COSC160: Data Structures
Graph Structures

Jeremy Bolton, PhD
Assistant Teaching Professor

Supplemental Slides provided by A. Gates and L. Singh

A special thanks to D. Harder for use of presentation material.

GEORGETOWN UNIVERSITY

Outline

I. Graphs vs. Trees
    I. Terminology
        I. Paths
    II. Traversals
        I. Class Exercise: Design a Traversal Scheme
        II. DFS
        III. BFS
II. Paths
III. Implementations
IV. Applications
    I. Maps / Networks
    II. Matching Problem

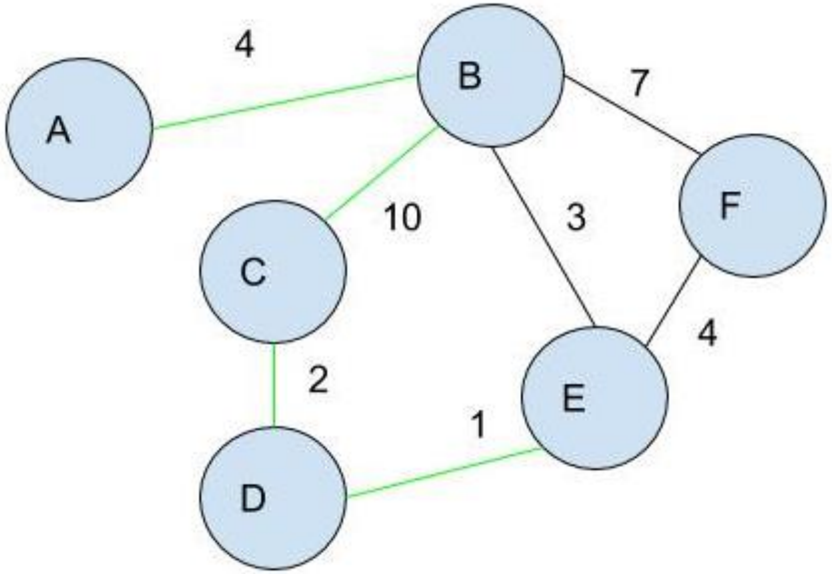
GEORGETOWN UNIVERSITY

Graphs

- Definition:
  - A graph is a 2-tuple: G = (N,E)
  - N is a set of nodes
  - E is a set of edges
- Note a tree is a type of graph
  - With added constraints

Graph Terminology

- An edge e is **incident** on a node n_1 if e = (n_1,n_i) or e = (n_i, n_1)
  - A directed edge e emanates from n1 if e = (n_1,n_i)
  - A directed edge e terminates at n1 if e = (n_i, n_1)
- A **path** on a graph between two nodes n_1 and n_i , is a sequence of edges e_1, e_2, … ,e_j where e_1 emanates at n_1 and e_i terminates at n_i, and all intermediate edges e_k are appropriately connected, ie, e_k terminates at n_{k+1} and e_{k+1} emanates from n_{k+1}.
- A **loop** is a path that emanates and terminates at the same node.
- A **simple path** is a path that contains no loops



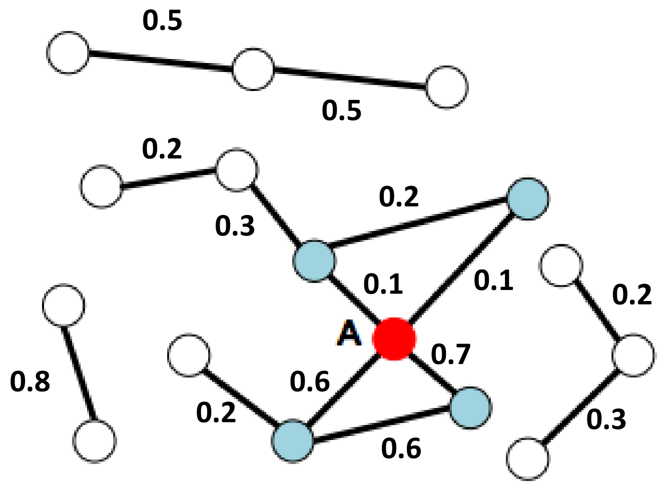
(A,B) , (B,C) , (C,D) , (D,E)

OR (more compactly)

A – B – C – D – E

Graph Terminology

- A **directed graph** consists of edges which have implied direction.
- An **undirected graph** consists of edges without implied direction.
- A **mixed graph** consists of edges with and without direction.
- An **attributed graph** is a graph where attributes are associated with the edges or nodes (usually the edges)
- A **weighted graph** is a graph with weight attributes associated with the edges.



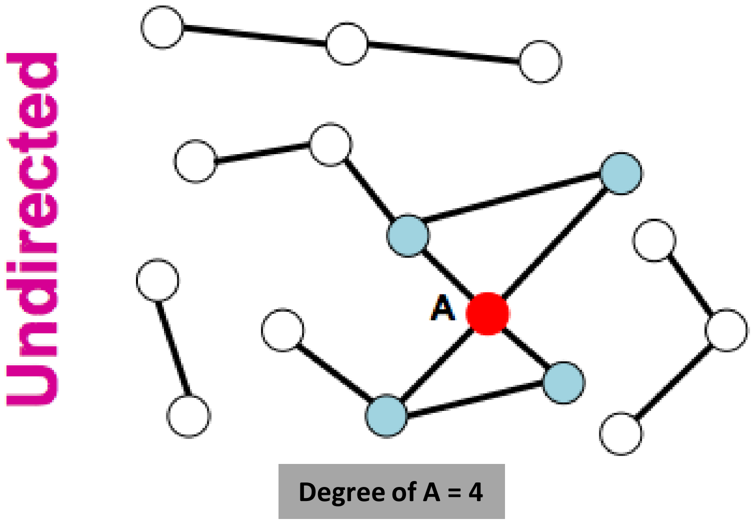
Edge Weights

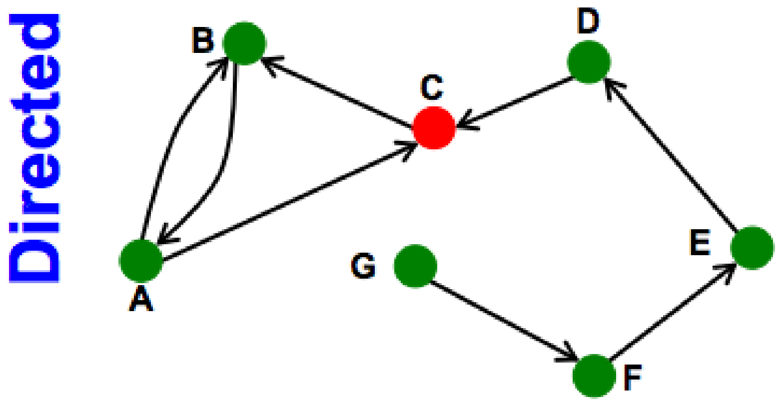
Quantify the relationship between two nodes.

GEORGETOWN UNIVERSITY

Graph Terminology

- A graph is **connected** if there exists a path from any node to any other node.
- A **fully connected** simple graph is a graph with the maximum number of edges (*Assuming it is not a multi-graph*!):
  - $(n-1)^2$ edges: with no self-loops.
  - $(n)^2$ edges: with self-loops
- A **simple graph** is a graph such that there is never multiple edges connected the same node pair.
- A **multigraph** is a graph where there exists multiple edges connecting the same node pair.
- The **order** or **degree** of a node is the number of edges incident upon it.
  - In-degree: the number of edges terminating at a node
  - Out-degree: the number of nodes emanating at a node





Node Degree:

The number of neighbors an Individual node has.

In directed graphs, we have in-degrees and out-degrees.

- **Sink**: nodes with out-degree = 0
- **Source:** nodes with in-degree = 0

Images: Jure Leskovec

Implementation of a Graph

- How might we implement a Graph Structure?

- Chaining:
  - Nodes and pointers

- Array:
  - Adjacency Matrix

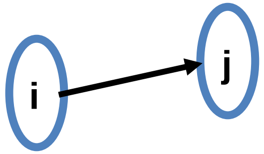
- Efficient (chain or array):
  - Sparse matrix

GEORGETOWN UNIVERSITY

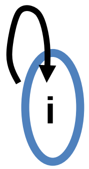
Different Ways to Represent a Graph: Adjacency Matrix M

- Representing edges (who is adjacent to whom) as a matrix

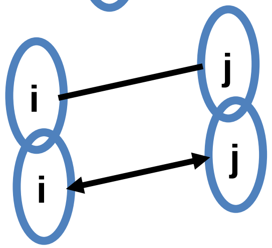
  - M_{ij} = 1 if node i has an edge to node j
    = 0 if node i does not have an edge to j



  - M_{ii} = 1 if the network has self-loops

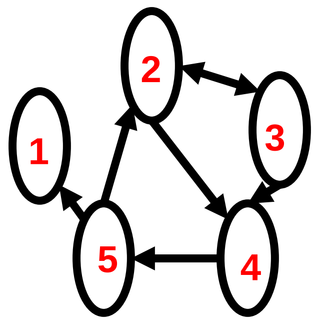


  - $M_{ij} = M_{ji}$ if the network is undirected,
    or if i and j share a reciprocated edge



GEORGETOWN UNIVERSITY

Adjacency Matrix Example

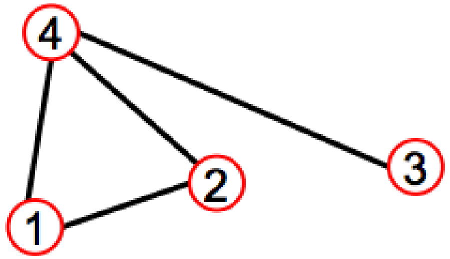


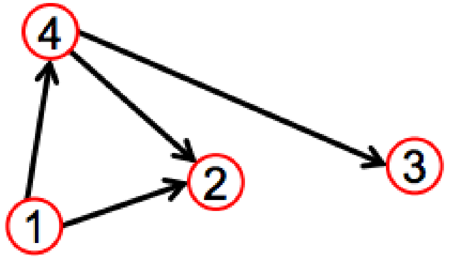
M =

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	1	1	0
3	0	1	0	1	0
4	0	0	0	0	1
5	1	1	0	0	0

GEORGETOWN UNIVERSITY

Compute the Adjacency Matrices





M =

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

SYMMETRIC

	1	2	3	4
1	0	1	0	1
2	0	0	0	0
3	0	0	0	0
4	0	1	1	0

NOT SYMMETRIC

Analysis of Adjacency Matrix Implementation

- Space requirements
  - $O(N^2)$ where N is the number of nodes

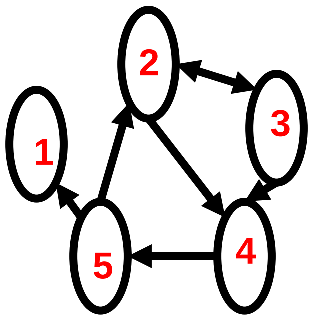
- Time requirements
  - Creation / initialization: $O(N^2)$

- In many applications, graphs are very sparse!
  - A sparse representation may be more efficient.

Different Ways to Represent a Graph – Adjacency List

Keep track of all the edges in the graph

- Edge Set
  2 3
  2 4
  3 2
  3 4
  4 5
  5 2
  5 1
- Node Set with edges
  1:
  2: 3 4
  3: 2 4
  4: 5
  5: 1 2



Adjacency List Implementation (Sparse)

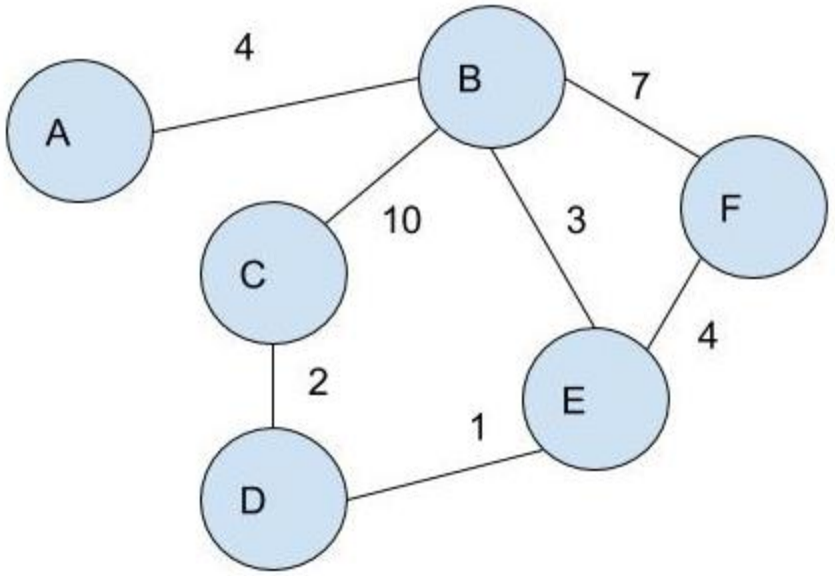
- Space Requirements:
  - O(N+E), where E ≤ N^2 is the number of edges
    - Inequality holds assuming there are no repeated edges (with different weights)
    - The number of edges is quite low in sparse graphs

- Time Requirements:
  - Creation / initialization: O(N+E), where E is the number of edges

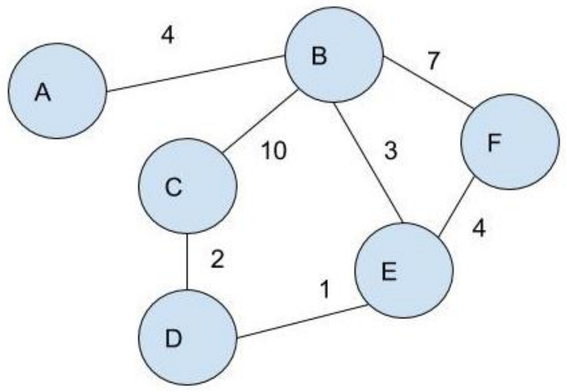
Traversing a Graph

- Class Discussion:
  - Design a graph traversal algorithm assuming graph is connected.

- Notes: Similar to tree, but there may be cycles!
  - Thus must assure no looping during traversal



Traversing a Graph



- DFS

$function\ DFS(node)$
$\quad \text{stack}.push(node)$
$\quad while(\ stack\ is\ not\ empty\ )$
$\quad\quad thisNode\ \coloneqq stack.pop(\ \ )$
$\quad\quad for\ all\ nodes\ c\ adjacent\ to\ thisNode\ \boldsymbol{that\ have\ not\ been\ previously\ visited}$
$\quad\quad\quad \text{if c}\ is\ not\ null, stack.push(c)$

- BFS

$function\ BFS(node)$
$\quad \text{queue}.add(node)$
$\quad while(\ queue\ is\ not\ empty\ )$
$\quad\quad thisNode \coloneqq queue.dequeue(\ \ )$
$\quad\quad for\ all\ nodes\ c\ adjacent\ to\ thisNode\ \boldsymbol{that\ have\ not\ been\ previously\ visited}$
$\quad\quad\quad if\ \ c\ is\ not\ null,\ \ queue.add(c)$

GEORGETOWN UNIVERSITY

Single Source Path Length: unweighted graphs

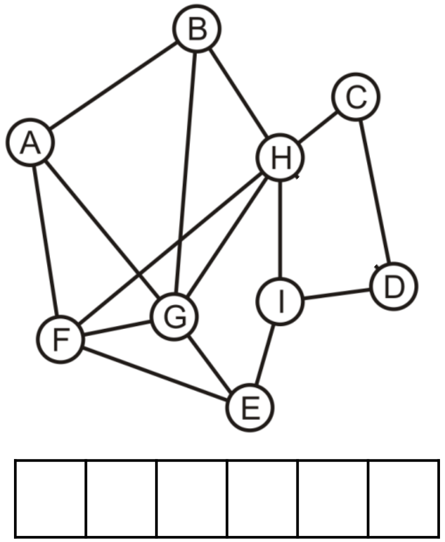
Problem: find the distance from one vertex v to all other vertices

- Use a breadth-first traversal
- Vertices are added in *layers*
- The starting vertex is defined to be in the zeroeth layer, L_0
- While the k^{th} layer is not empty:
  - All unvisited vertices adjacent to verticies in L_k are added to the $(k + 1)^{\text{st}}$ layer

