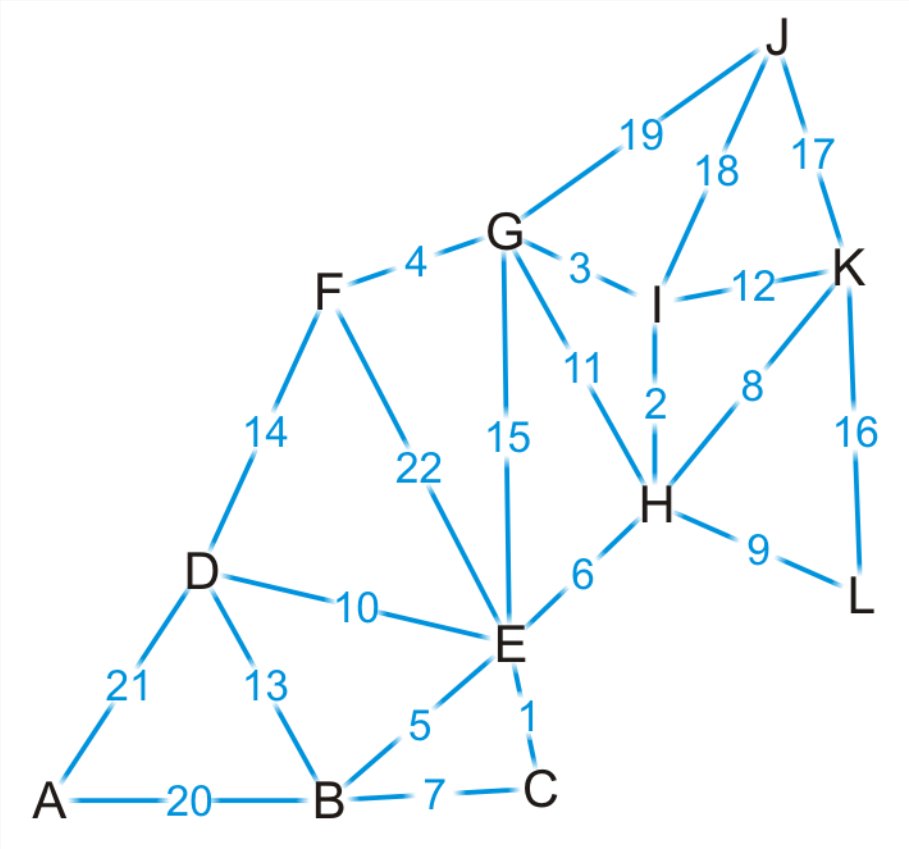
We have now found a path to vertex F



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?

