

Graphs

What is a graph?

- A data structure that consists of a set of **nodes** (*vertices*) and a set of **edges** (arcs, link) that connect the nodes to each other
- The set of edges describes **relationships** among the vertices
- **Formal definition of graphs**
 - A graph G is defined as follows:

$$G=(V,E)$$

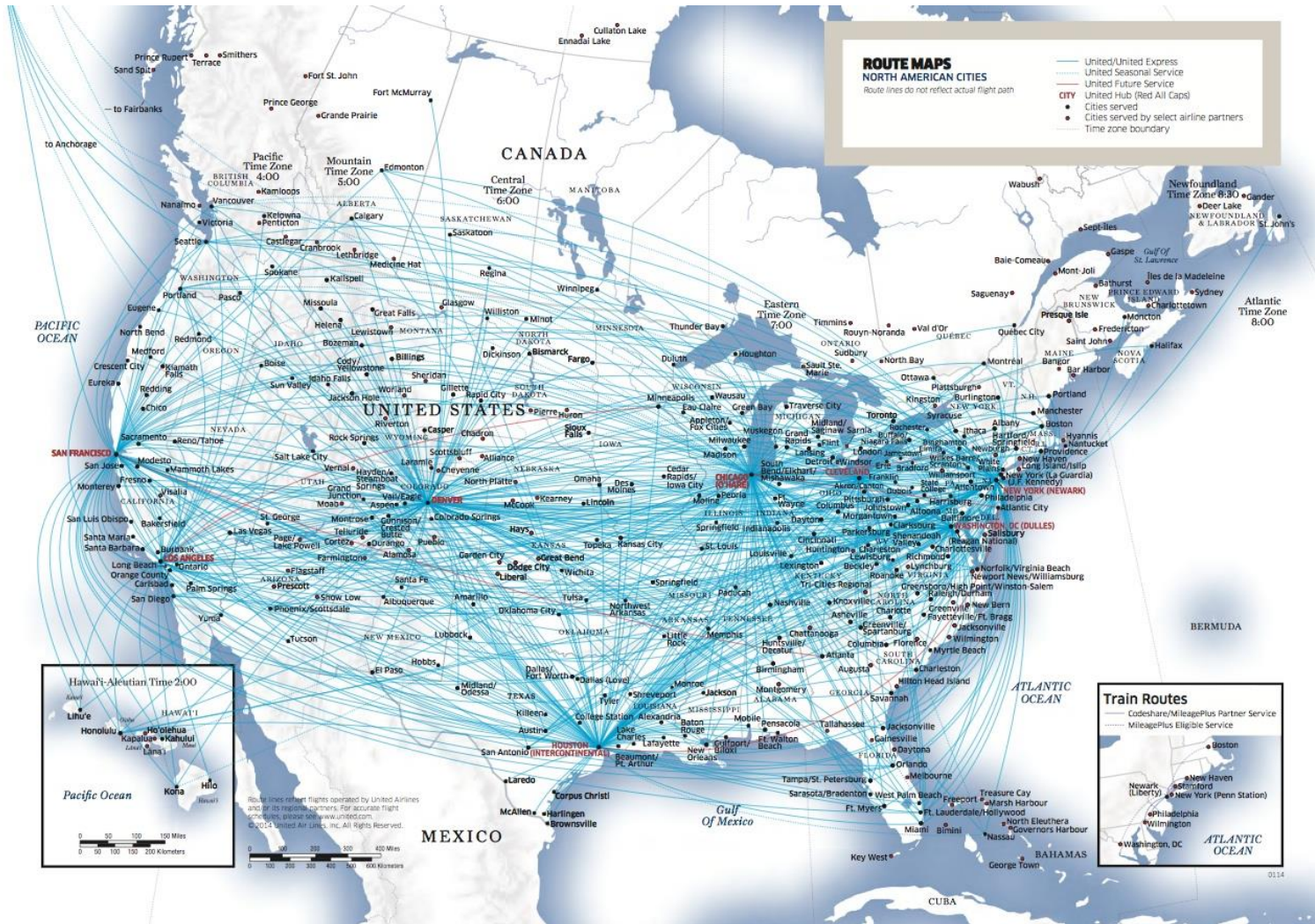
$V(G)$: a finite, nonempty set of vertices $\{v_1, v_2, \dots, v_n\}$