Any unvisited vertices are said to be an infinite distance from v

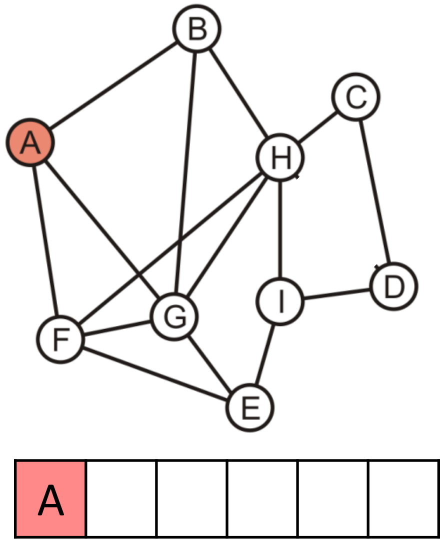
Determining Distances

Consider this graph: find the distance from A to each other vertex



Determining Distances

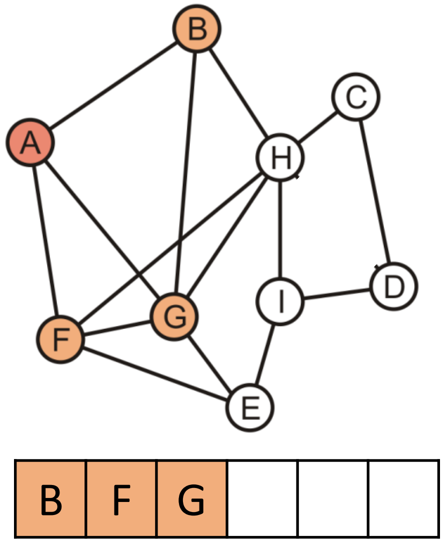
A forms the zeroeth layer, L_0



Determining Distances

The unvisited vertices B, F and G are adjacent to A

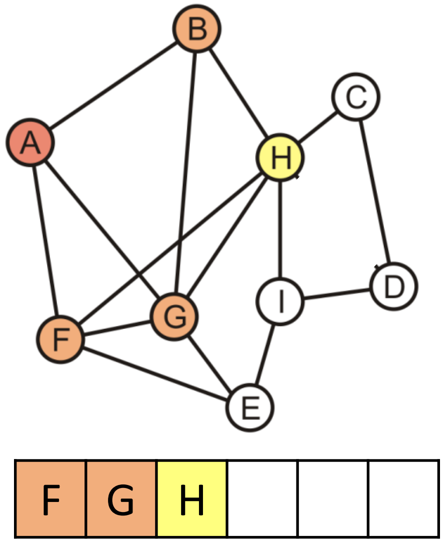
– These form the first layer, I



Determining Distances

We now begin popping L_1 vertices: pop B

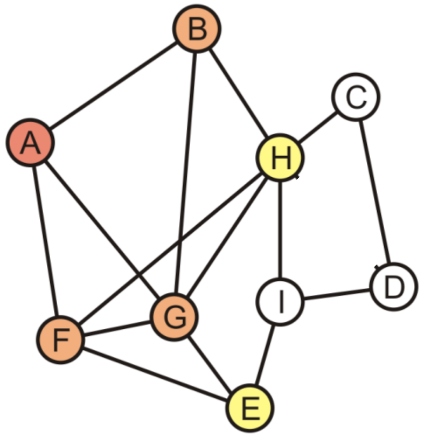
- H is adjacent to B
- It is tagged L_2

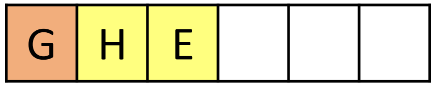


Determining Distances

Popping F pushes E onto the queue

– It is also tagged L_2

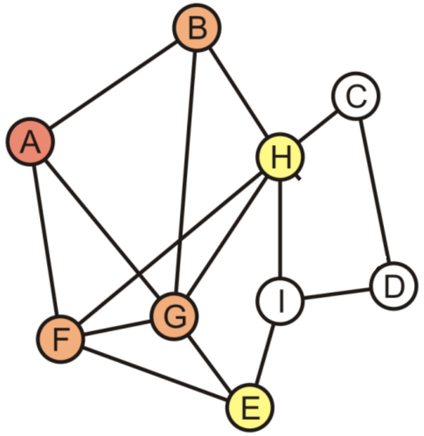




Determining Distances

We pop G which has no other unvisited neighbours

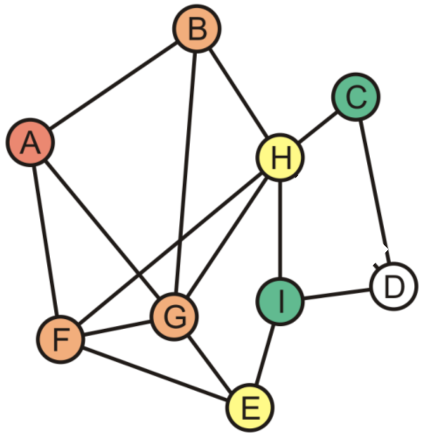
- G is the last L_1 vertex; thus H and E form the second layer, L_2



H	E				

Determining Distances

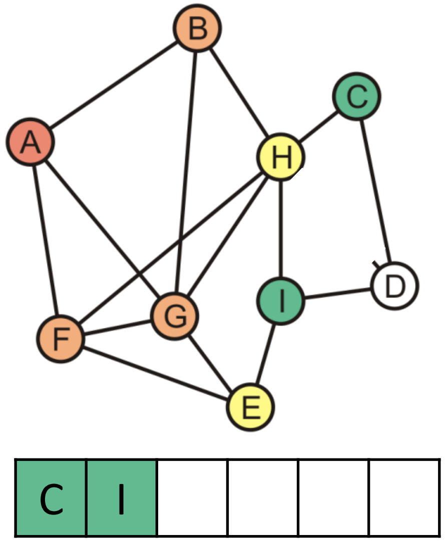
Popping H in L_2 adds C and I to the third layer L_3



E	C	I			

Determining Distances

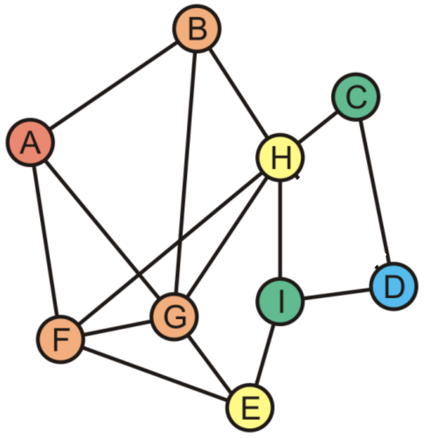
- E has no more adjacent unvisited vertices
  - Thus C and I form the third layer, L_3



Determining Distances

The unvisited vertex D is adjacent to vertices in L_3

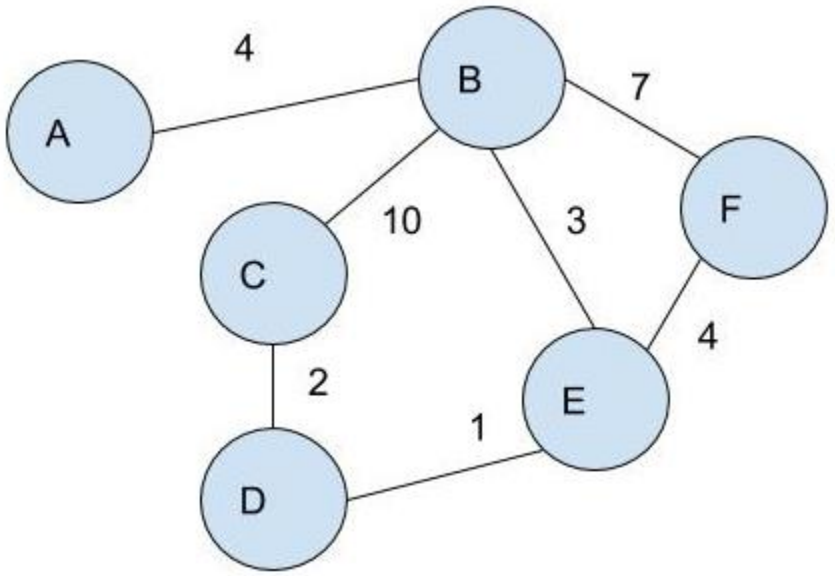
- This vertex forms the fourth layer, L_4



Finding Shortest Paths from 1 source: weighted graphs

- Class Discussion:
  - Given a graph, design an algorithm to find the shortest path between the two nodes
  - What is the shortest path between
    - A and C?
    - A and F?

- How would you do this?
  - Which scheme is most appropriate here?
    - BFS
    - DFS



Dijkstra's Algorithm: Shortest

$function\ shortestWeightedPath(s{:}\ source)$ *// computes shortest path from 1 source to all other nodes*

$N\ := list\ of\ all\ Nodes$

dist[∀ n ∈ N] := inf *// initialize distance to be inf , dist [j] is distance from source to node j*

dist[s] := 0 */ / distance to source is 0*

$V\ := \phi$ *// nodes visited*

$while\ V \neq N$

$\min := argmin_{i \notin V}(dist\ [i])$

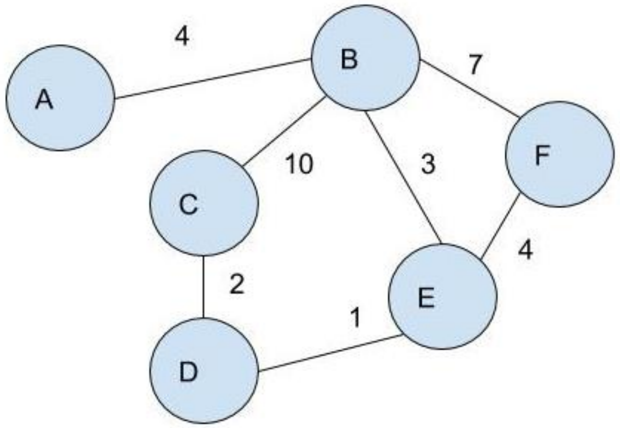
$V\ := V \cup \{min\}$

$for\ all\ nodes\ v \notin\ V\ adjacent\ to\ min$ // check all unvisited neighbors

$if\ dist[v] > dist[min] + weight(min, v)$

$dist[v] := dist[min] + weight(min, v)$ // update shortest dist

return dist



<u>A "conditional" BFS: </u>Continue BFS toward node with least aggregate weight

Example Graph Structure Implementation

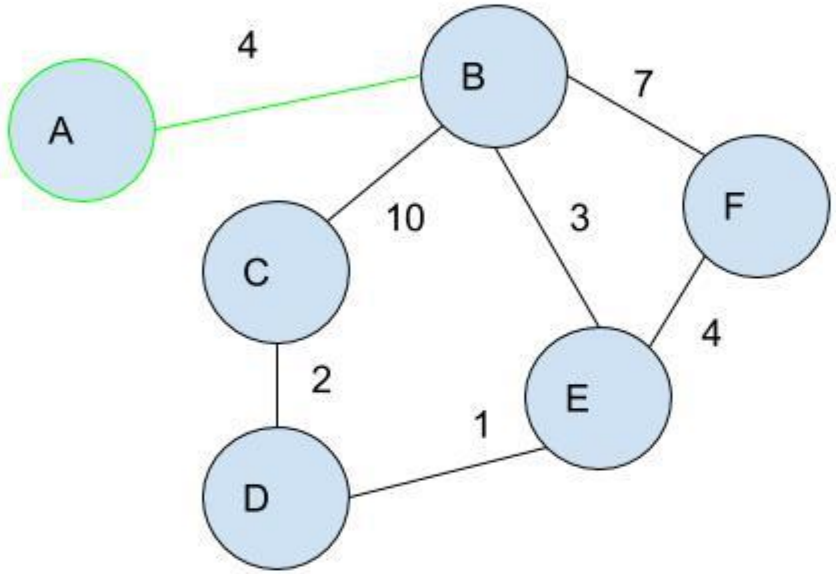
- Graph Structure: to allow for an efficient shortest path determination
  - Table with N rows, each col would hold
    - List of nodes names: implemented as a hash to allow for direct indexing
    - Adjacency List (represents edges and weights)
    - Marked (for any traversal)
      - Initialize all nodes as unmarked
      - During traversal, mark a node upon visit
    - Dist (for shortest path)
      - Keep track of shortest path from source to each node
    - Previous (for shortest path)
      - When updating shortest path, keep track of preceding node in shortest path. Allows for easy retrieval of nodes sequence of shortest path (in reverse)

```
function shortestWeightedPath(s: source) // computes shortest path from source to all other nodes
        N := list of all Nodes
        dist[∀ n ∈ N ] := inf  // initialize distance to be inf , dist [j] is distance from source to node j
        dist[s] := 0  // distance to source is 0
        V := ϕ  // nodes visited
        while V ≠ N
                min := argmin_{i∉V}(dist [i])
                V := V ∪ {min}
                for all nodes v ∉ V adjacent to min   // check all unvisited neighbors
                        if dist[v] > dist[min] + weight(min, v)
                                dist[v] := dist[min] + weight(min, v) // update shortest dist
                                prev[v] := min
        return dist
```

GEORGETOWN UNIVERSITY

Graph Structure Example:

Source is A.
Run Shortest Path and update graph table
Step 1: Visit A and Update Neighbors Distances



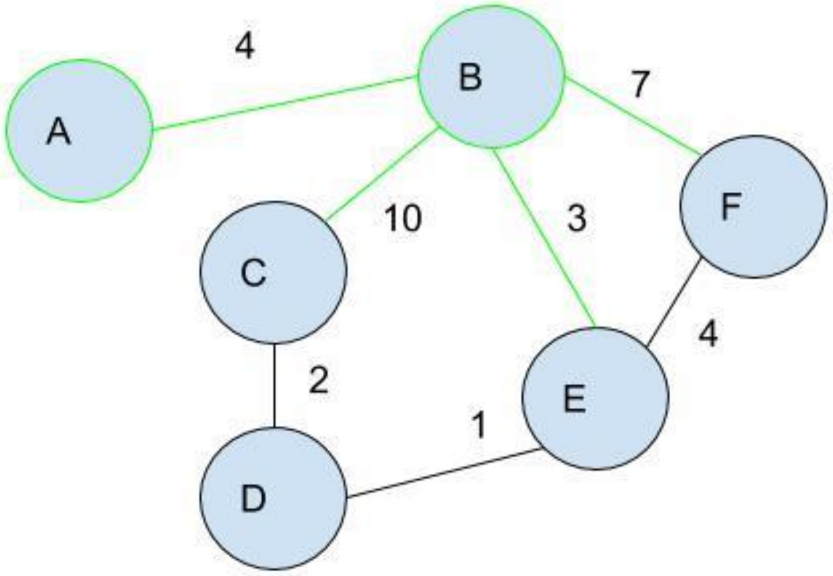
Try this at home:

1. Given a graph table implementation, try to algorithmically update the members of the table when computing the shortest path, that is implement, graph::sp(string node1, string node2)

Graph structure (already initialized)

Hash on Name	Name	Marked	Dist	Prev	adj				
0	A	1	0	NULL	LL →	B(4)			
1	B	0	4	A	LL →	A(4)	C(10)	E(3)	F(7)
2	C	0	inf	NULL	LL →	B(10)	D(2)		
3	D	0	inf	NULL	LL →	C(2)	E(1)		
4	E	0	inf	NULL	LL →	B(3)	D(1)	F(4)	
5	F	0	inf	NULL	LL →	B(7)	E(4)		

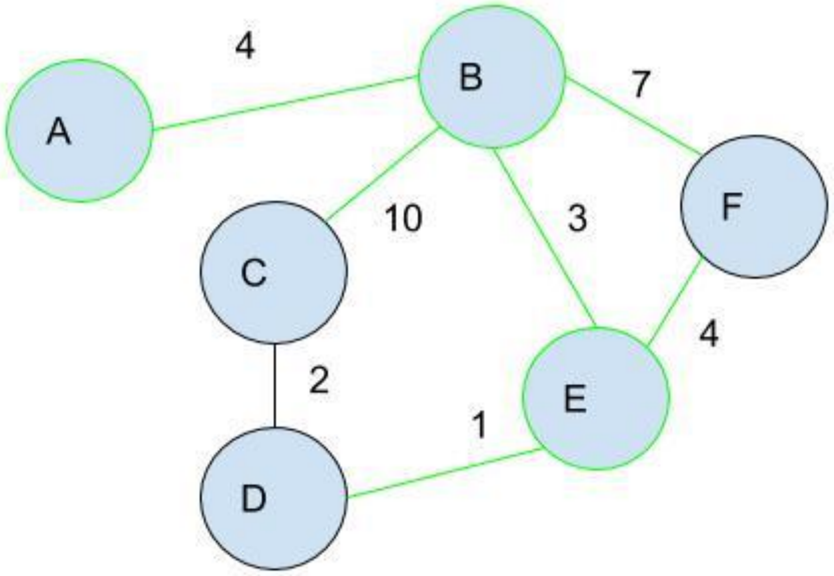
Graph Structure Example



- argmin of dist (not previously visited) is B
- B is visited
  - marked
  - prev is updated
- B's neighbors are updated in dist

Hash on Name	Name	Marked	Dist	Prev	adj
0	A	1	0	NULL	LL → B(4)
1	B	1	4	A	LL → A(4) → C(10) → E(3) → F(7)
2	C	0	14	NULL	LL → B(10) → D(2)
3	D	0	inf	NULL	LL → C(2) → E(1)
4	E	0	7	NULL	LL → B(3) → D(1) → F(4)
5	F	0	11	NULL	LL → B(7) → E(4)