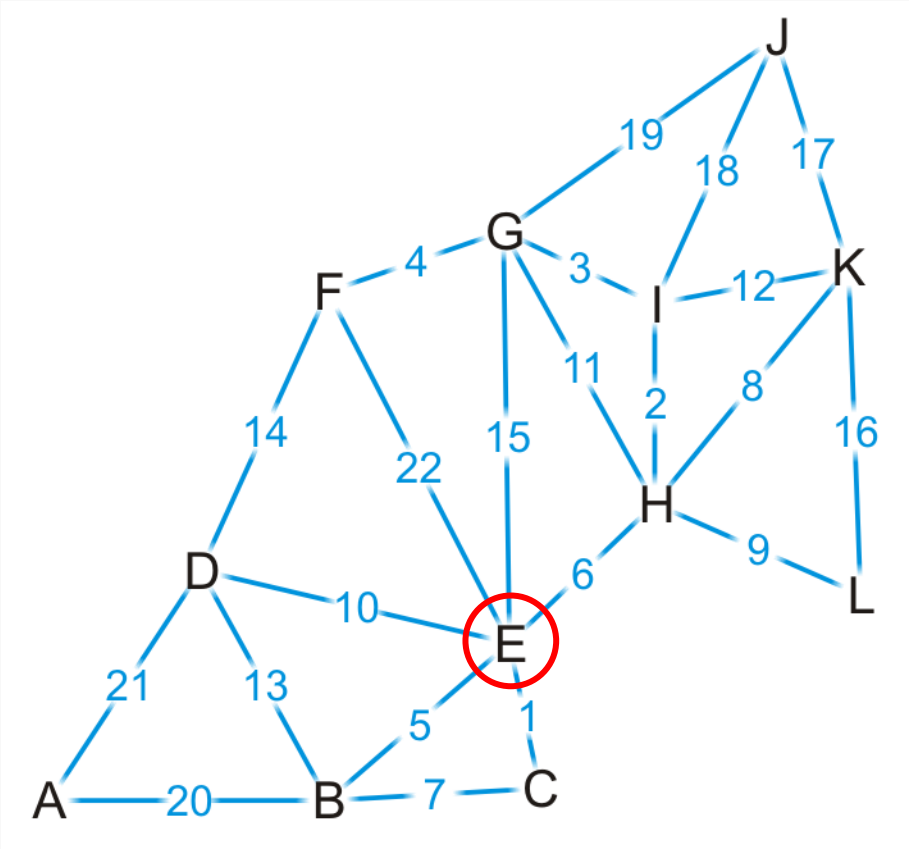


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

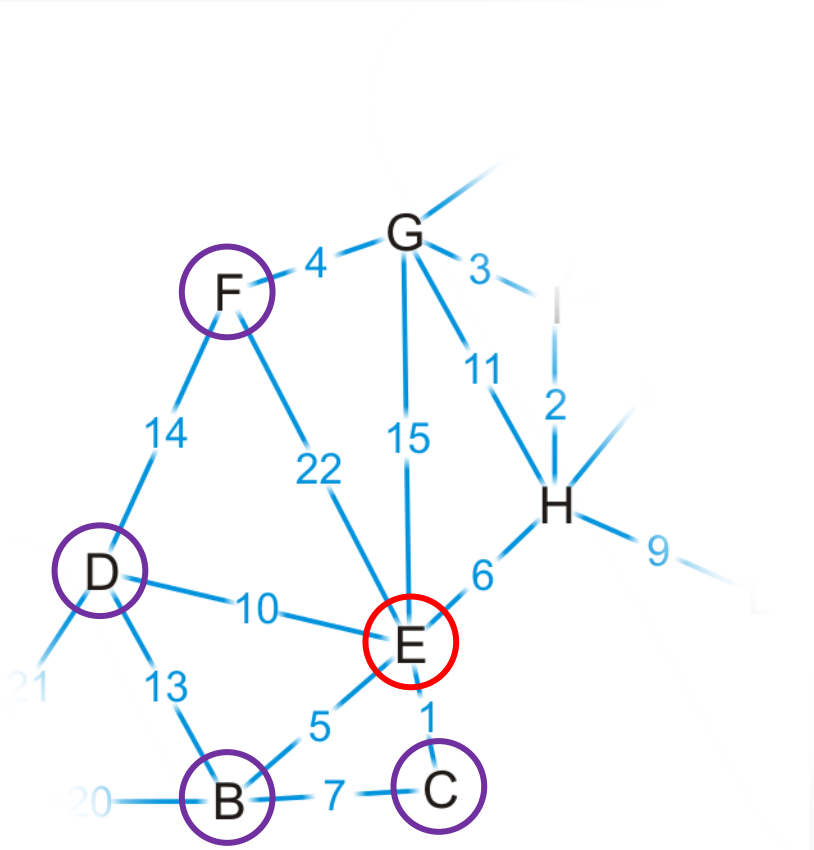


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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:

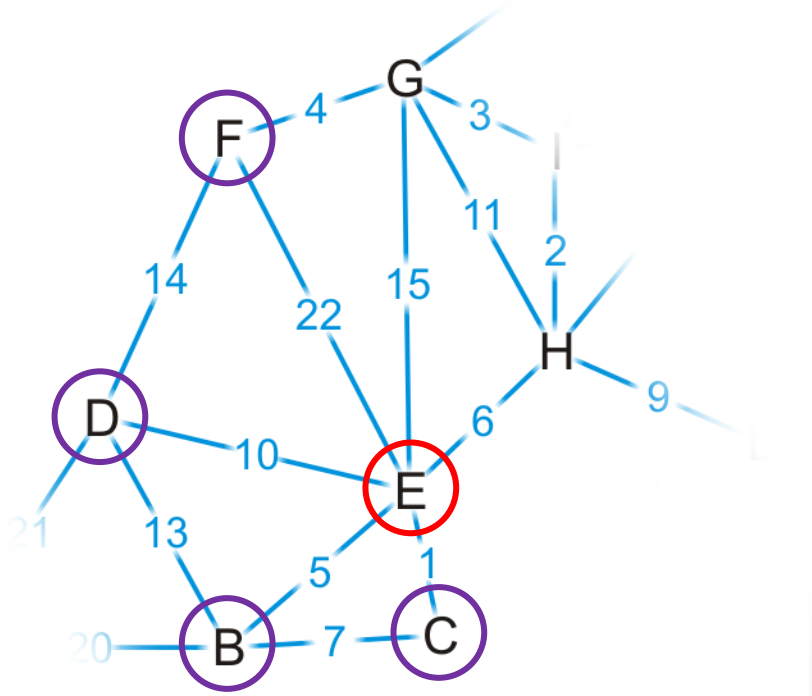
(K, H, E, B) of length 14 + 5 = 19

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

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

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

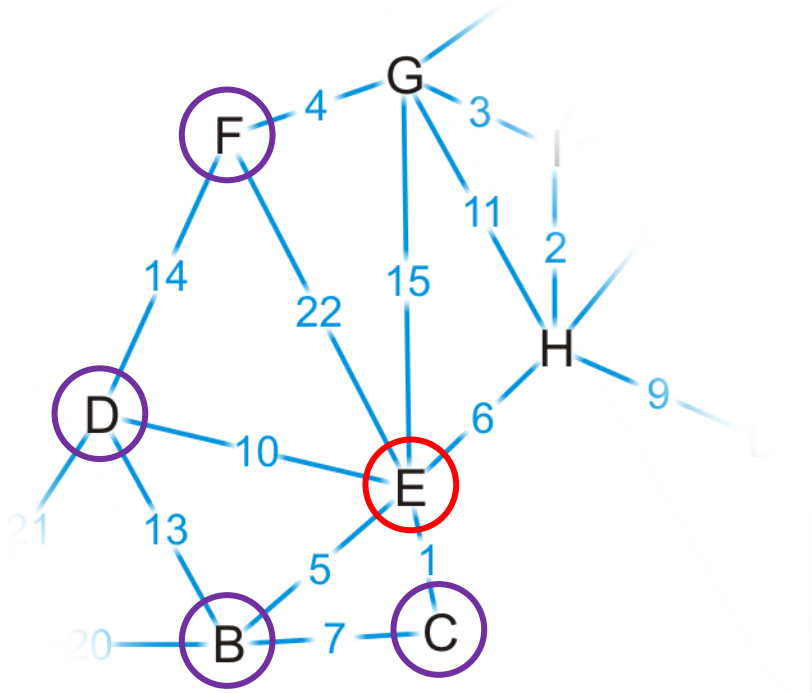
- Which do we update?