$E(G)$: a set of edges $\{e_1, e_2, \dots, e_m\}$ where each e_i connects two vertices (v_{i1}, v_{i2})

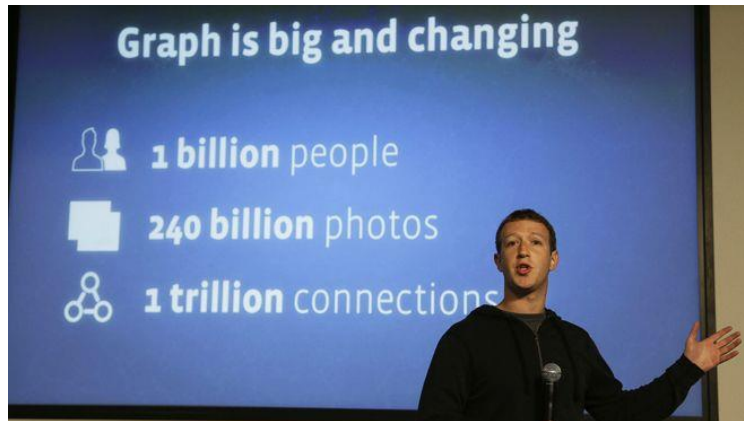
$$|E| \leq |V|^2$$

Airline routes

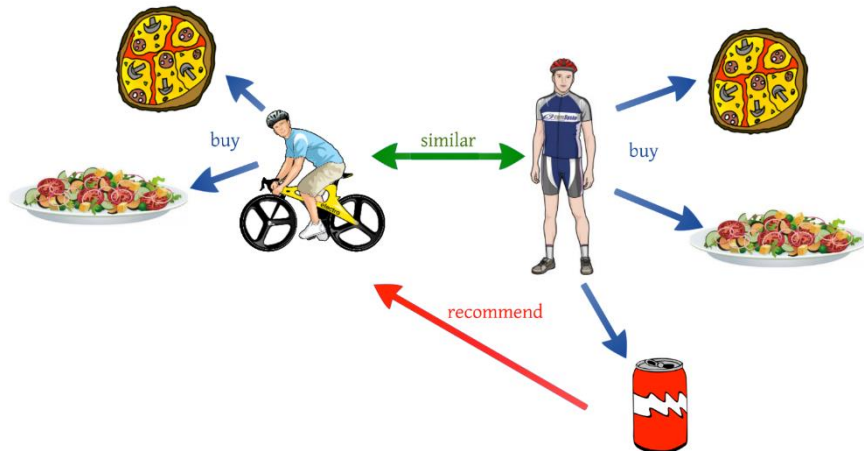
- What is the best flight from one city to another?
- Total Number of Airports by Country <http://chartsbin.com/view/1395>



Social networks (e.g., facebook)

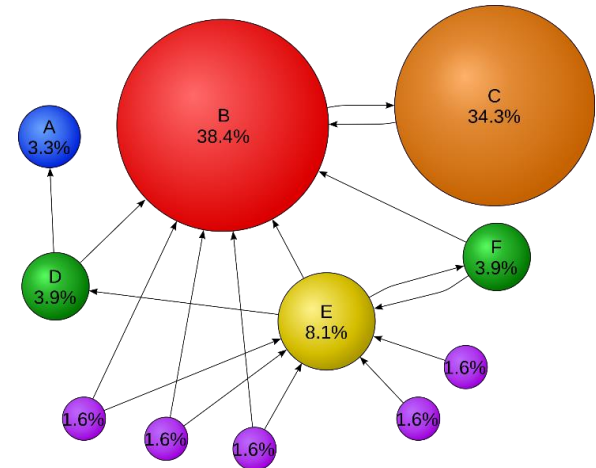


Recommendation systems (e.g., Amazon)



Google's PageRank: number and quality of a page determine the importance of a website

<https://en.wikipedia.org/wiki/PageRank>

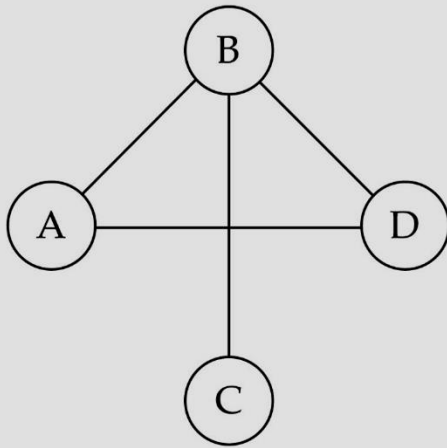


Maps and driving directions.
What is the shortest route?