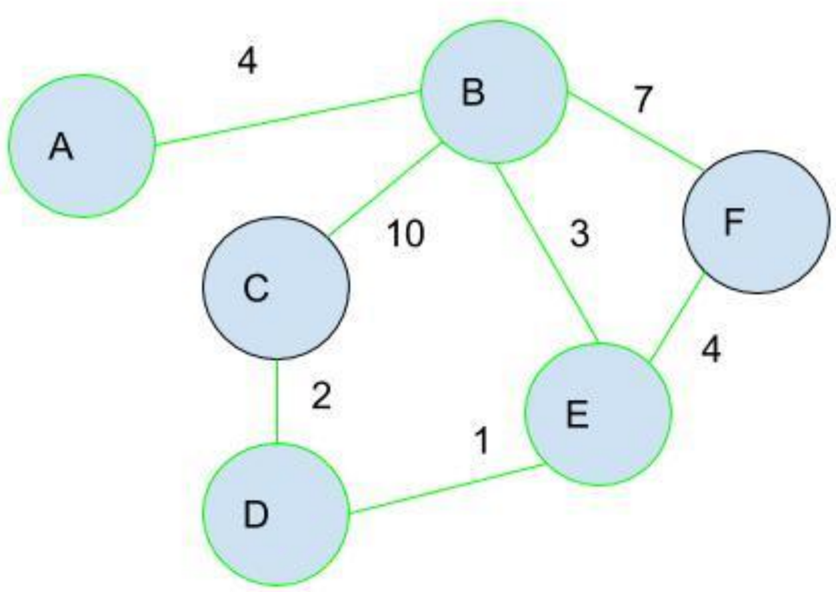
Graph Structure Example



- argmin of dist (not previously visited) is E
- E is visited
  - marked
  - prev is updated
- E's neighbors are updated in dist

Hash on Name	Name	Marked	Dist	Prev	adj
0	A	1	0	NULL	LL → B(4)
1	B	1	4	A	LL → A(4) → C(10) → E(3) → F(7)
2	C	0	14	NULL	LL → B(10) → D(2)
3	D	0	8	NULL	LL → C(2) → E(1)
4	E	1	7	B	LL → B(3) → D(1) → F(4)
5	F	0	11	NULL	LL → B(7) → E(4)

Graph Structure Example



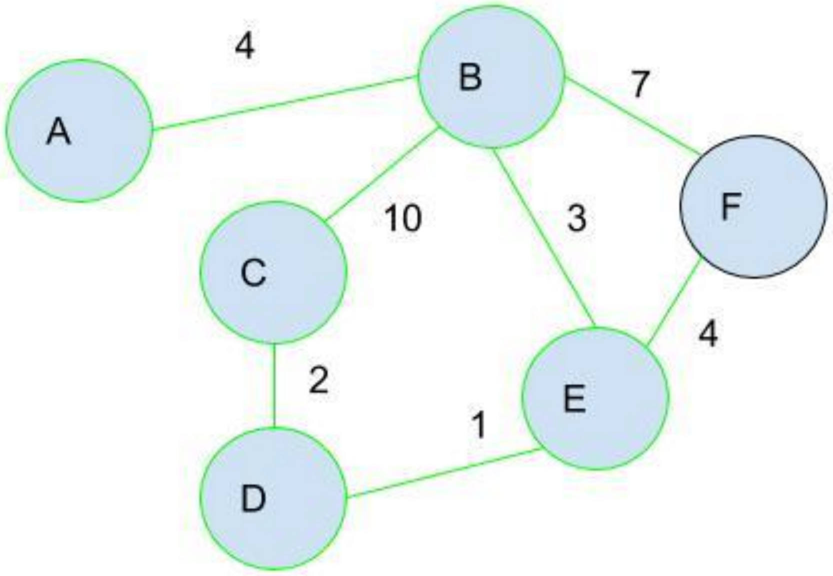
- argmin of dist (not previously visited) is D

- D is visited
  - marked
  - prev is updated

- D's neighbors are updated in dist

Hash on Name	Name	Marked	Dist	Prev	adj
0	A	1	0	NULL	LL → B(4)
1	B	1	4	A	LL → A(4) → C(10) → E(3) → F(7)
2	C	0	10	NULL	LL → B(10) → D(2)
3	D	1	8	E	LL → C(2) → E(1)
4	E	1	7	B	LL → B(3) → D(1) → F(4)
5	F	0	11	NULL	LL → B(7) → E(4)

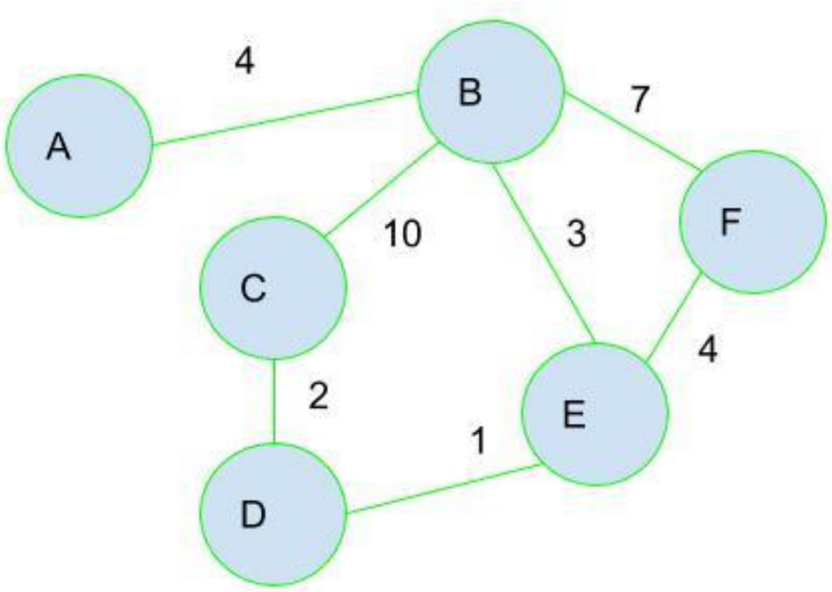
Graph Structure Example



- argmin of dist (not previously visited) is C
- C is visited
  - marked
  - prev is updated
- C's neighbors are updated in dist

Hash on Name	Name	Marked	Dist	Prev	adj
0	A	1	0	NULL	LL → B(4)
1	B	1	4	A	LL → A(4) → C(10) → E(3) → F(7)
2	C	1	10	D	LL → B(10) → D(2)
3	D	1	8	E	LL → C(2) → E(1)
4	E	1	7	B	LL → B(3) → D(1) → F(4)
5	F	0	11	NULL	LL → B(7) → E(4)

Graph Structure Example

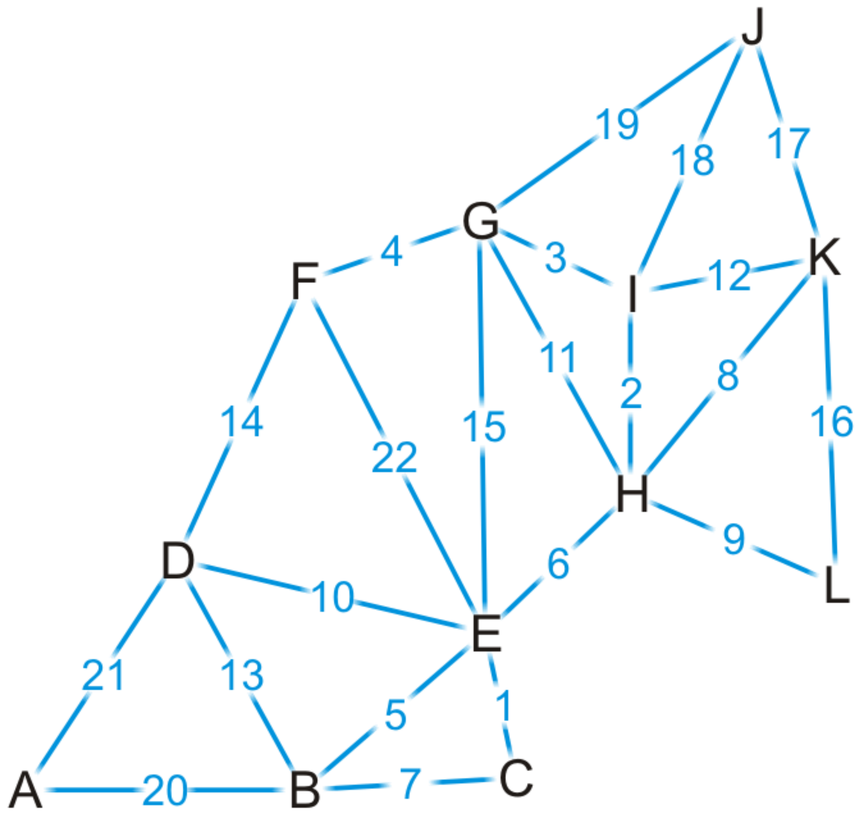


- argmin of dist (not previously visited) is B
- B is visited
  - marked
  - prev is updated
- B's neighbors are updated in dist
- All items in dist are marked .. DONE!

Hash on Name	Name	Marked	Dist	Prev	adj
0	A	1	0	NULL	LL → B(4)
1	B	1	4	A	LL → A(4) → C(10) → E(3) → F(7)
2	C	1	10	D	LL → B(10) → D(2)
3	D	1	8	E	LL → C(2) → E(1)
4	E	1	7	B	LL → B(3) → D(1) → F(4)
5	F	1	11	(E or B)	LL → B(7) → E(4)

Another Example

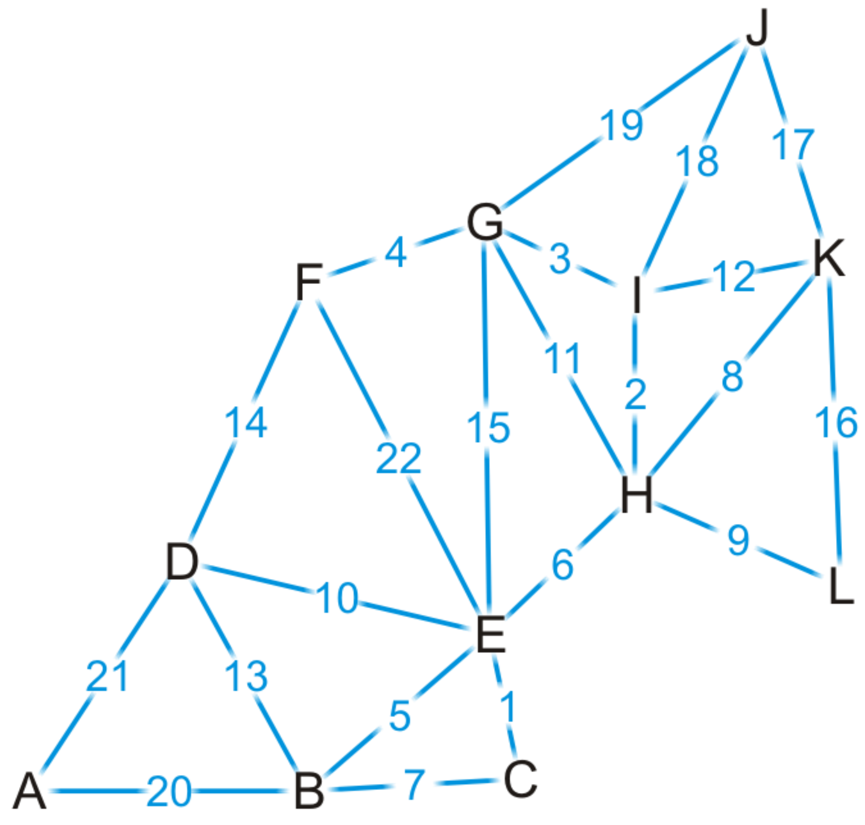
Find the shortest distance from (K) to every other node



Example

We set up our table

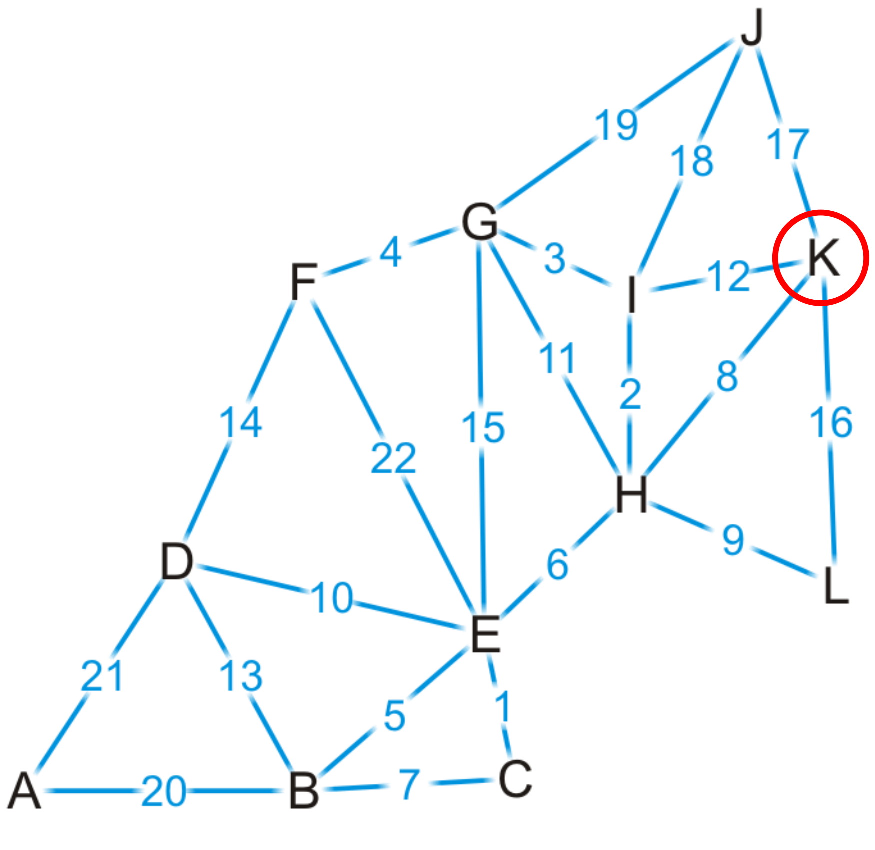
– Which unvisited vertex has the minimum distance to it?



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

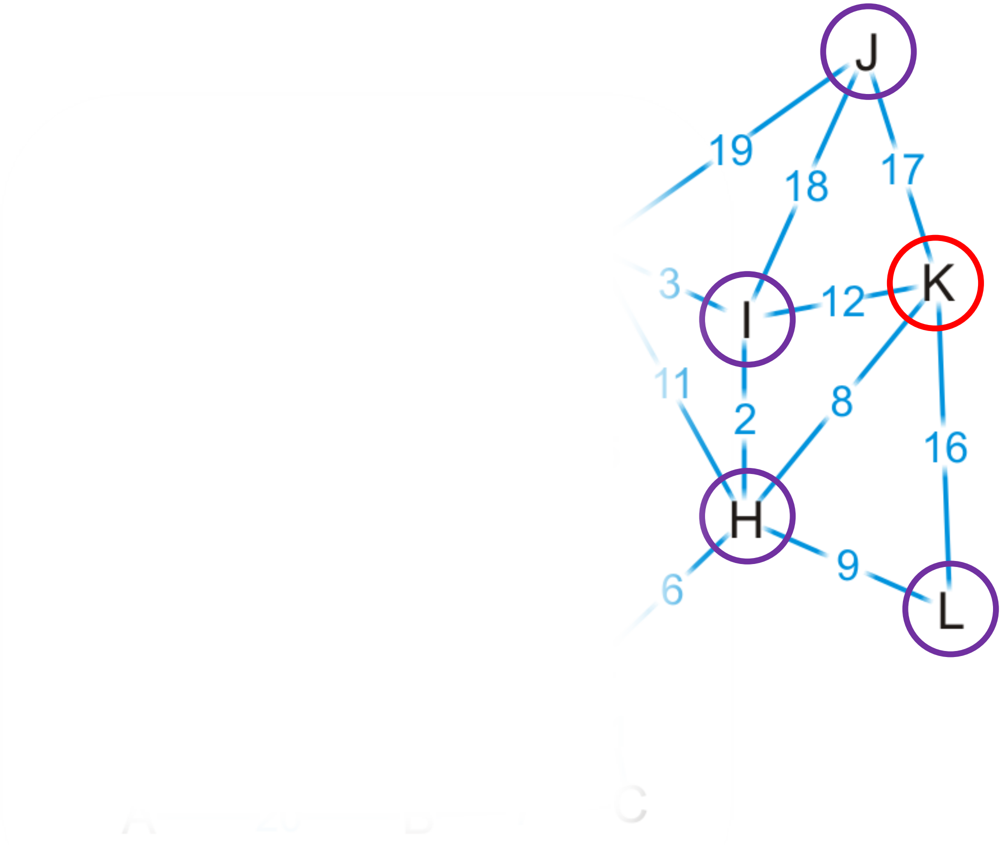
We visit vertex K



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

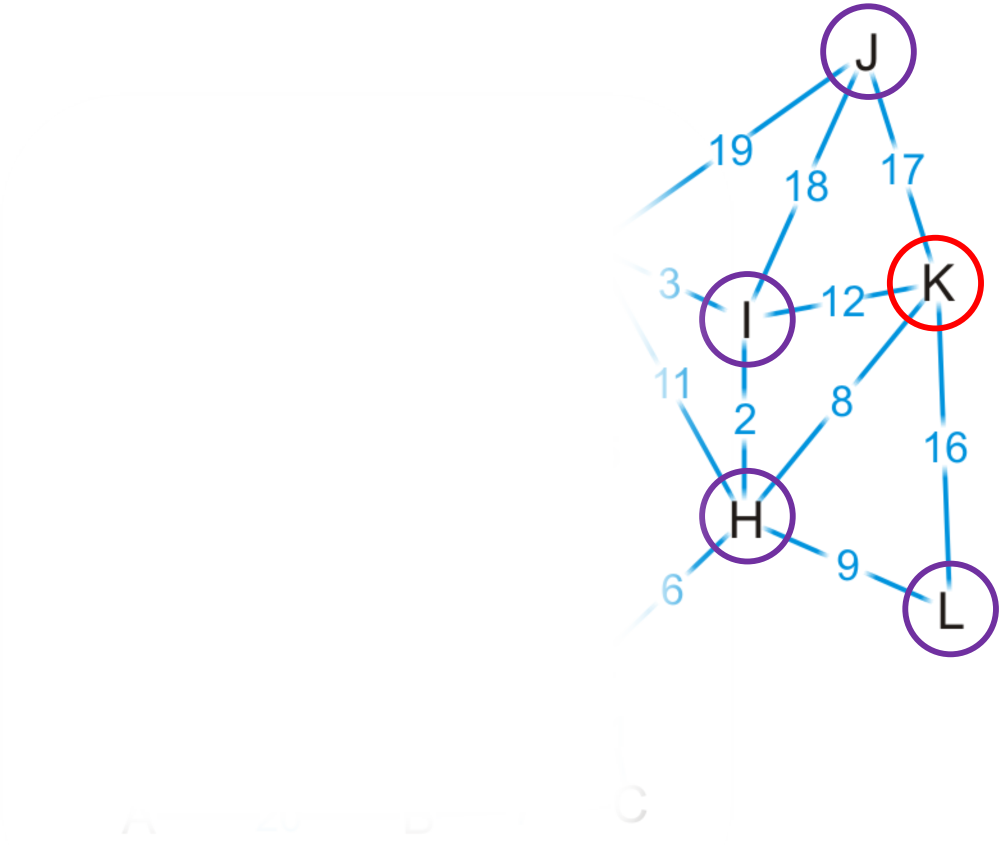


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	∞	Ø
I	F	∞	Ø
J	F	∞	Ø
K	**T**	**0**	**Ø**
L	F	∞	Ø