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

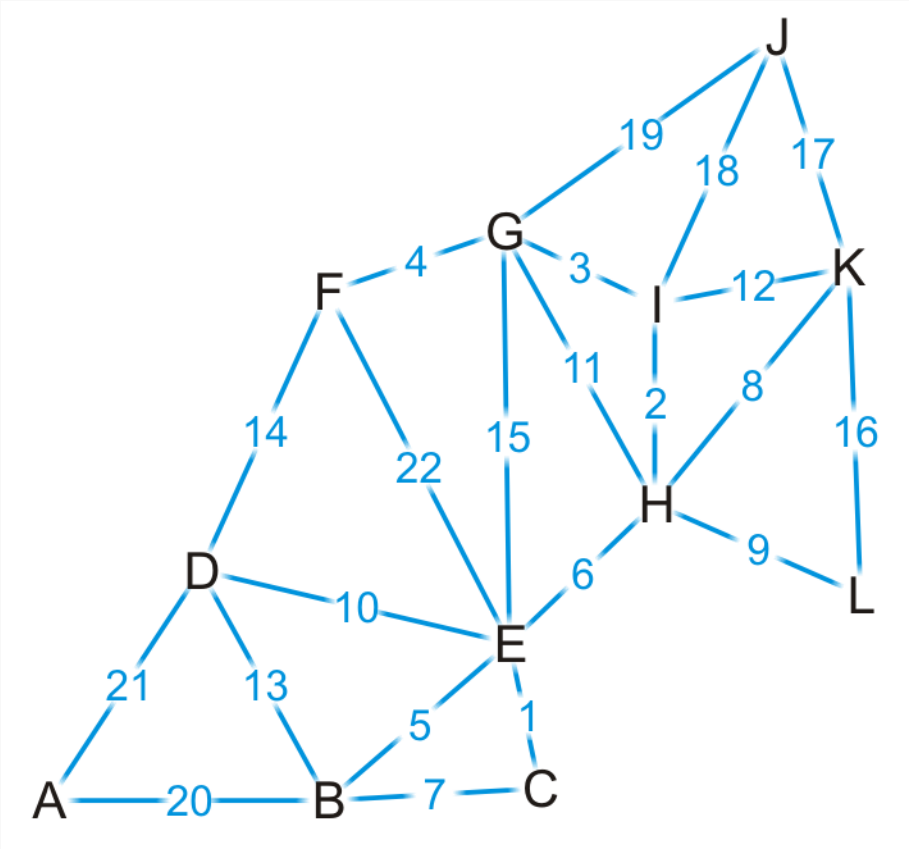
We've discovered paths to vertices B, C, D



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?

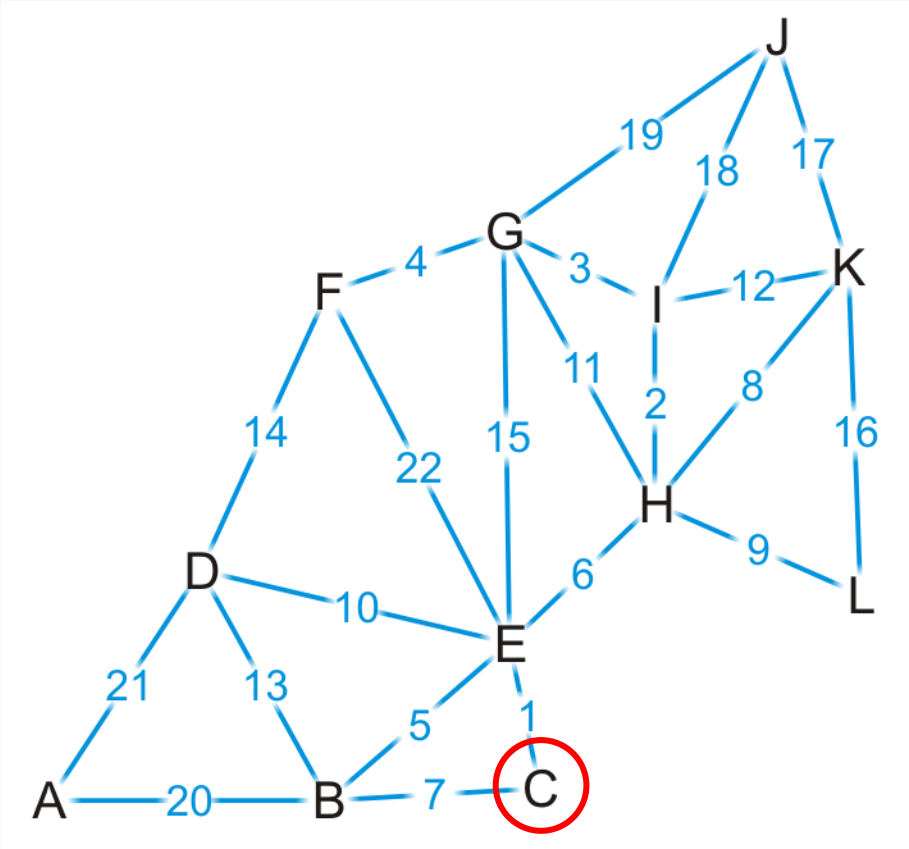


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	F	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