Directed vs. undirected graphs

When the edges in a graph have **no direction**, the graph is called **undirected** (e.g., flight network)

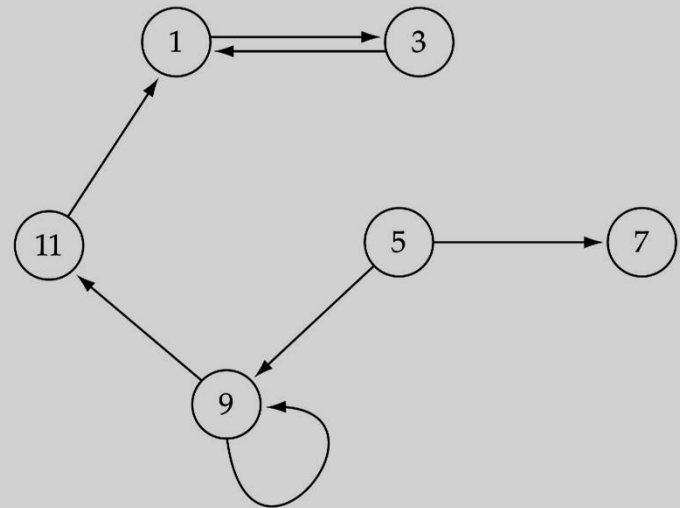


$V(\text{Graph1}) = \{ A, B, C, D \}$

$E(\text{Graph1}) = \{ (A, B), (A, D), (B, C), (B, D) \}$

When the edges in a graph have a direction, the graph is called **directed** (or **digraph**) (e.g., route network)

(b) Graph2 is a directed graph.



$V(\text{Graph2}) = \{ 1, 3, 5, 7, 9, 11 \}$

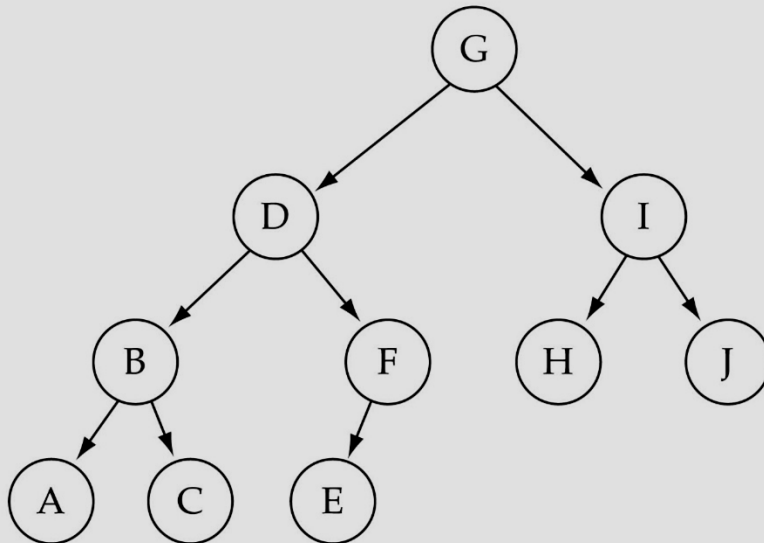
$E(\text{Graph2}) = \{ (1, 3), (3, 1), (11, 1), (9, 11), (5, 9), (9, 9), (5, 7), (7, 9) \}$

If the graph is directed, the **order** of the vertices in each edge is important.

Trees vs graphs

- Every tree is a graph with some restrictions:
 - the tree is **directed**
 - there are **no cycles** (directed or undirected)
 - there is a **directed path from the root to every node**

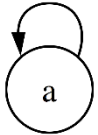
(c) Graph3 is a directed graph.