GEORGETOWN UNIVERSITY

Example

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

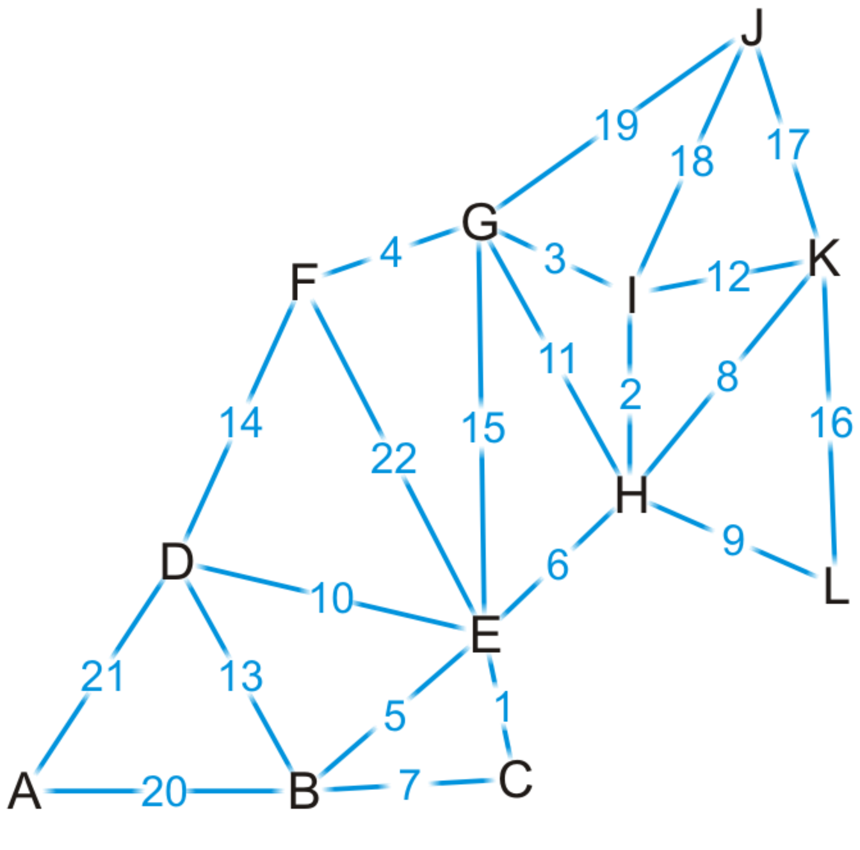


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	**8**	**K**
I	F	**12**	**K**
J	F	**17**	**K**
K	**T**	**0**	**Ø**
L	F	**16**	**K**

Example

We’re finished with vertex K

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



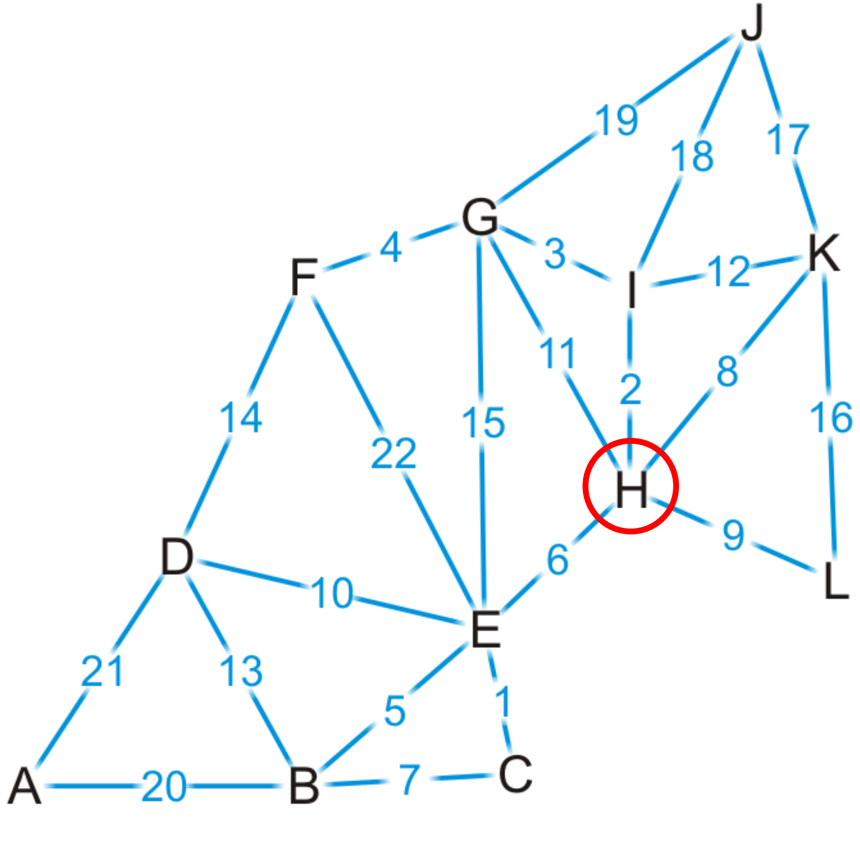
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	∞	Ø
F	F	∞	Ø
G	F	∞	Ø
H	F	8	K
I	F	12	K
J	F	17	K
K	T	0	Ø
L	F	16	K

GEORGETOWN UNIVERSITY

Example

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

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



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

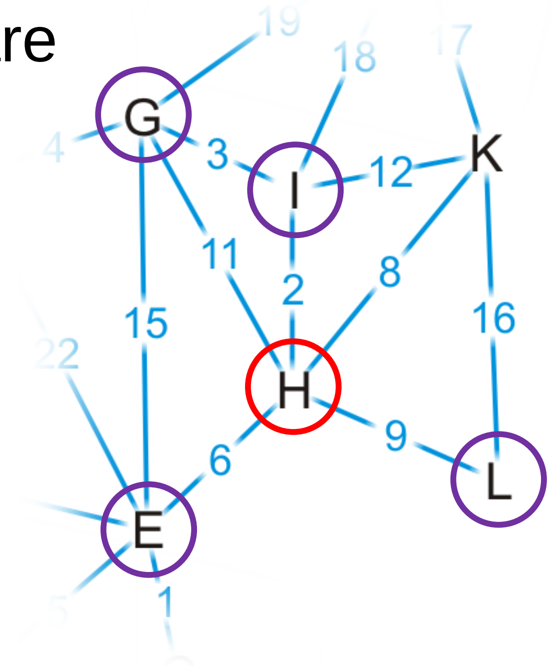
Example

Consider these paths:

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

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

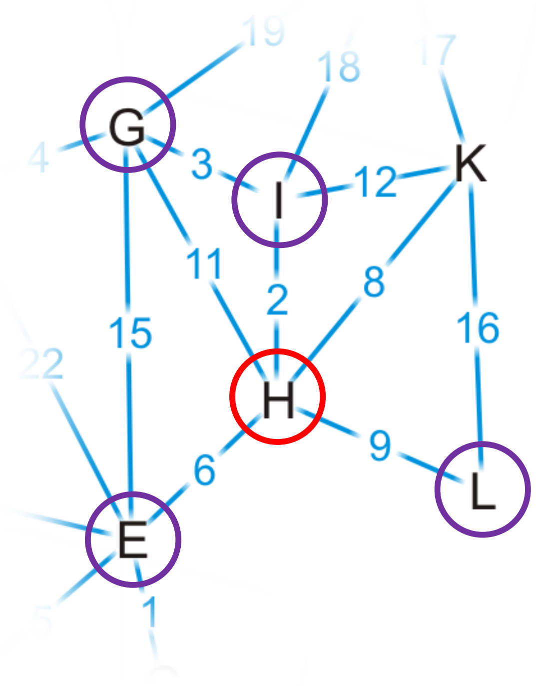
– Which of these are shorter than any known path?



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

Example

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

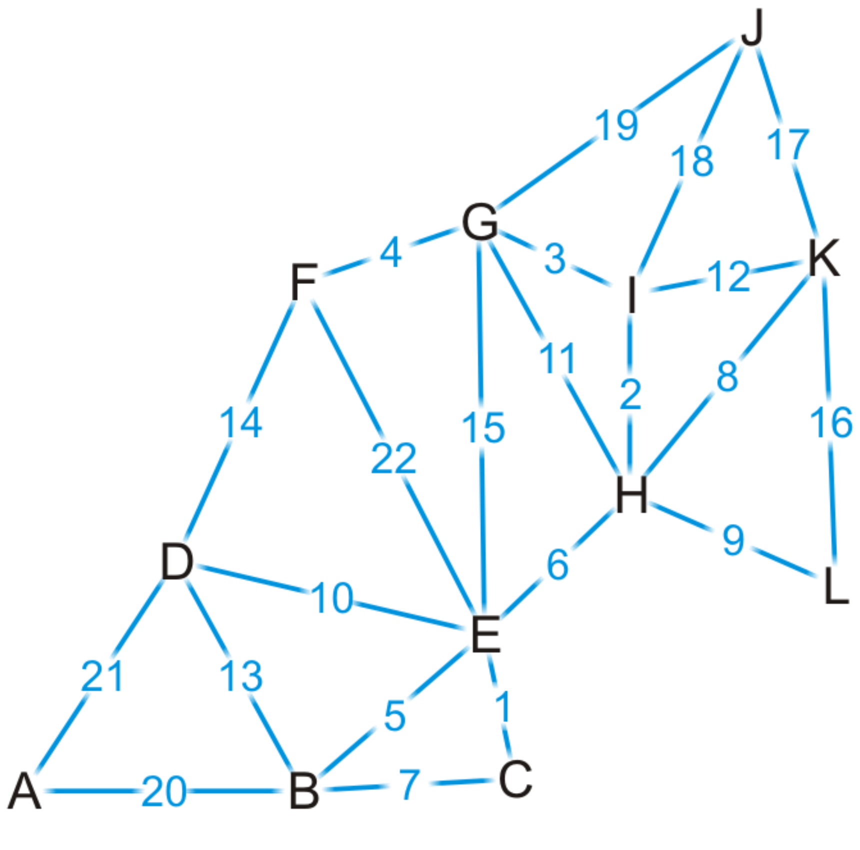


Vertex	Visited	Distance	Previous
A	F	∞	∅
B	F	∞	∅
C	F	∞	∅
D	F	∞	∅
E	F	**14**	**H**
F	F	∞	∅
G	F	**19**	**H**
H	**T**	**8**	**K**
I	F	**10**	**H**
J	F	17	K
K	T	0	∅
L	F	16	K

Example

We are finished with vertex H

– Which vertex do we visit next?



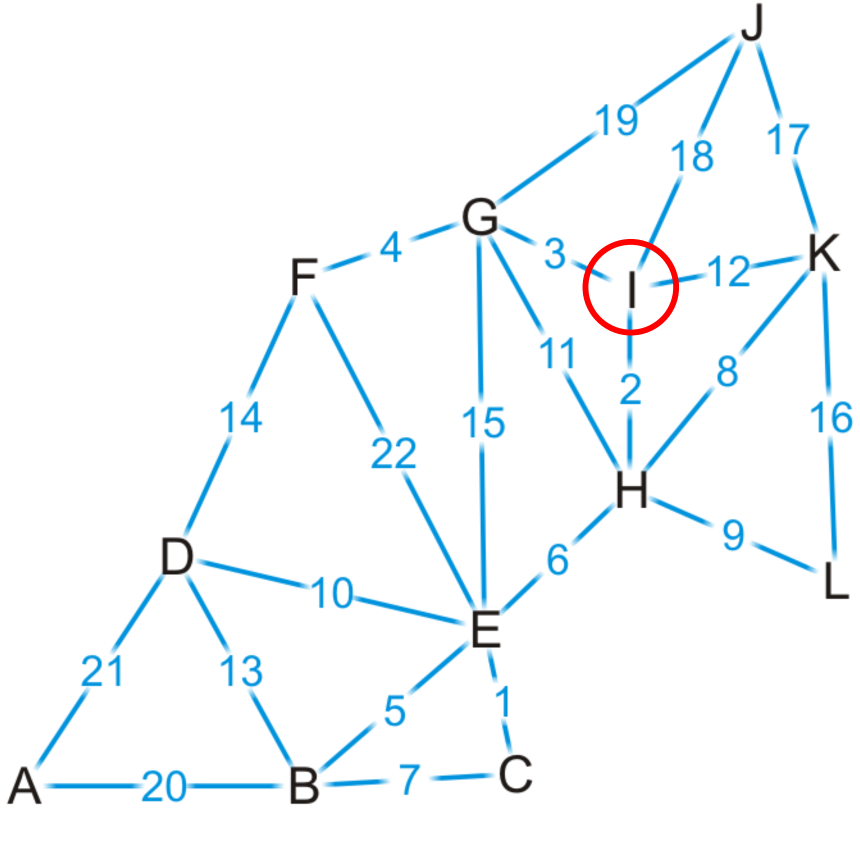
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	F	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

GEORGETOWN UNIVERSITY

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

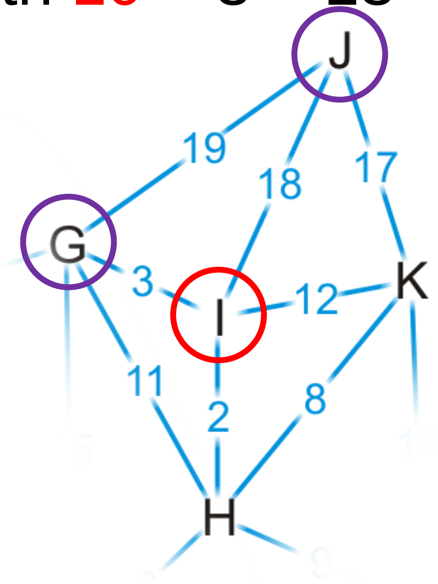


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	19	H
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

Consider these paths:

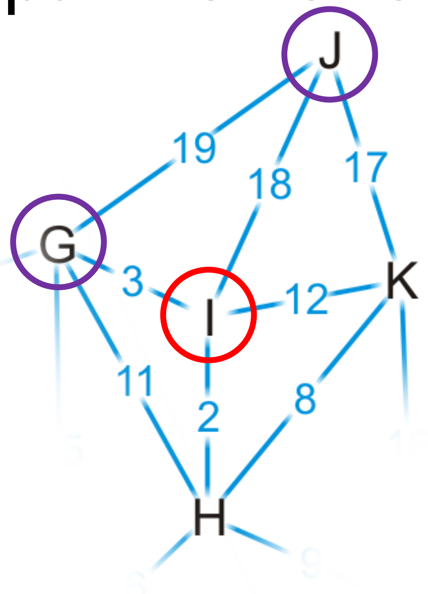
(K, H, I, G) of length 10 + 3 = 13   (K, H, I, J) of length 10 + 18 = 28



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

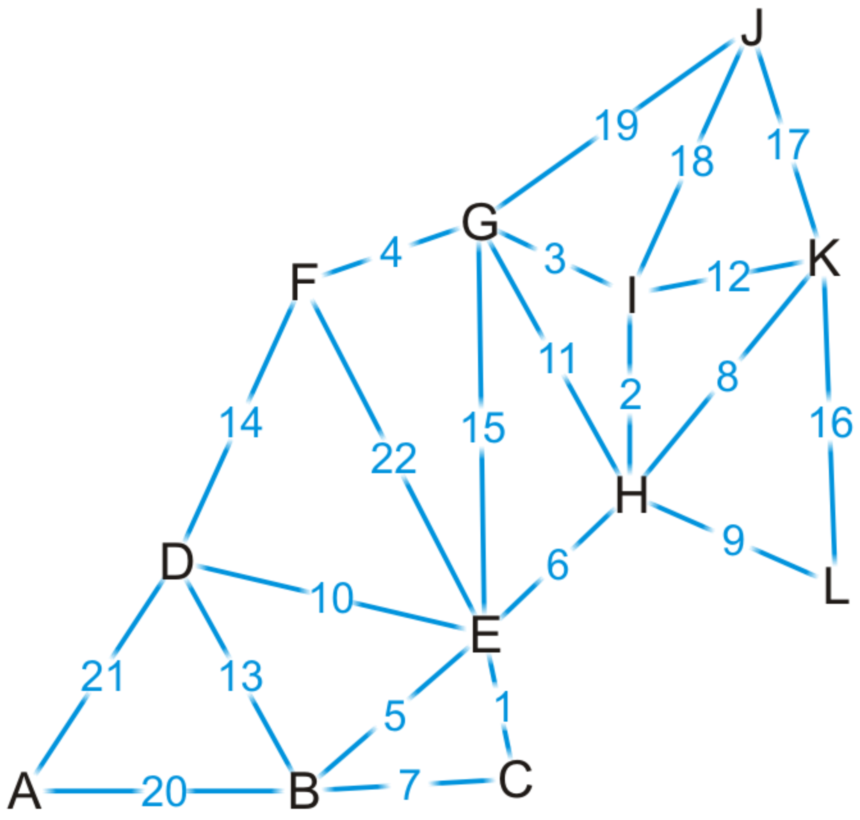


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	**13**	**I**
H	T	8	K
I	**T**	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K

GEORGETOWN UNIVERSITY

Example

Which vertex can we visit next?