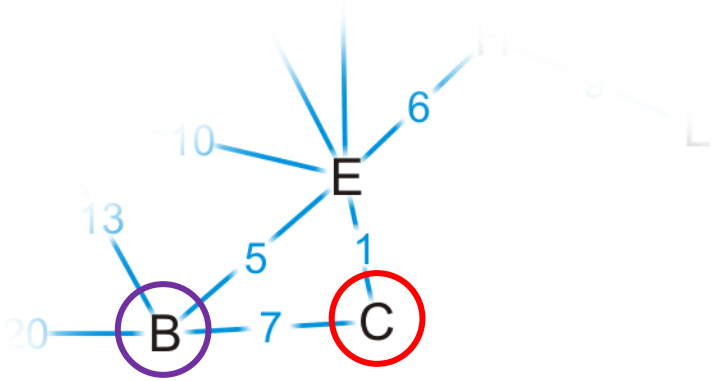


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

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

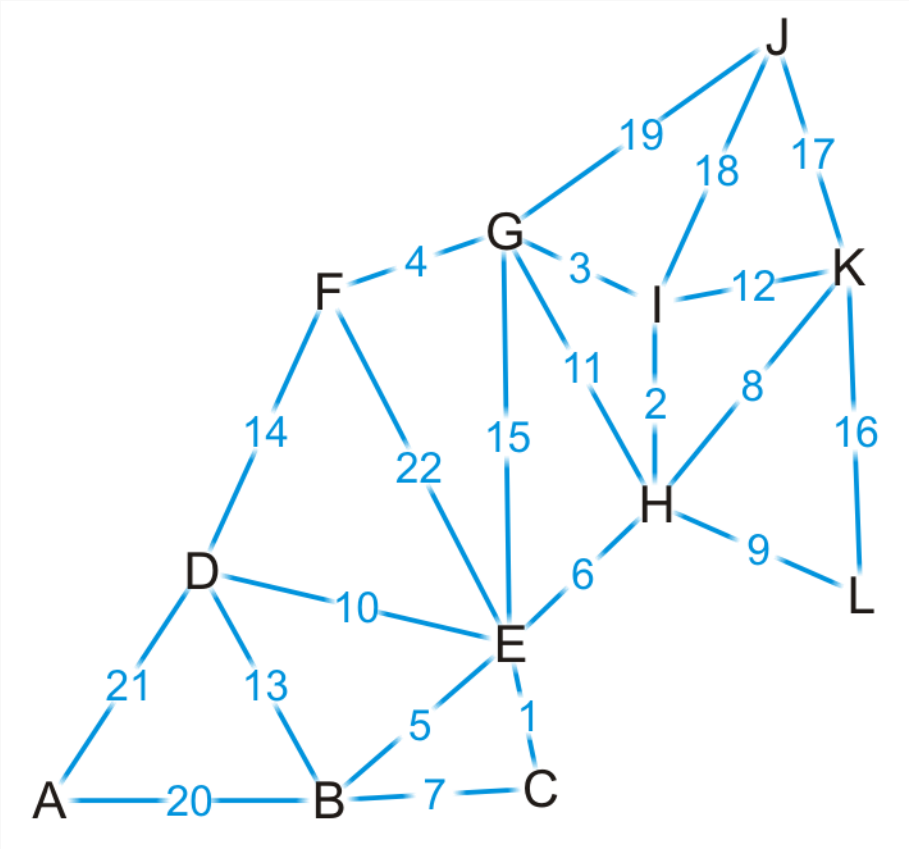
- We have already discovered a shorter path through vertex E



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?