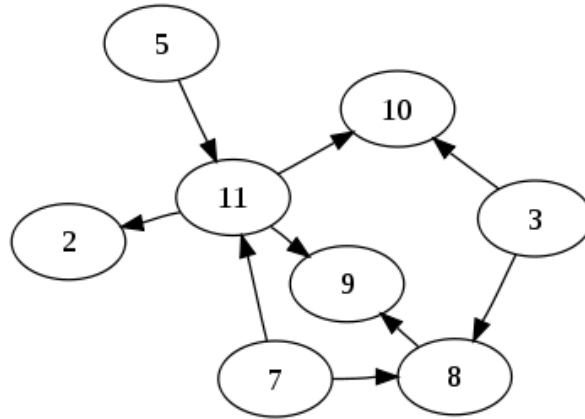
$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$

$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$

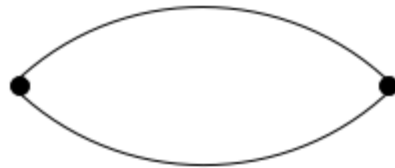
Graph terminology

- A **subgraph** consists of a subset of a graph's vertices and a subset of its edges
- **Adjacent nodes**: two nodes are adjacent if they are connected by an edge
- **Path**: a sequence of vertices that connect two nodes in a graph
- **Path length**: number of *edges* on a path, which is equal to the number of vertices - 1.
- **Simple path**: a path where all of the vertices are **distinct**.
- **Loop** (self-loop) is an edge that connects a vertex to itself. A diagram showing a circular vertex labeled 'a' with a curved arrow starting from the top of the circle and pointing back to the top, representing a self-loop.
- **Cycle**: a path that begins and ends at same vertex and its length is at least 1 (e.g., {**A**, D, E, B, **A**})
- **Simple cycle**: cycle that does not pass through other vertices more than once

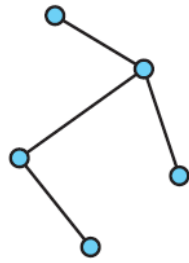
- **Directed Acyclic Graph (DAG):** directed graphs with *no cycles*.



- **Multiple edges** (parallel edges or a multi-edge) are two or more edges that are incident to the same two vertices (e.g., multiple flights between 2 cities).



- **Connected graph:** each pair of distinct vertices has a **path** between them
- **Complete graph:** each pair of distinct vertices has an **edge** between them
- A graph that is (a) connected (b) disconnected and (c) complete



(a)



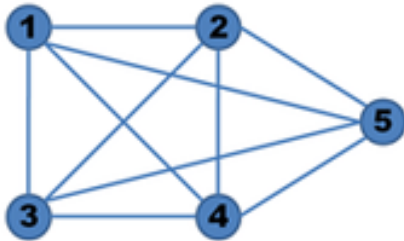
(b)



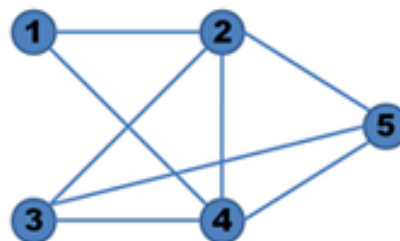
(c)

- **Dense graph:** number of edges is *close* to the **maximal** number of edges.
- **Sparse graph:** number of edges is *close* to the **minimal** number of edges.