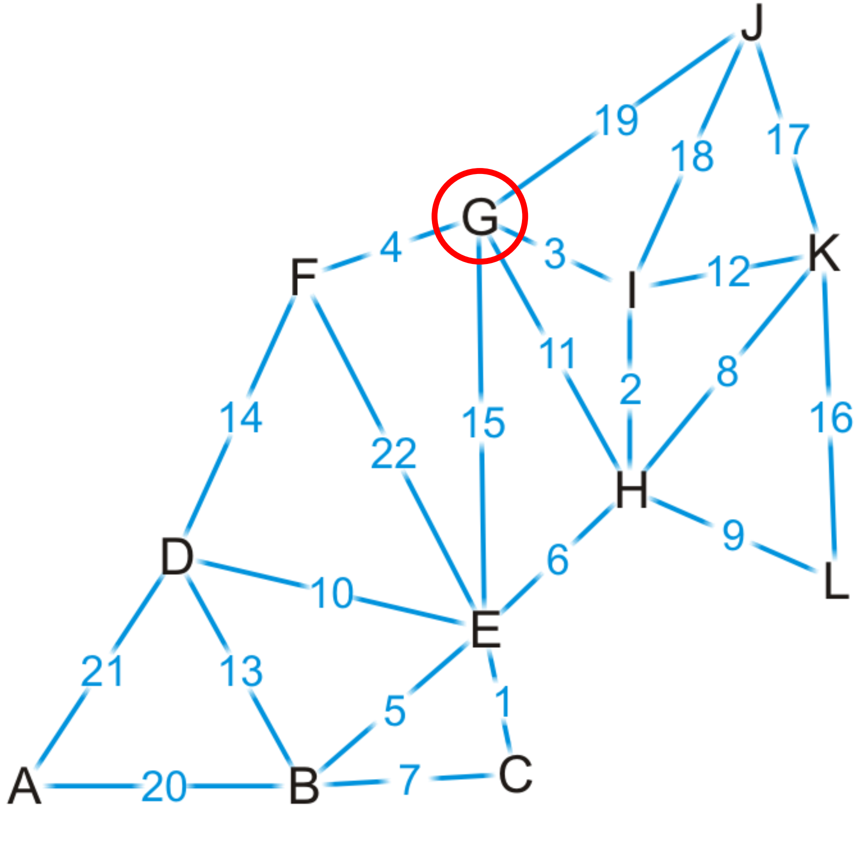


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

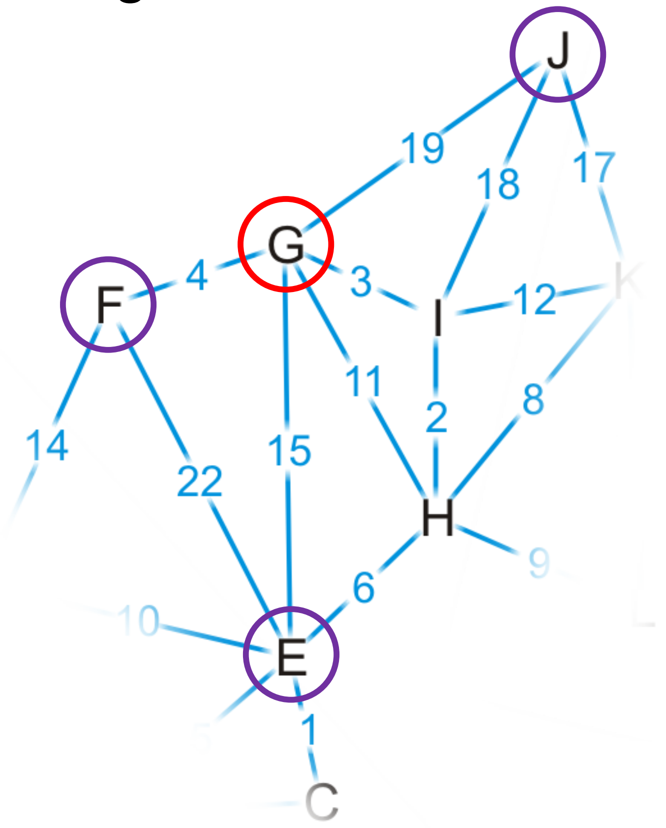
Example

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

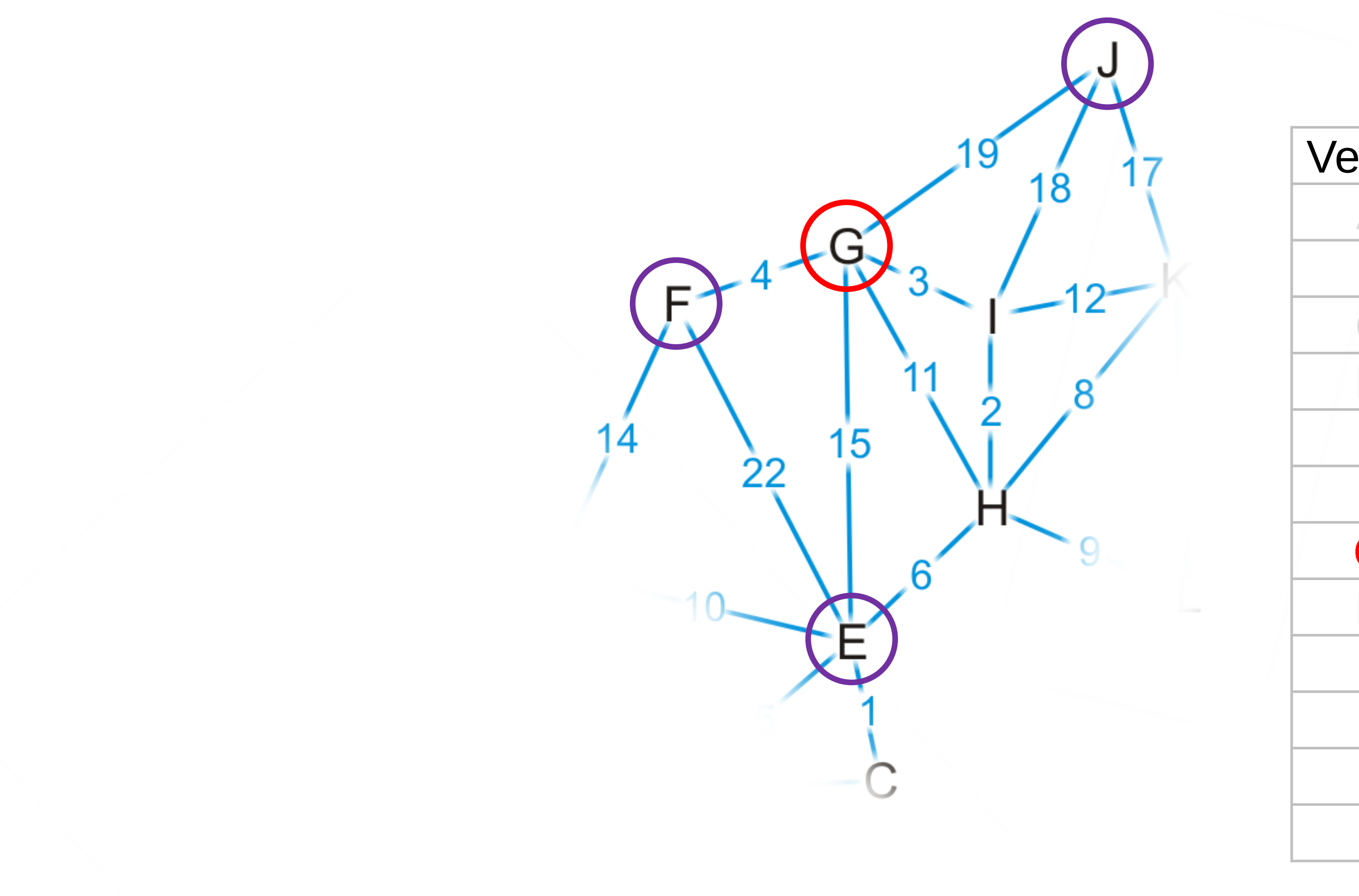
– Which do we update?



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

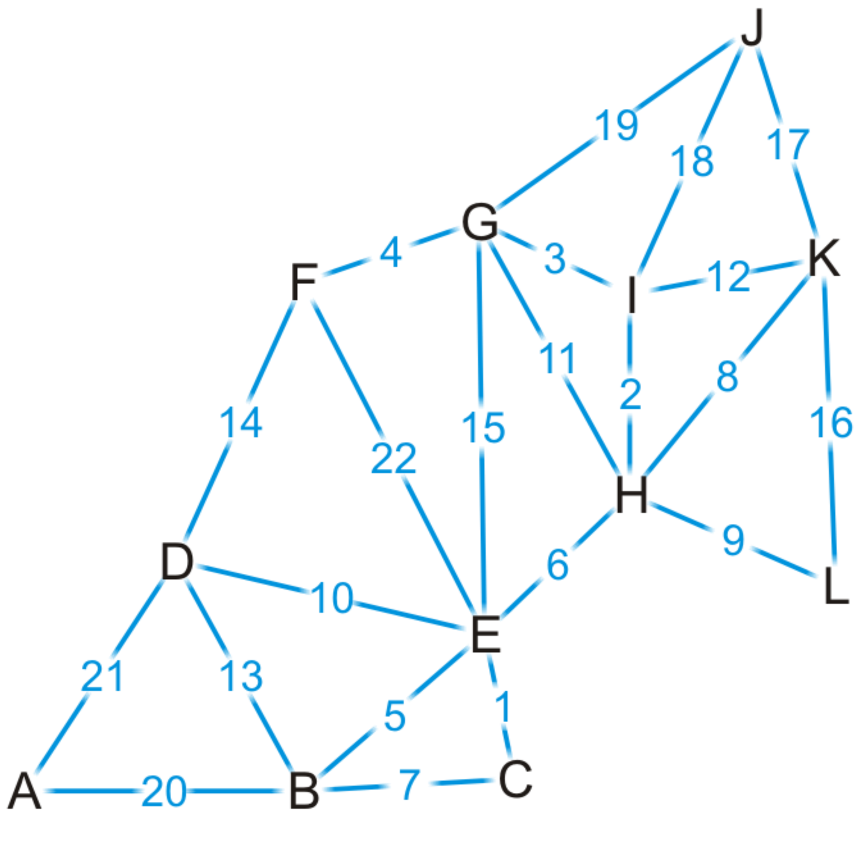
We have now found a path to vertex F



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?



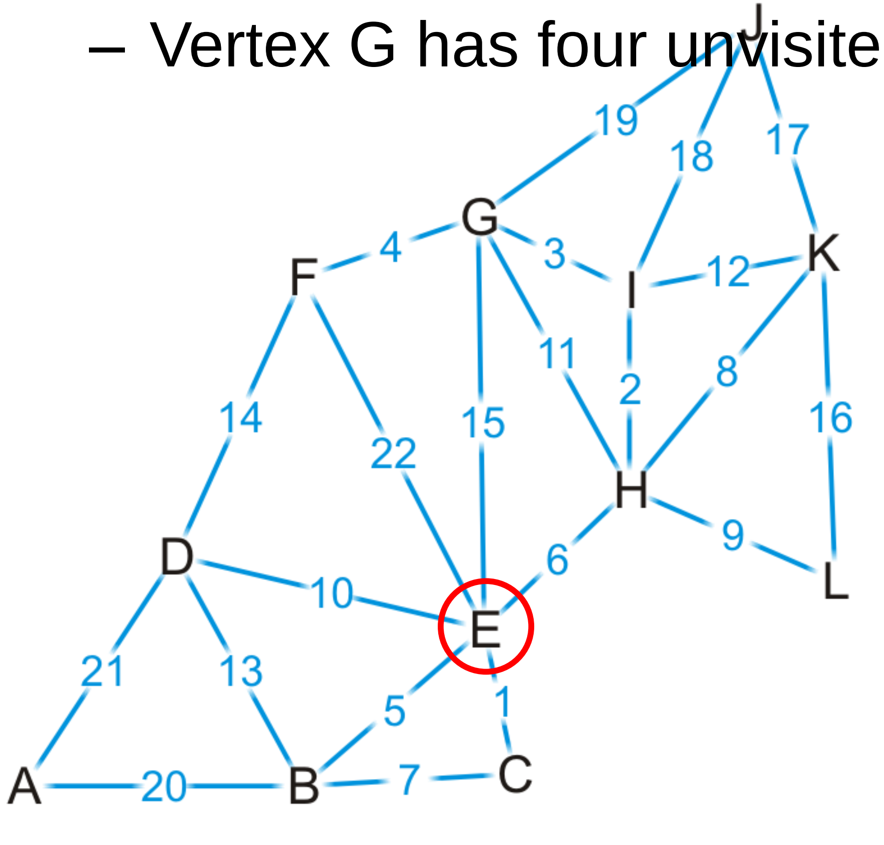
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

GEORGETOWN UNIVERSITY

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

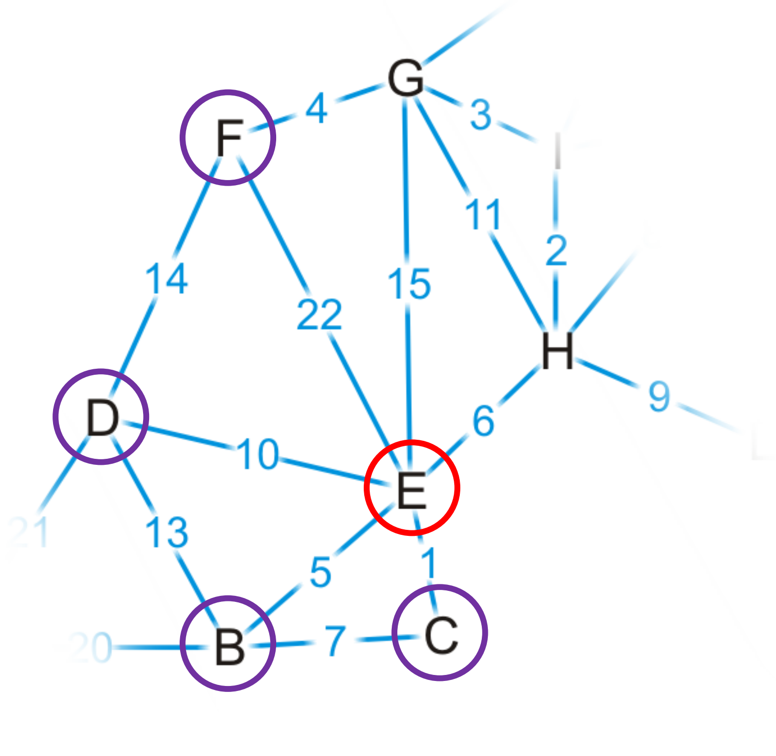


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

GEORGETOWN UNIVERSITY

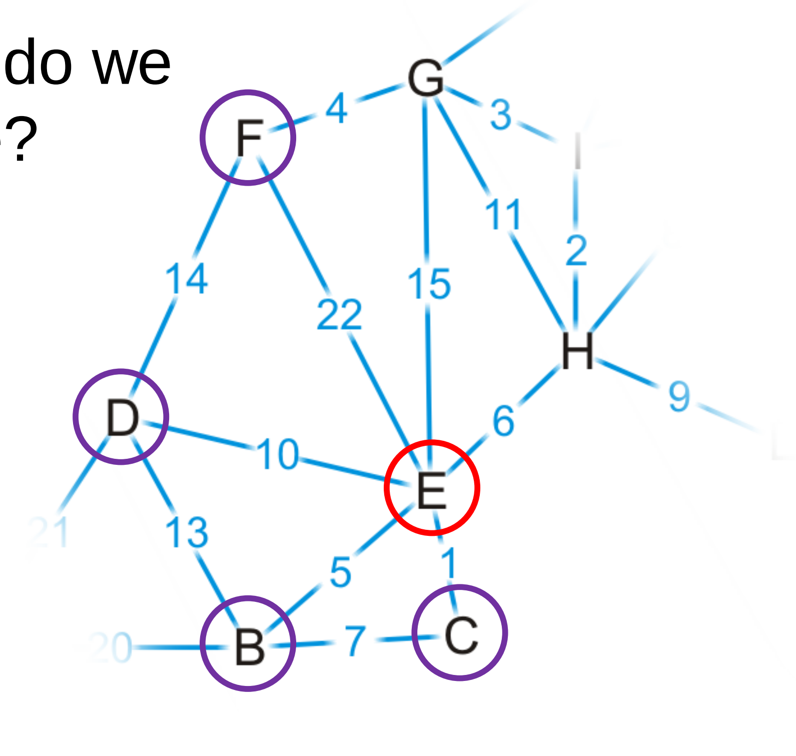
Example

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?