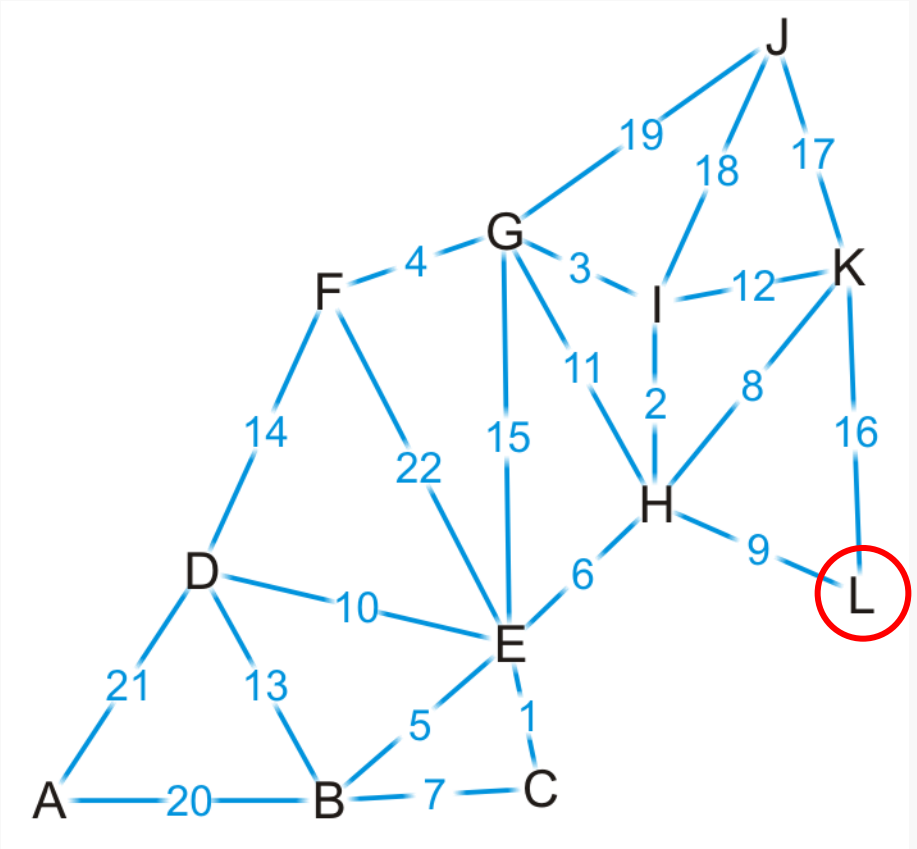


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

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

- Vertex L has no unvisited neighbors

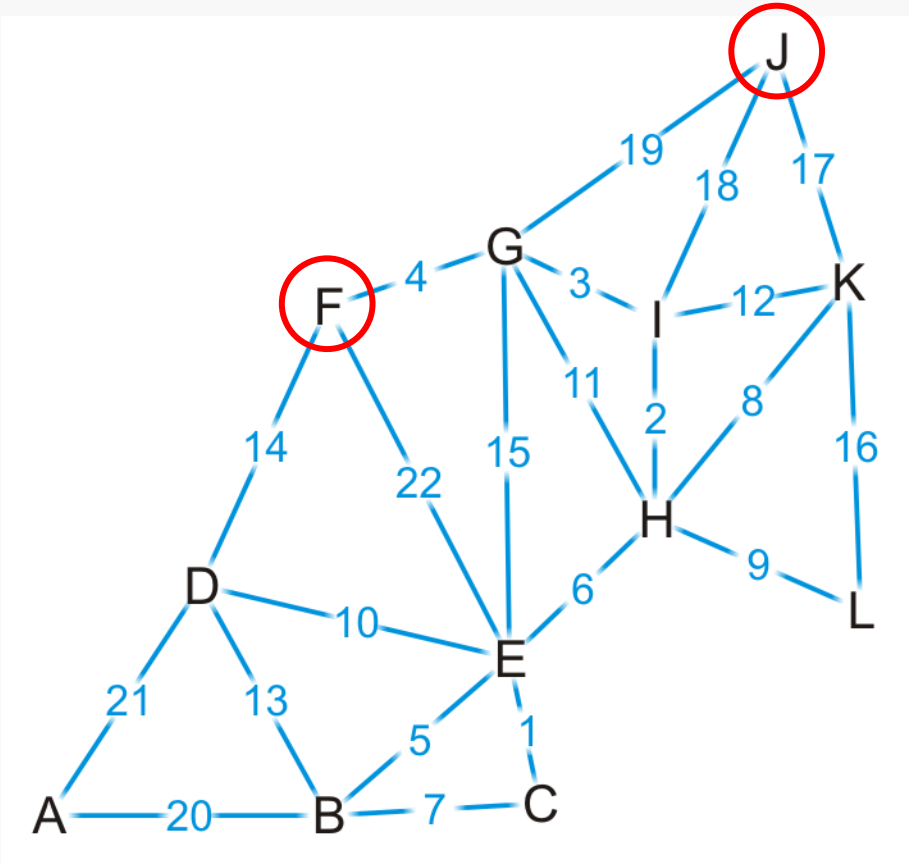


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	**T**	**16**	**K**

Example

Where to next?