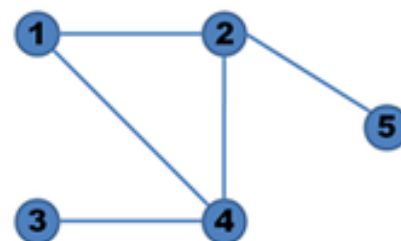
Complete Graph



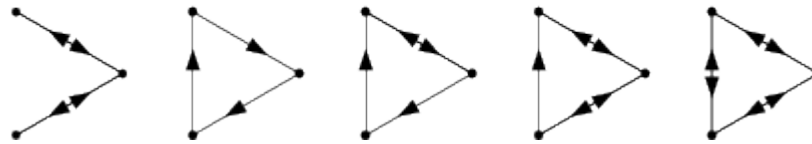
Dense Graph



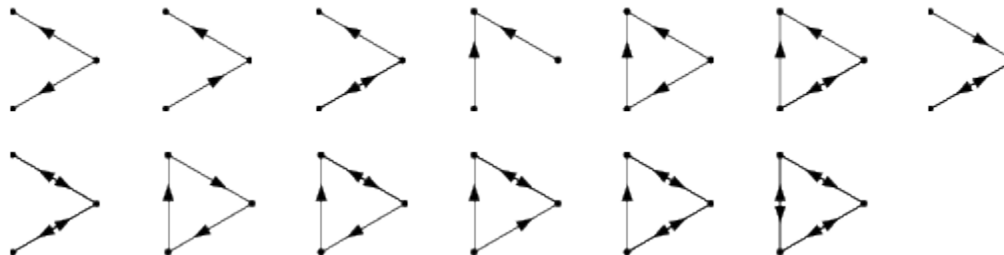
Sparse Graph



- **Degree of a node:** The number of edges it has
- The **in-degree** of a node is the number of in-edges it has
- The **out-degree** of a node is the number of out-edges it has
- A digraph is said to be **strongly connected** if there is a path between any two vertices in **both directions**. The nodes in a strongly connected digraph therefore must all have **indegree of at least 1**.

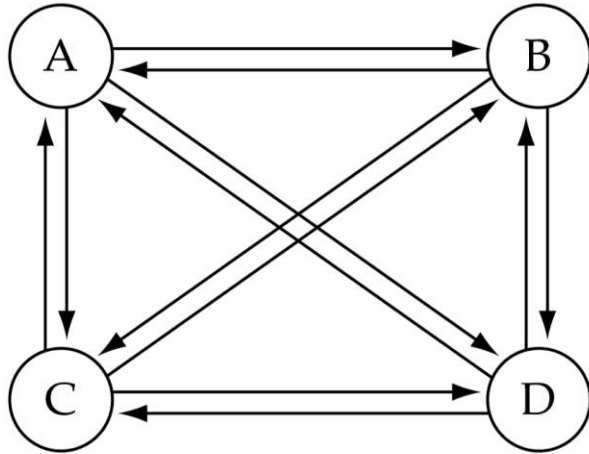


- A digraph is said to be **weakly connected** if there is a path between any two vertices, **ignoring direction**. The nodes in a weakly connected digraph therefore must all have **either outdegree or indegree of at least 1**.



What is the number of edges in a complete directed graph with N vertices?

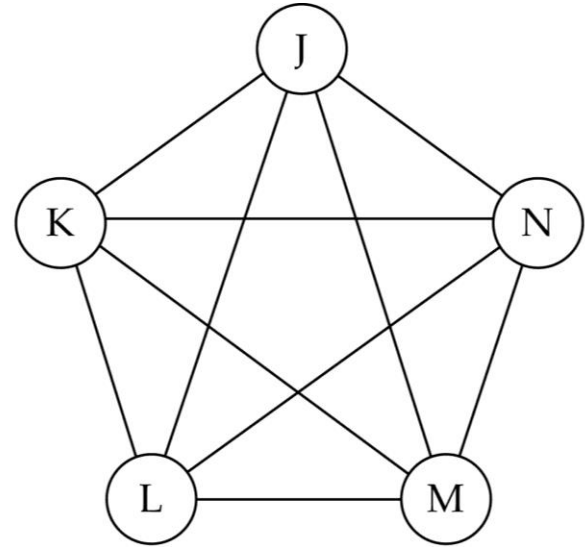
$$N * (N-1)$$



(a) Complete directed graph.