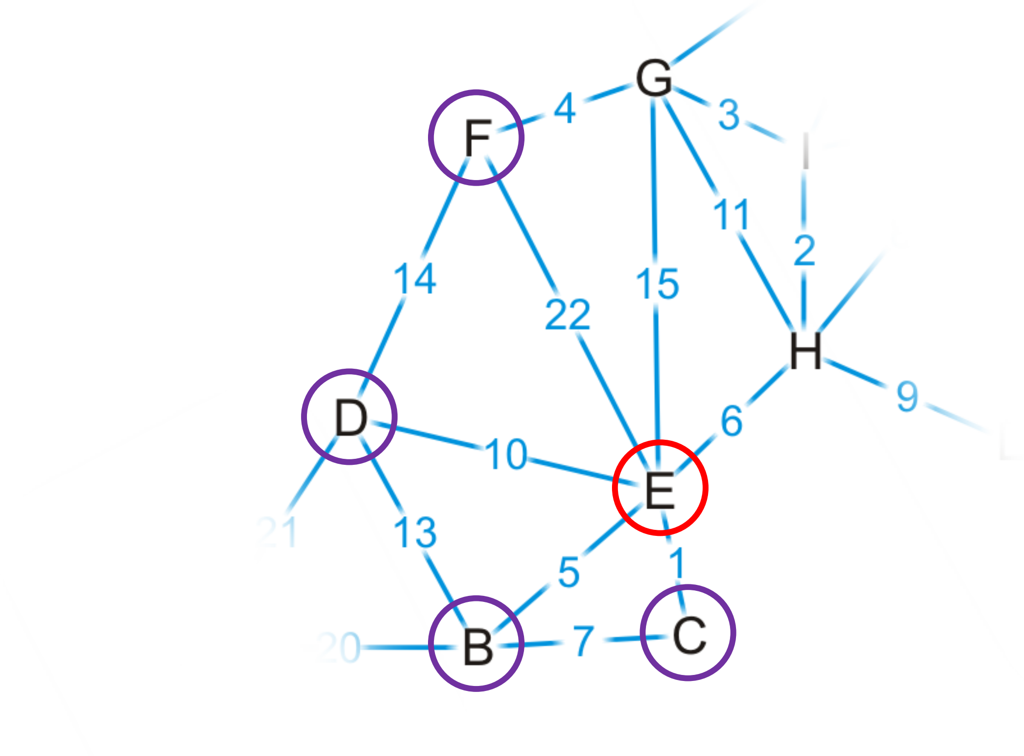


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	**T**	**14**	**H**
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	[illegible]

GEORGETOWN UNIVERSITY

Example

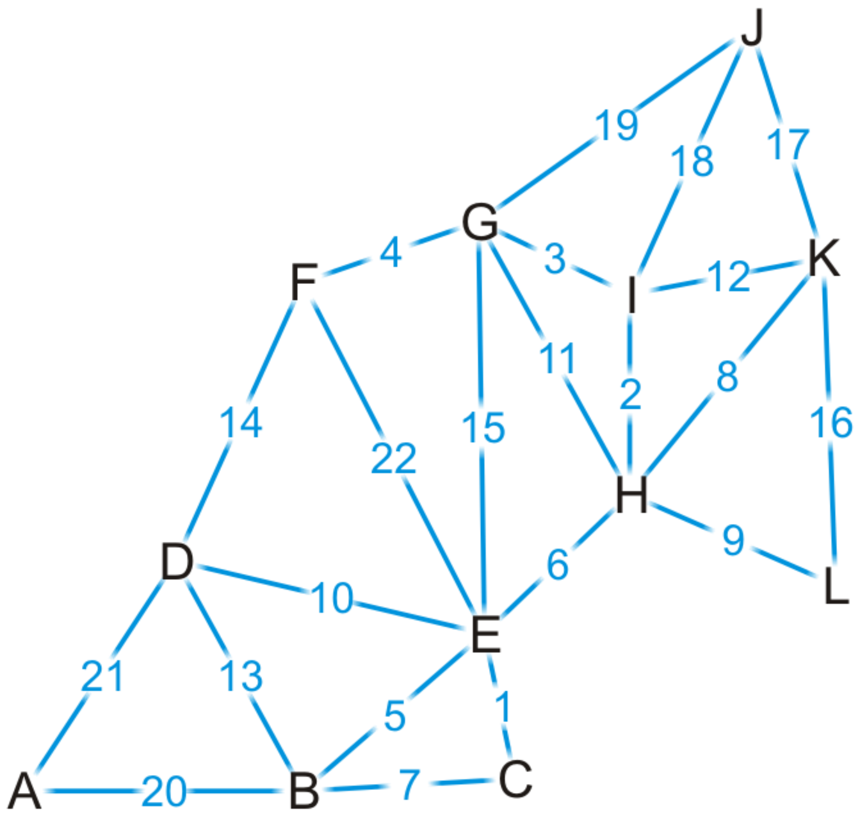
We’ve discovered paths to vertices B, C, D



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?

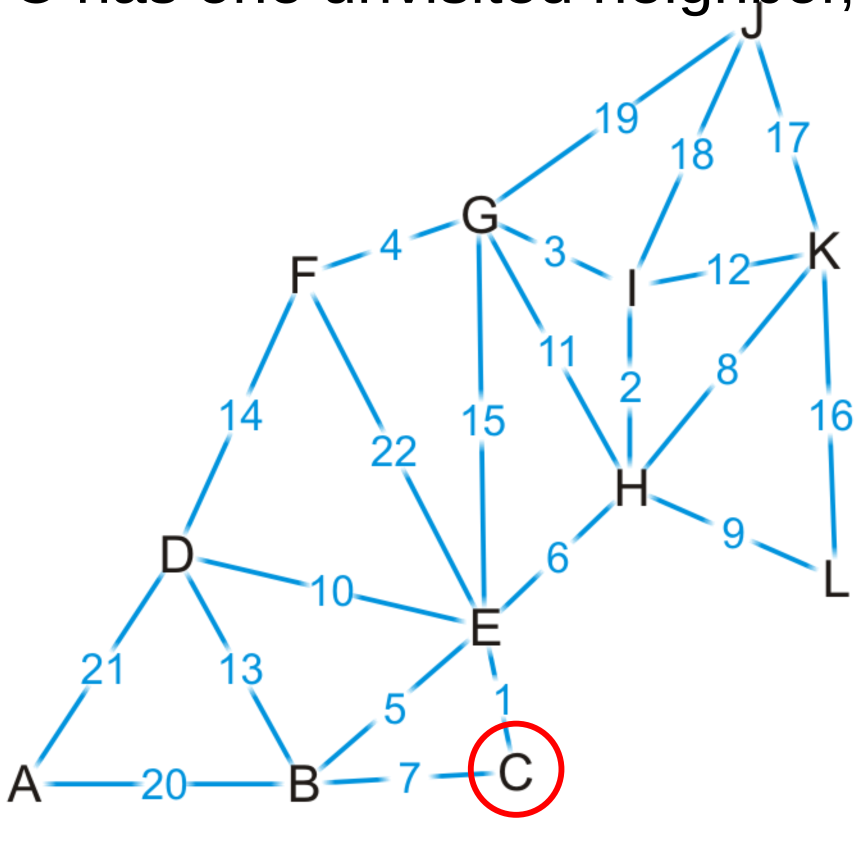


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	F	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

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

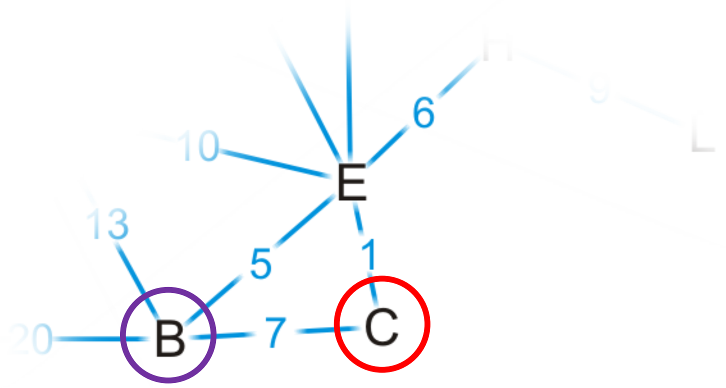


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

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

– We have already discovered a shorter path through vertex E

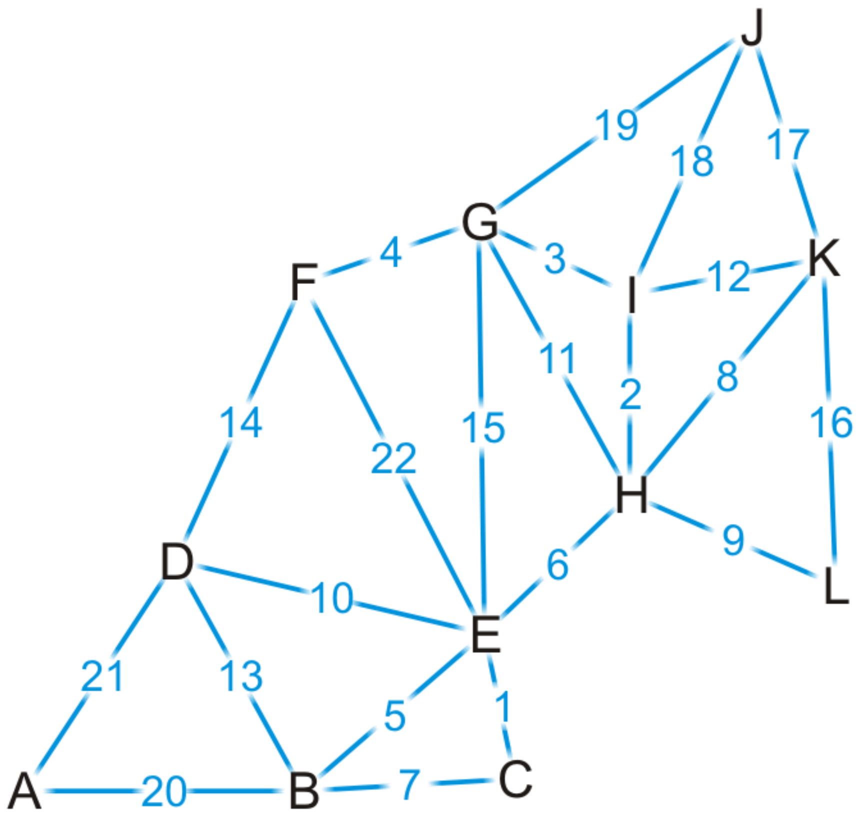


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	**T**	**15**	**E**
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

GEORGETOWN UNIVERSITY

Example

Where to next?

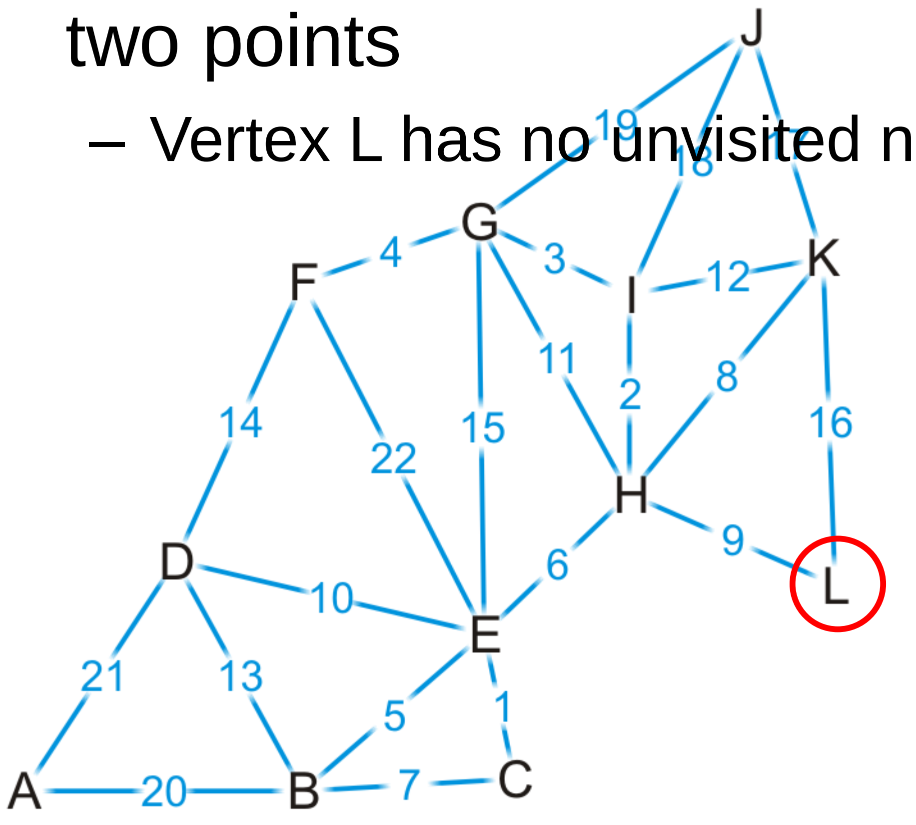


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	F	16	K

Example

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

– Vertex L has no unvisited neighbors

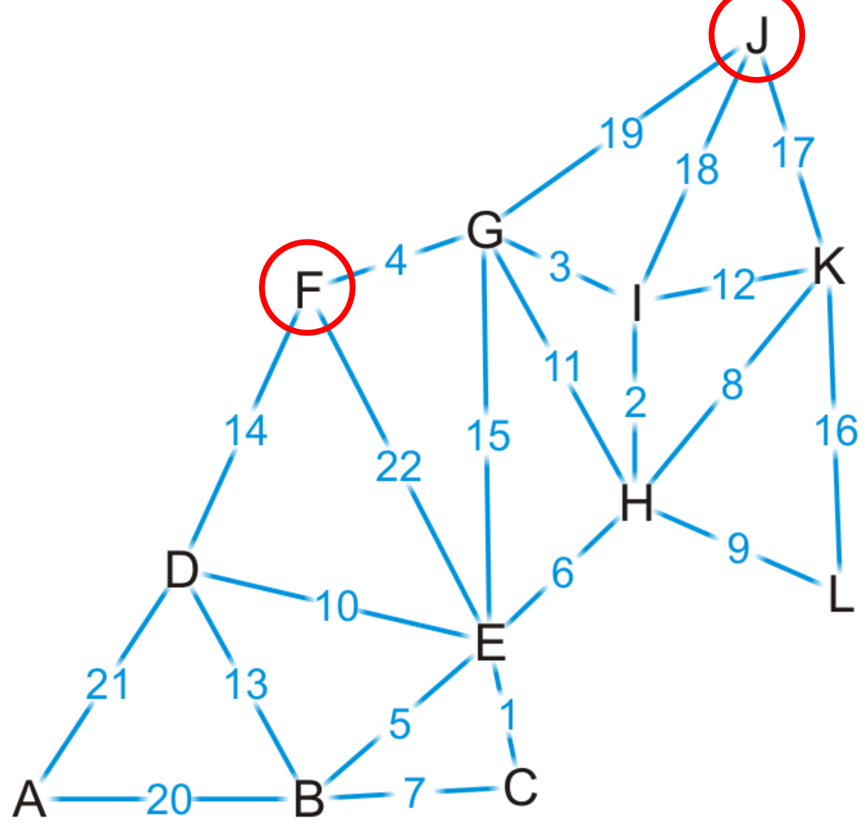


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	F	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	**T**	**16**	**K**

Example

Where to next?