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

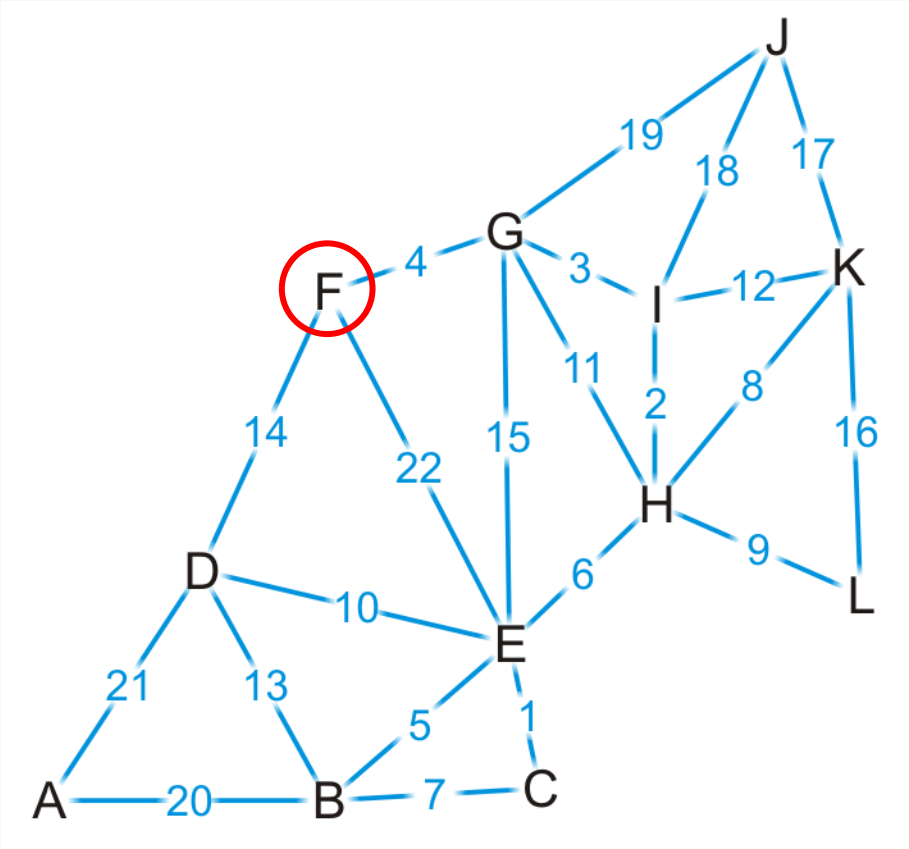


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Let's visit vertex F first

- It has one unvisited neighbor, vertex D

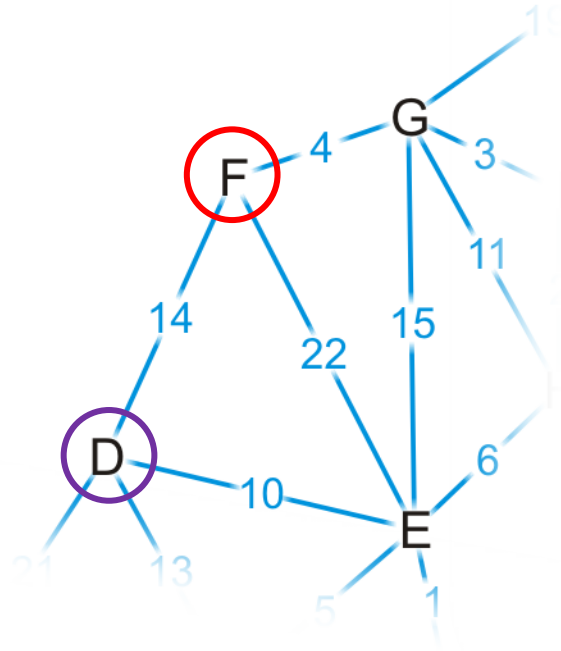


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

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

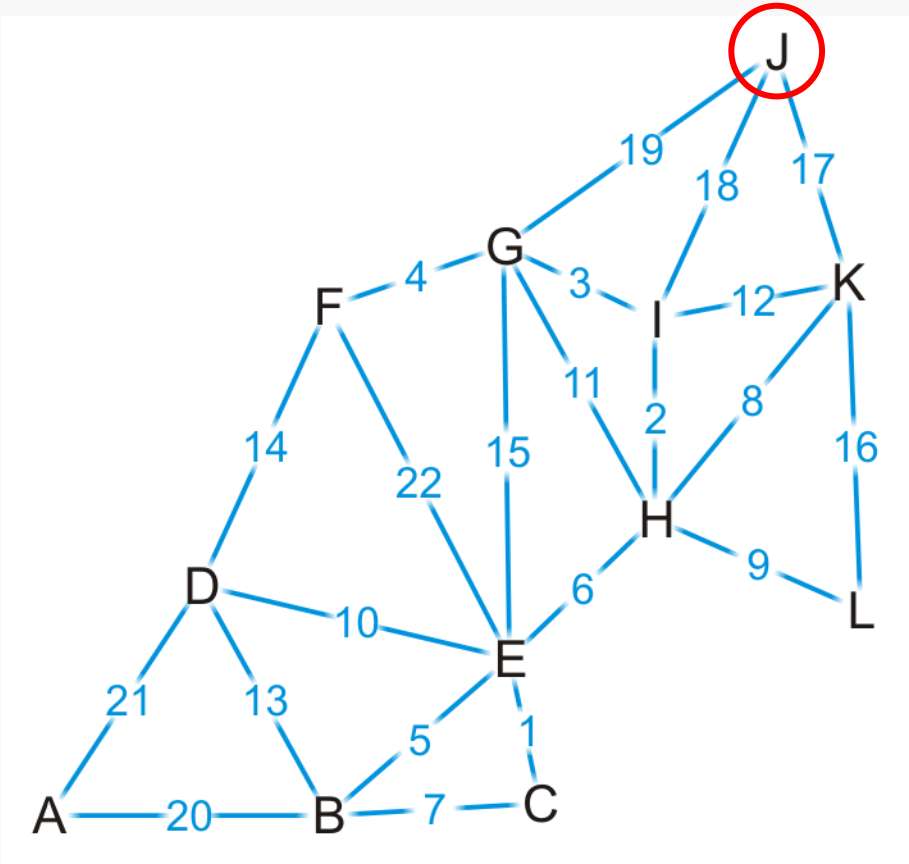


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

Example

Now we visit vertex J

- It has no unvisited neighbors



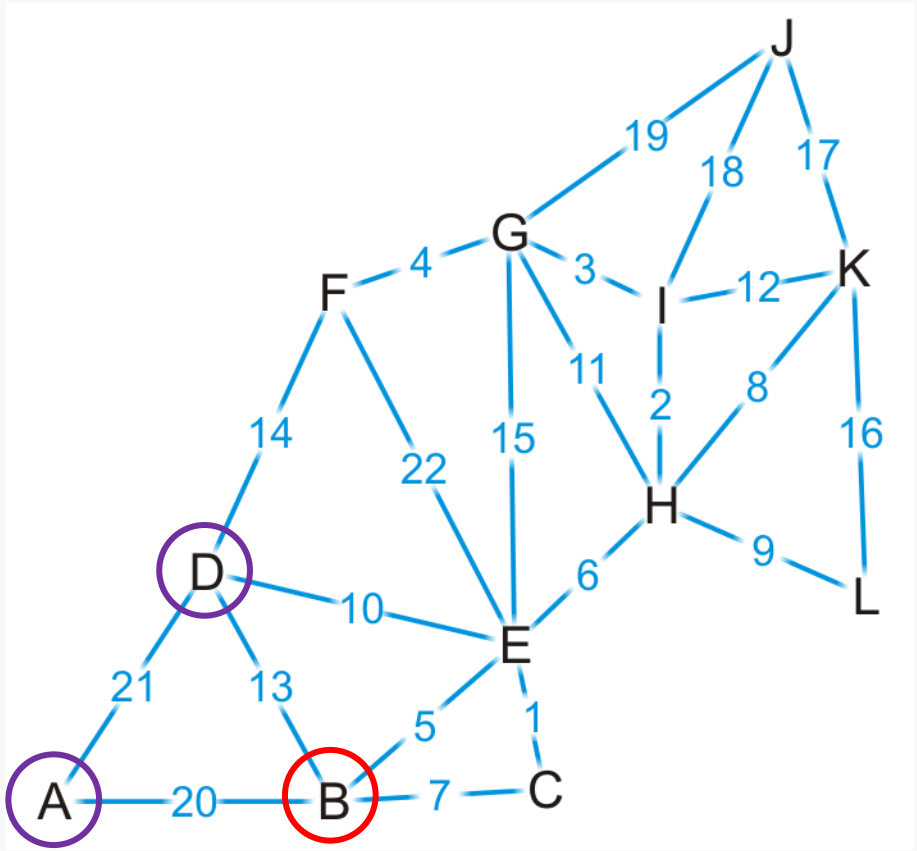
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	**T**	**17**	**K**
K	T	0	Ø
L	T	16	K