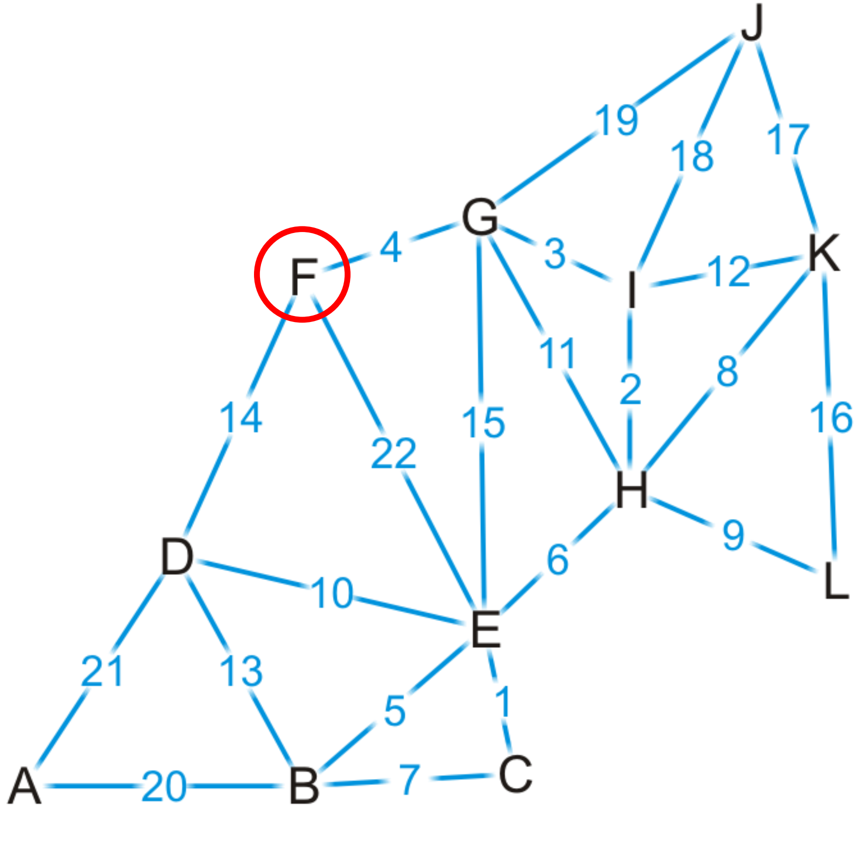
– Does it matter if we visit vertex F first or vertex J first?



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

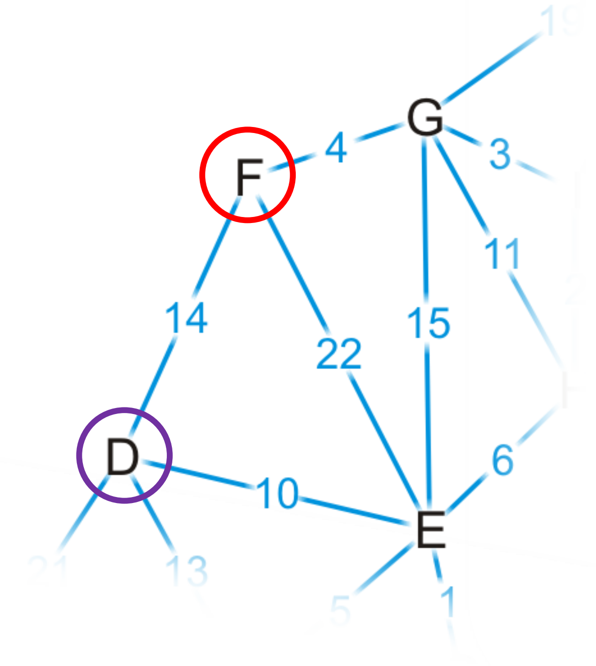


Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

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



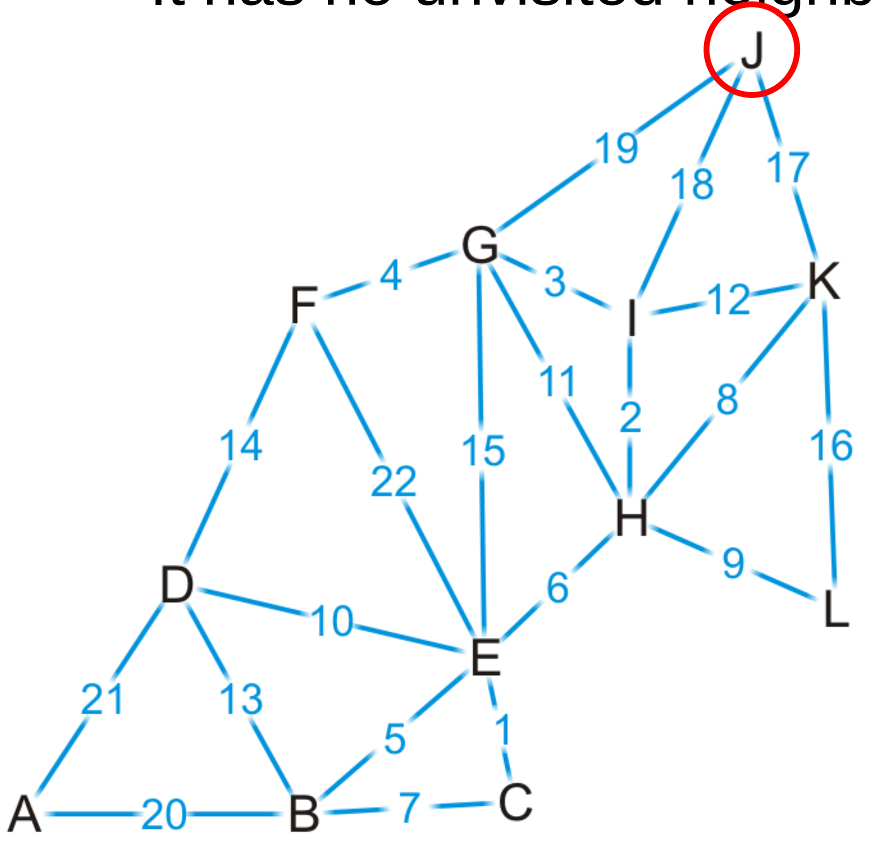
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	19	E
C	T	15	E
D	F	24	E
E	T	14	H
F	**T**	**17**	**G**
G	T	13	I
H	T	8	K
I	T	10	H
J	F	17	K
K	T	0	Ø
L	T	16	K

GEORGETOWN UNIVERSITY

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

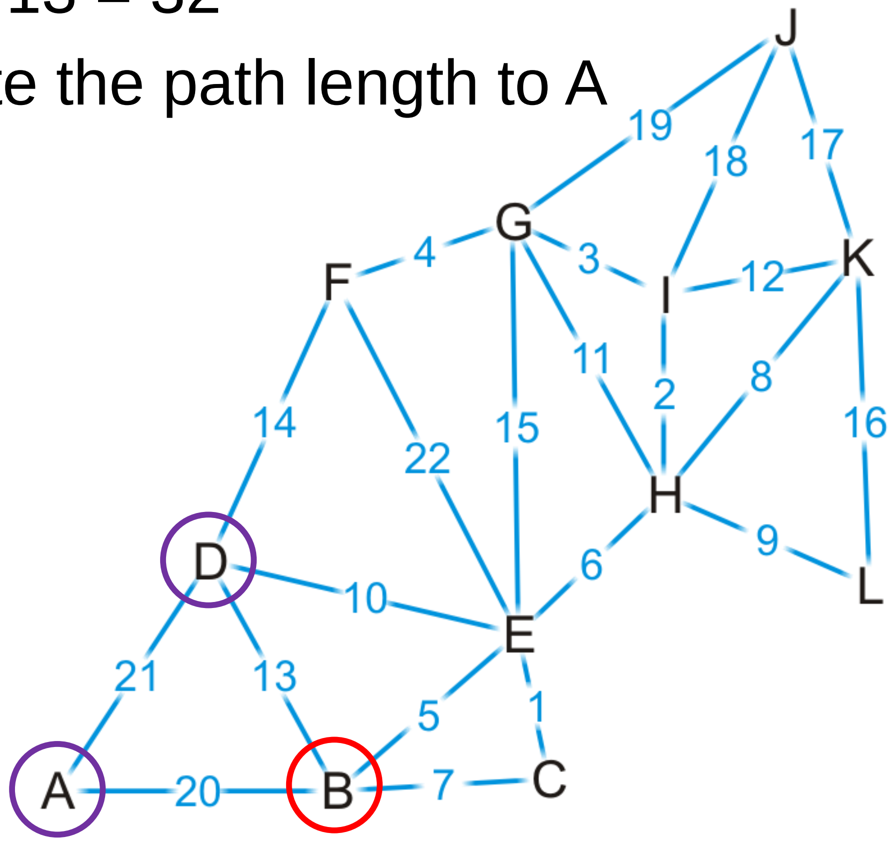
GEORGETOWN UNIVERSITY

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

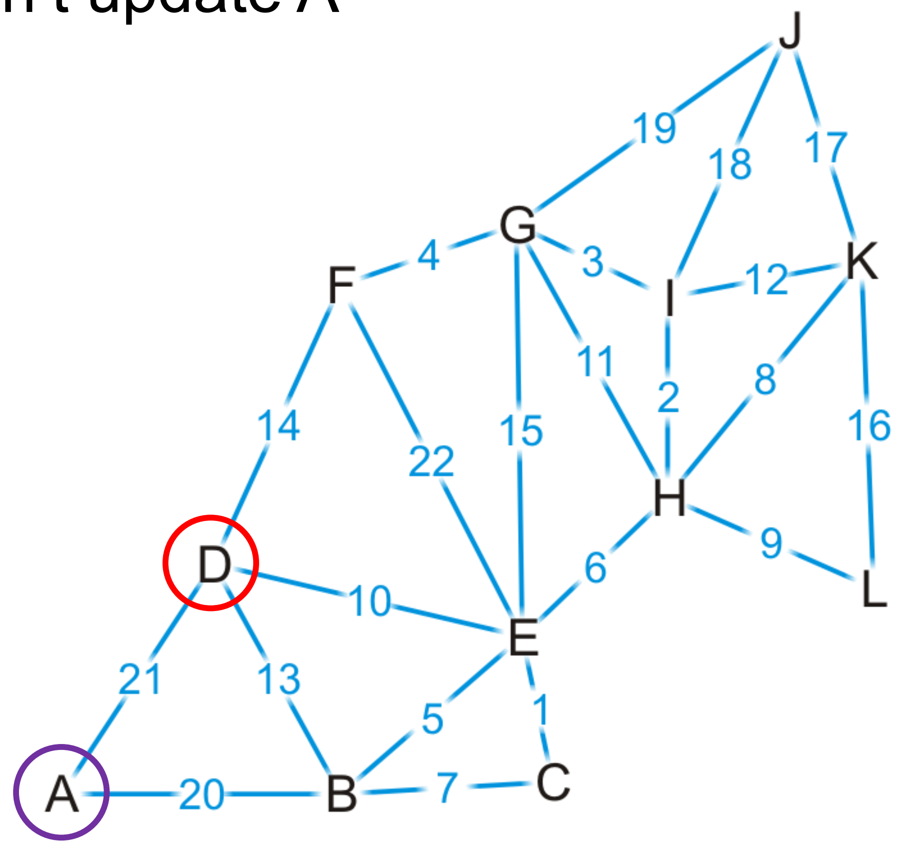


Vertex	Visited	Distance	Previous
A	F	**39**	**B**
B	**T**	**19**	**E**
C	T	15	E
D	F	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

Example

Next we visit vertex D

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



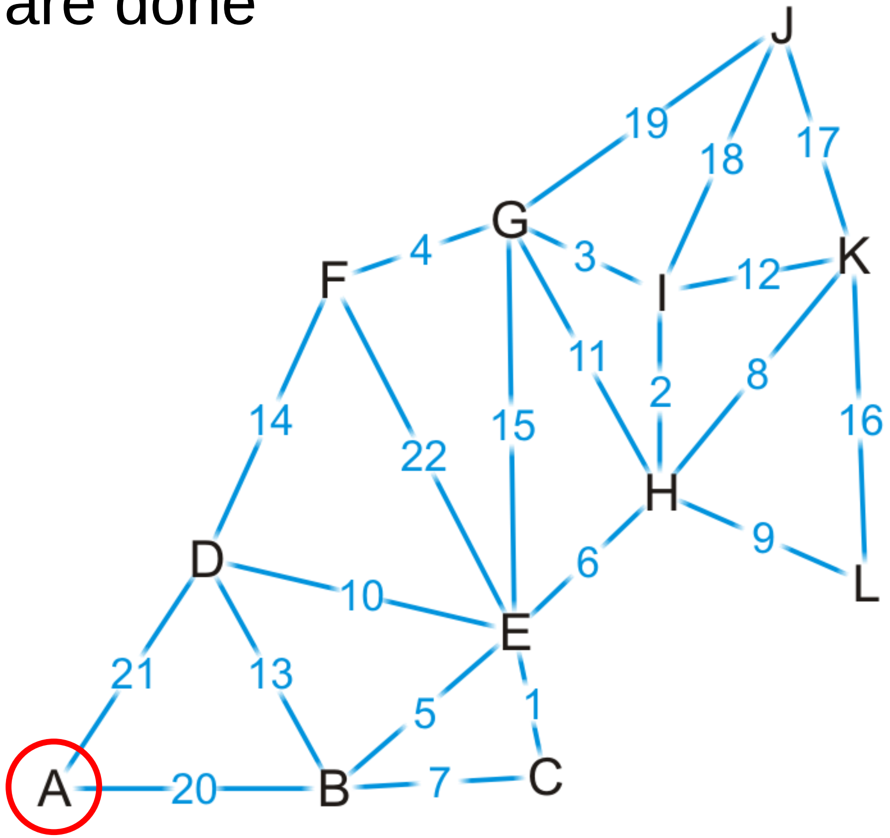
Vertex	Visited	Distance	Previous
A	F	39	B
B	T	19	E
C	T	15	E
D	**T**	**24**	**E**
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

GEORGETOWN UNIVERSITY

Example

Finally, we visit vertex A

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

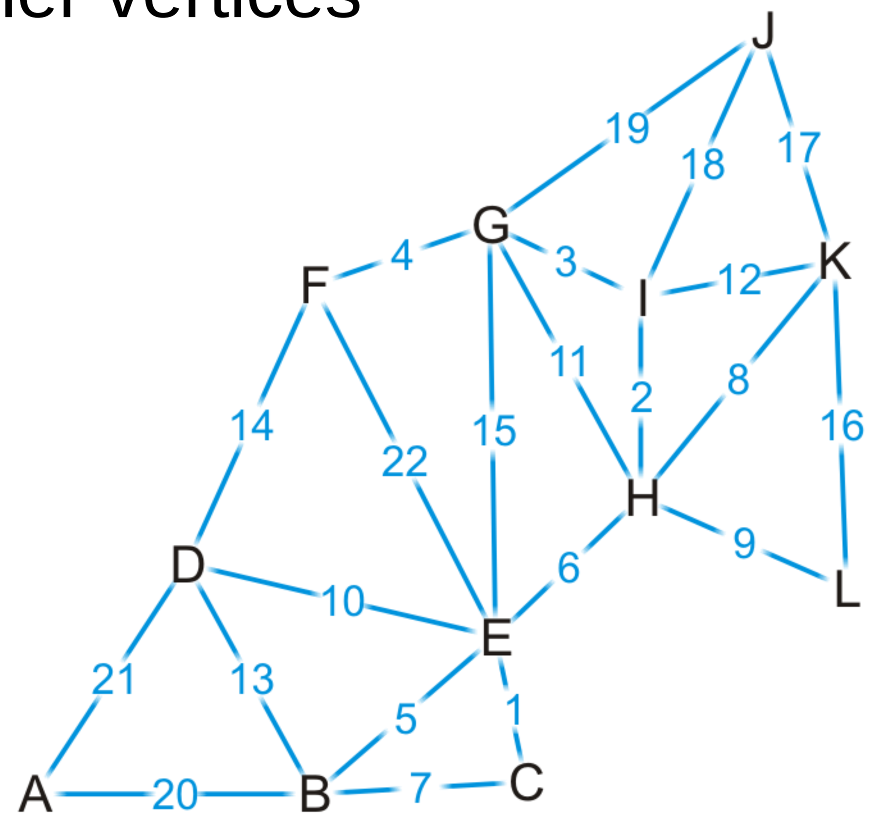


Vertex	Visited	Distance	Previous
A	**T**	**39**	**B**
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