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

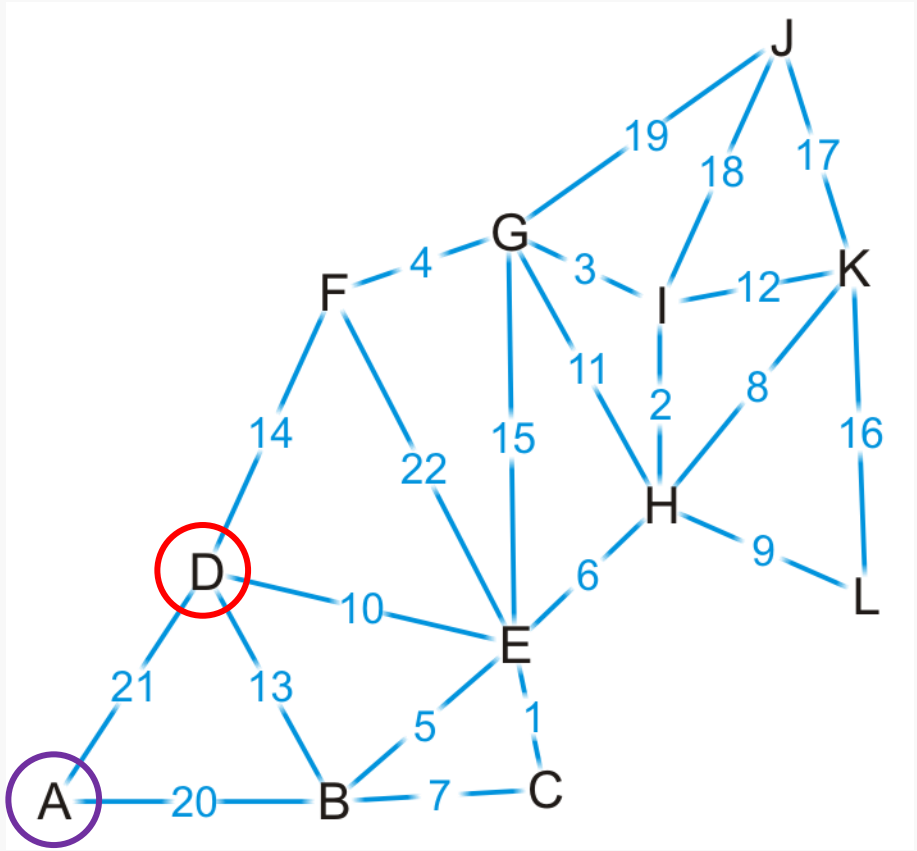


Vertex	Visited	Distance	Previous
A	F	**39**	**B**
B	**T**	**19**	**E**
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex D

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

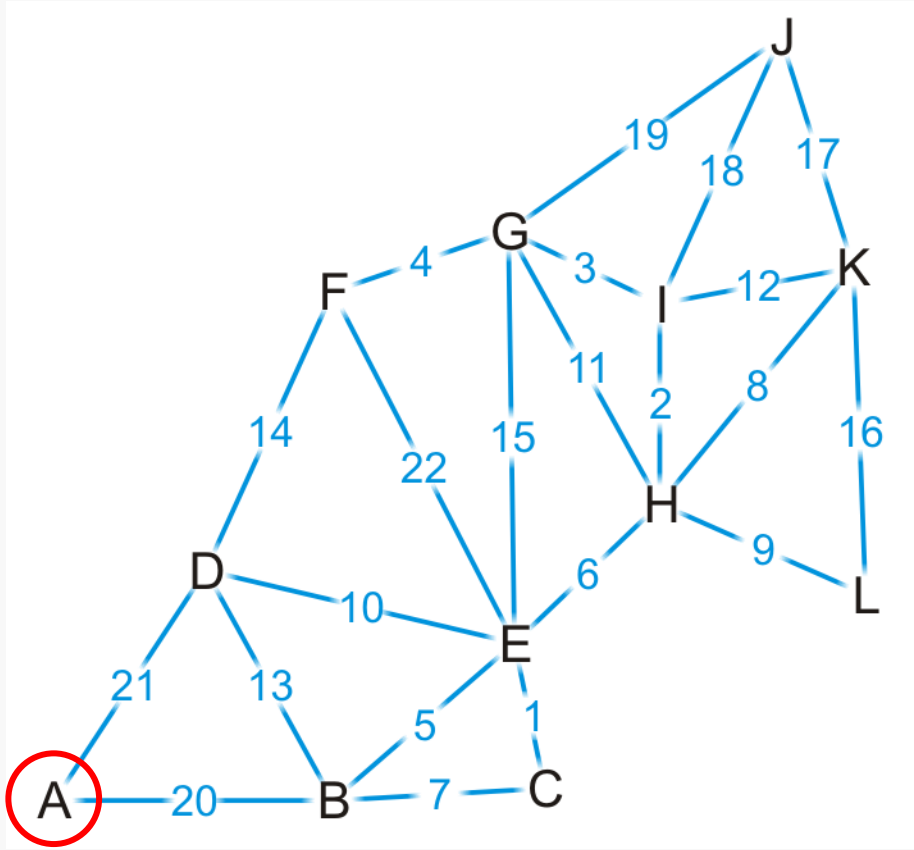


Vertex	Visited	Distance	Previous
A	F	39	B
B	T	19	E
C	T	15	E
D	**T**	**24**	**E**
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Finally, we visit vertex A