GEORGETOWN UNIVERSITY

Example

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

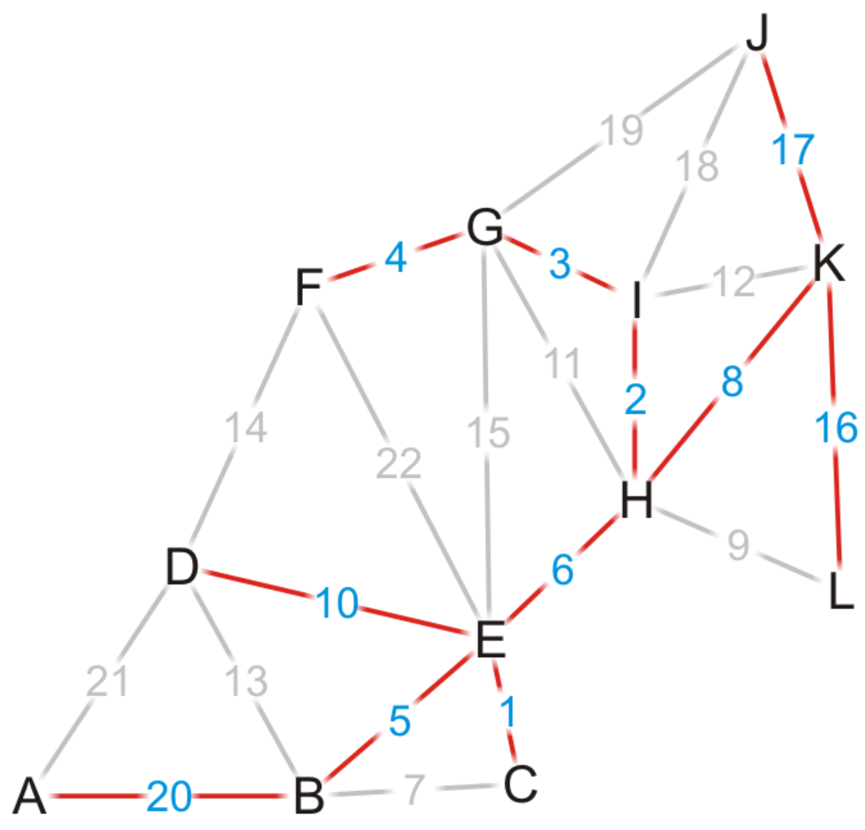


Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

GEORGETOWN UNIVERSITY

Example

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

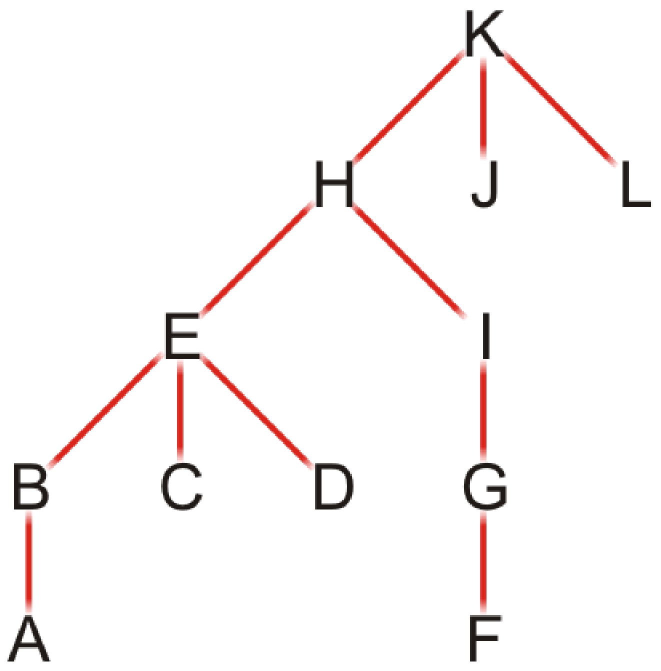


Vertex	Visited	Distance	Previous
A	T	39	B
B	T	19	E
C	T	15	E
D	T	24	E
E	T	14	H
F	T	17	G
G	T	13	I
H	T	8	K
I	T	10	H
J	T	17	K
K	T	0	Ø
L	T	16	K

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

GEORGETOWN UNIVERSITY

Comments on Dijkstra's algorithm

Questions:

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

Implementation and analysis

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

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

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

Can we do better?

- Recall, we only need the closest vertex
- How about a priority queue?
  - Try using a binary min heap

Implementation and analysis

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

- The priority queue will also requires $O(|V|)$ memory
- We must use an adjacency list, not an adjacency matrix

Pick Min From Heap: We iterate $|V|$ times, each time finding the *closest* vertex to the source

- Initialize: Place the distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Each pick is constant, but then must swapDown $O(\ln(|V|))$
- Each p, the work required for this is $O(|V| \ln(|V|))$

Repeatedly Update Heap:

- Recall that each edge visited may result in a new edge being placed to the very top of the heap.
- Thus, the work required for this is $O(|E| \ln(|V|))$
- NOTE: there must be a way to "eliminate" repeated nodes in the heap to keep the size restricted to V. But How? Can we do this in constant time?

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

Thus a priority heap may be preferred when the graph is sparse.

GEORGETOWN UNIVERSITY

Applications of Graphs

- Modeling/Analyzing Networks
  - Social networks
  - Computer networks
  - Markov Processes
  - Connected components / Image analysis
- Flows
  - Matching Problem
  - Critical Paths
  - Security: points of weakness

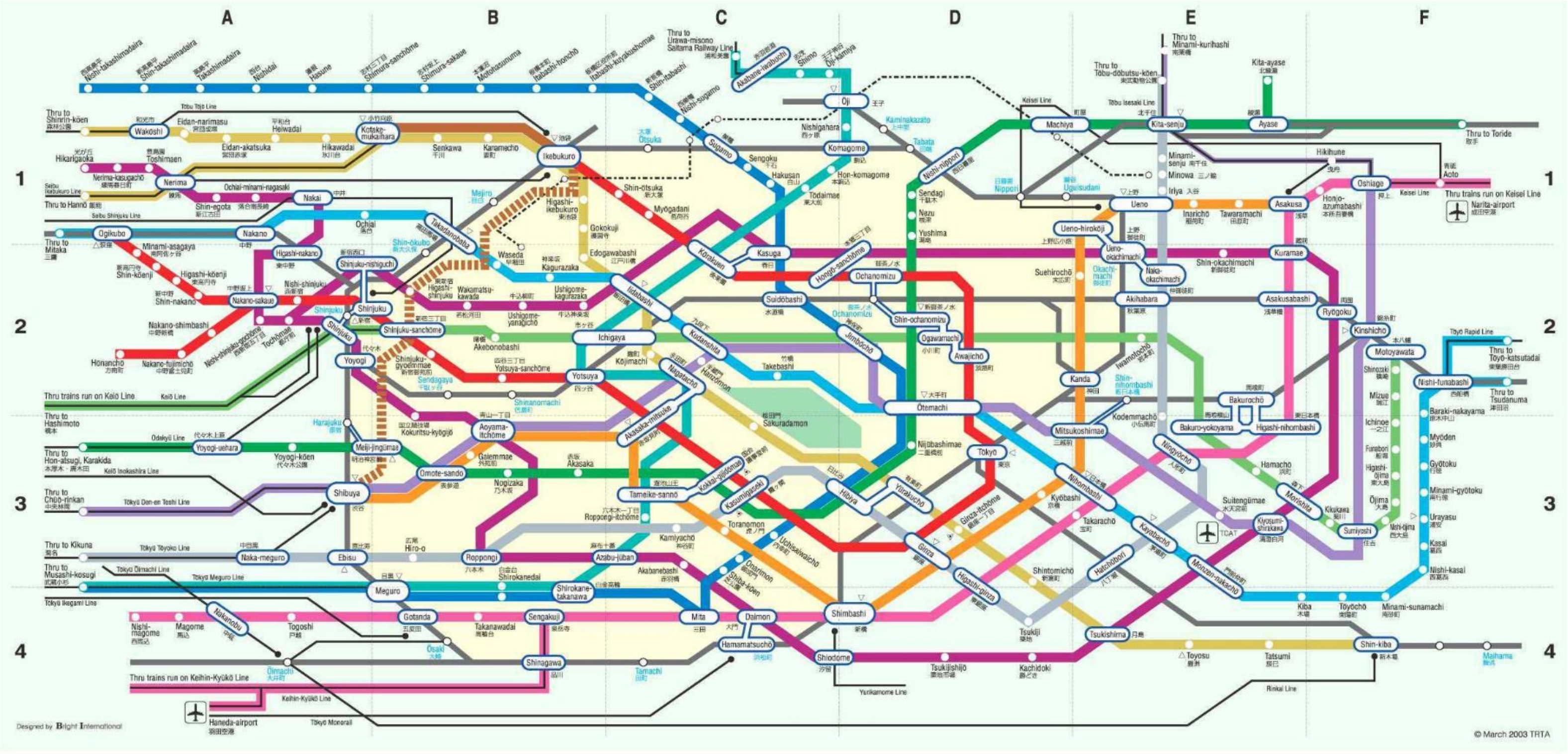
Common Paths on a Graph

- Shortest Paths
  - Path connecting two nodes with minimum aggregate weight
- Hamilton Paths:
  - Path that visits each node once
- Euler Paths
  - Path that traverses each edge once

Tokyo Railway Network

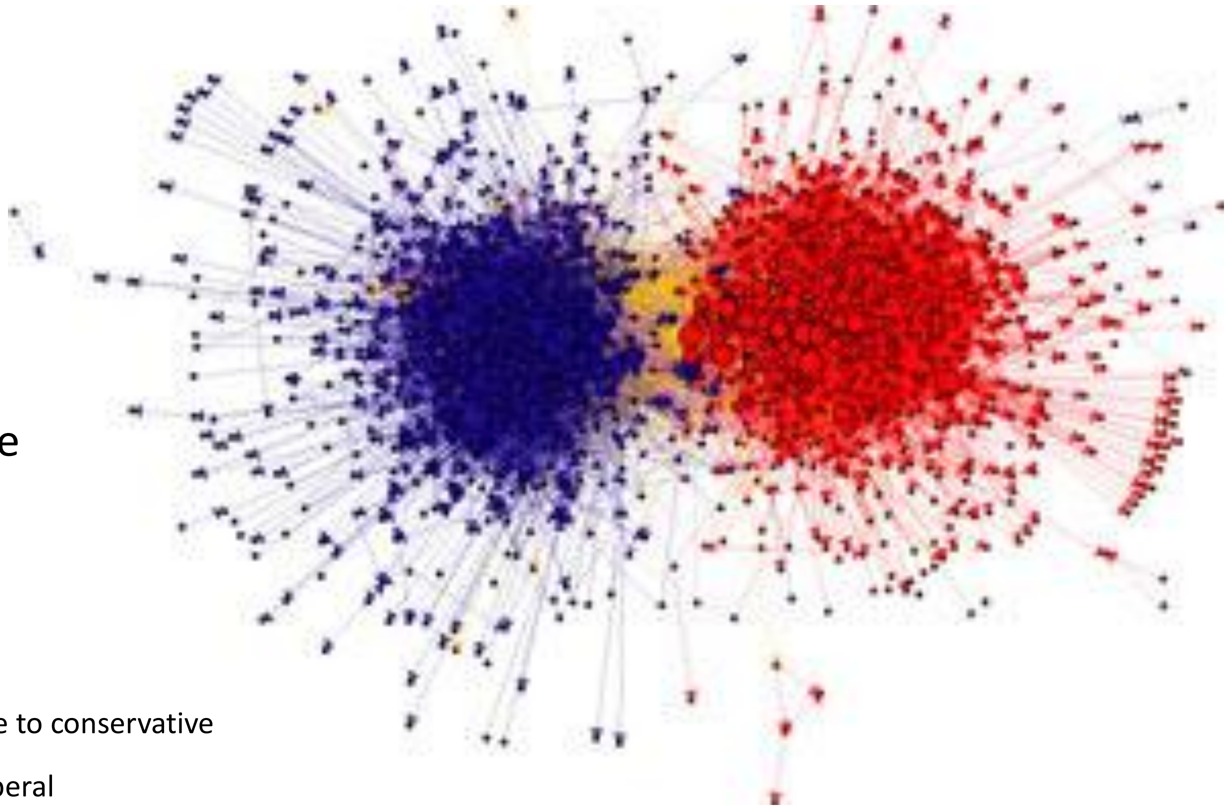
How would you represent this as a graph?
What are the nodes? Edges?
Analysis

- Shortest Path
- Traffic (max) Flow/ Bottle Neck
- Critical points / critical edges



Source: TRTA, March 2003 - Tokyo rail map

Social Network Example: Political Blogs Community Structure



Conservative

Liberal

conservative to conservative

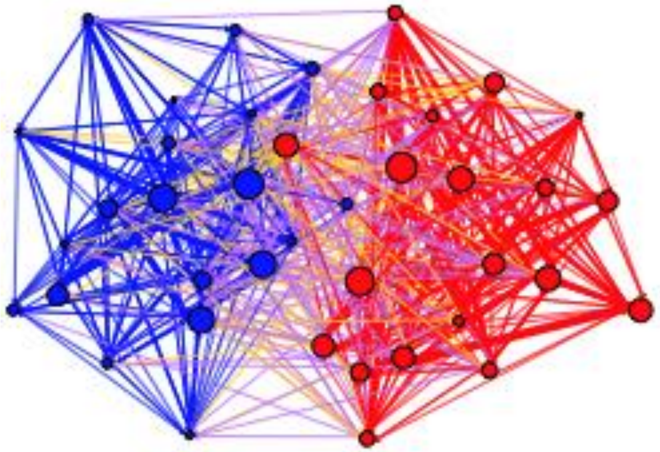
liberal to liberal

between conservative & liberal

Image source: Adamic & Glance, 2005

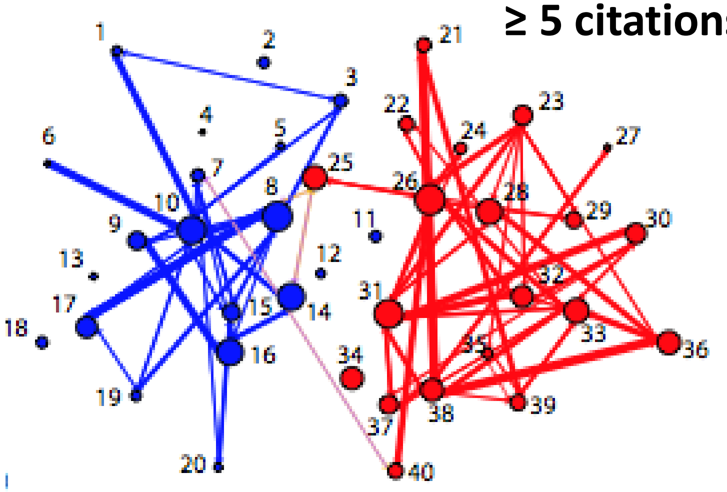
GEORGETOWN UNIVERSITY

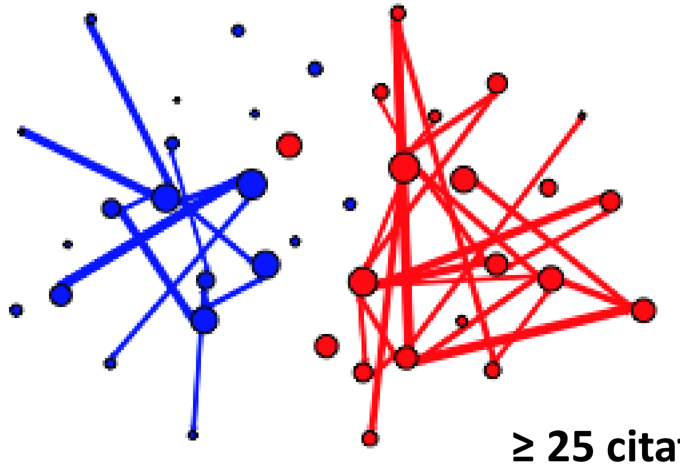
Strength of Community



Citations of posts of top 20 liberal & conservative blogs

≥ 5 citations





≥ 25 citations

Difficult Problems on a Graph

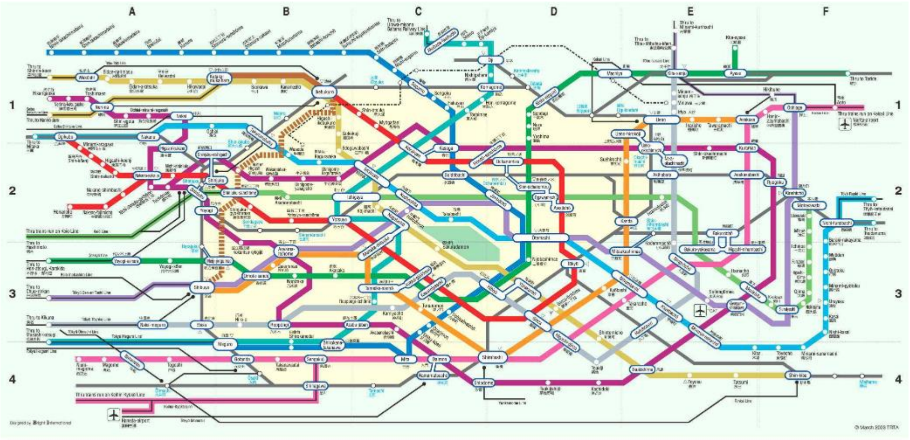
- The general graph does not have many constraint imposed on edges (unlike trees)
  - Problems that represented as graphs may have fairly high computational complexity

- What is the time complexity of the previously noted Shortest Path Algorithm?

Example: Traveling Salesman Problem

- Assume a salesman must visit a collection of cities to sell a product. The salesman wishes to ***minimize*** the total distance traveled. What is the best itinerary?

- How would you describe the solution to this problem?



GEORGETOWN UNIVERSITY

Summary of Graphs

- A very general structure used to model many problem types
- Implementations
  - Chaining
  - Adjacency Matrix
  - Sparse Matrix (graphs in application are generally very very sparse!)
- Time complexities can be fairly high compared to trees (given reduced structural constraints)

Appendix

Jeremy Bolton, PhD
Assistant Teaching Professor

Spanning Trees

- Idea: Given a graph, produce a subgraph that is a tree (that connects all nodes with n-1 edges)

- Algorithms
  - Kruskals Algorithm
  - Prim's Algorithm