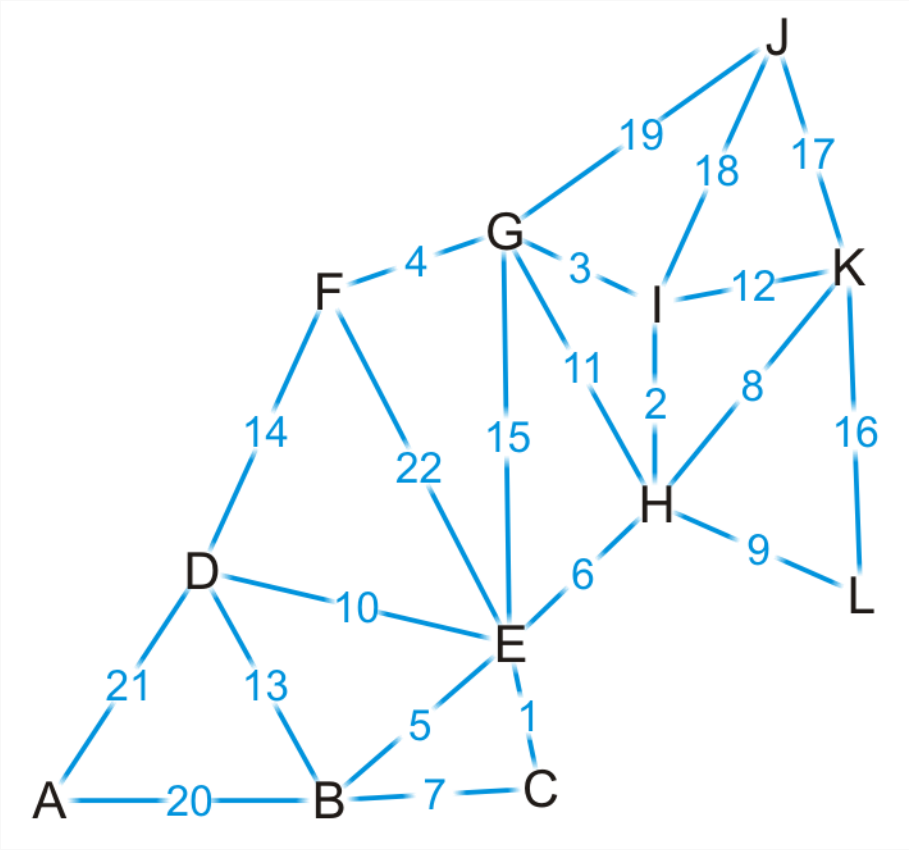
- It has no unvisited neighbors and there are no unvisited vertices left
- We are done



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

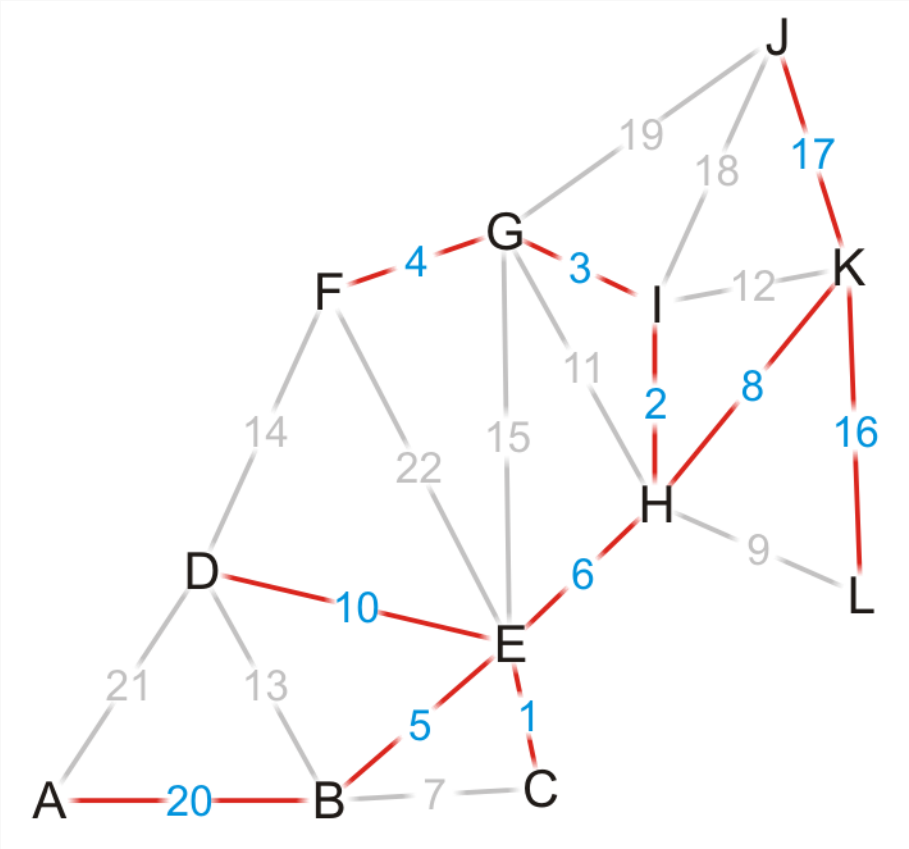
Thus, we have found the shortest path from vertex K to each of the other vertices



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

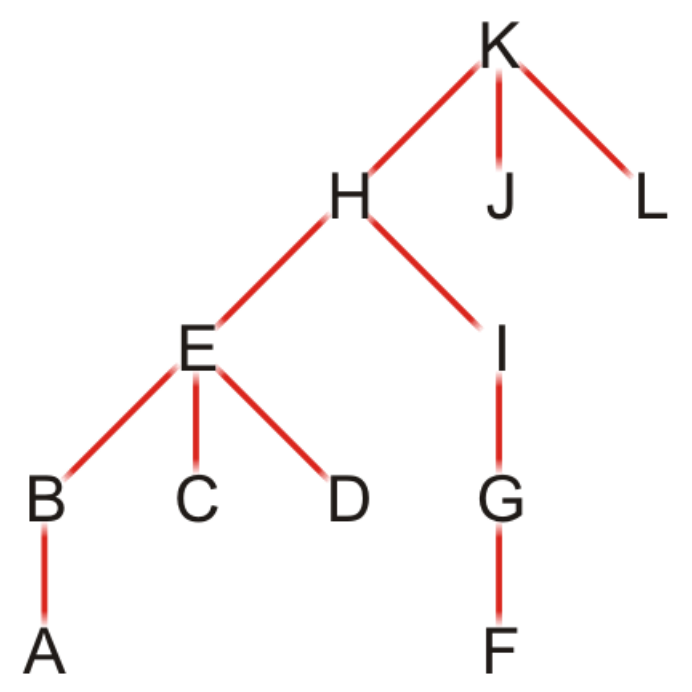


Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

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

Java Code

- **Listing 14.2: The path.java Program** (page 703)
- Read the code explanation first (page 698)
  - **sPath** Array
  - **DistPar** Class
  - **path()** method
  - **getMin()** function

Vietnam National University of HCMC
International University
School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io