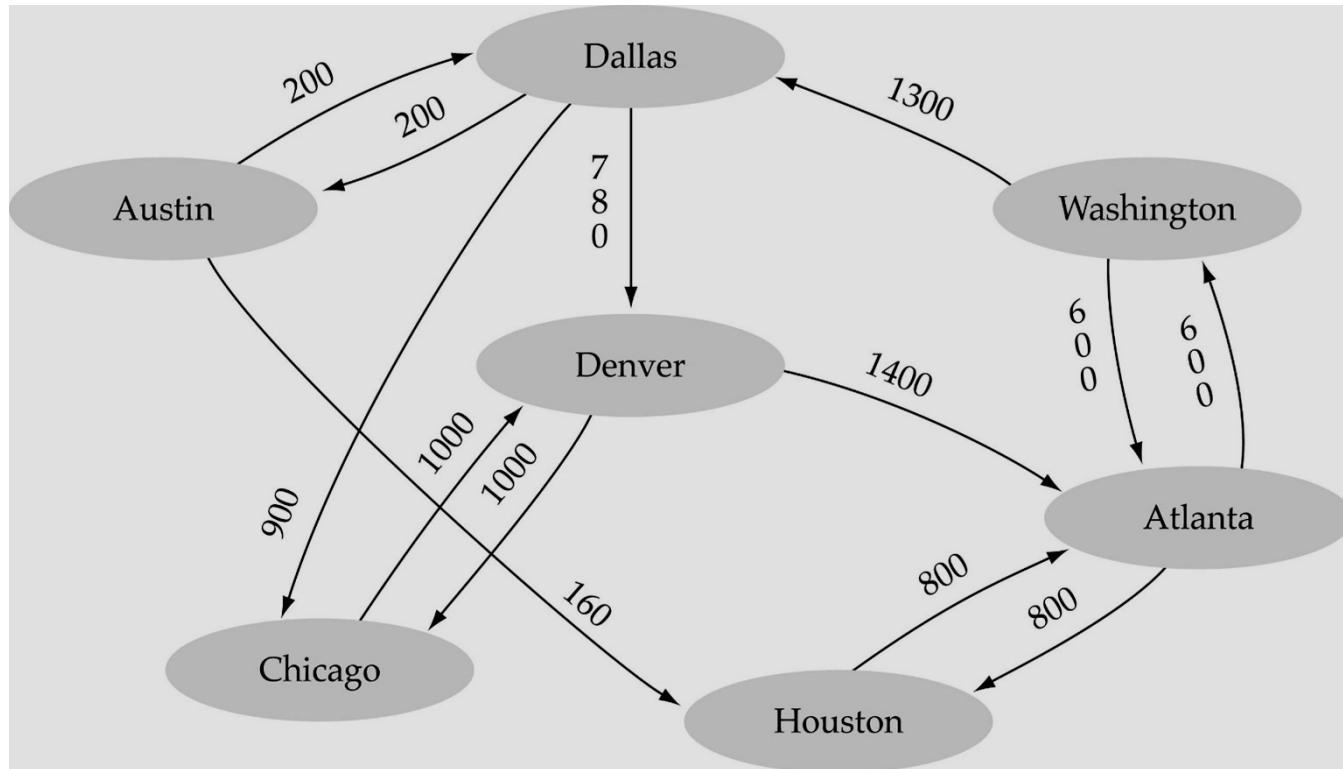
What is the number of edges in a complete undirected graph with N vertices?

$$N * (N-1) / 2$$



(b) Complete undirected graph.

Weighted graph: a graph in which each edge carries a value (**weight** or cost). The weight may represent distance, driving time, cost of building the highway, etc.





Software Engineering Intern, BS

- Chicago, IL, USA
- Qualifications: currently pursuing a Bachelor's degree in Computer Science or related technical field.
- Excellent implementation skills (C++, Java, Python).
- Experience in systems software or algorithms.
- More positions
<https://careers.google.com/jobs#!t=jo&jid=/google/software-engineering-intern-bs-summer-320-n-morgan-st-600-chicago-il-60607-usa-4246980441&>

Google - interview invitation in 2014

As you may already know, I referred you for a position as [Software Engineer, YouTube Big Data](#)! Your background looks like a potential fit for some software engineering roles we have. As a next step, I'd like to start the process by setting up a Technical/Coding Phone Interview with you and one of our engineers. We will share a Google doc with you so you need to be online with a computer and phone to complete the phone interview.

We will evaluate your expertise in the following areas:

- Algorithms and Data Structures
- Computer Science Fundamentals
- Testing Strategy
- Software Design Knowledge

To get started, please answer the following questions as soon as possible:

- 1) Please let me know several times/dates in Pacific Time/PT that you'll be available for 45-minute technical phone interview?
- 2) What is your GPA for each degree completed and in progress? Please provide transcripts (unofficial is fine).
- 3) Please let me know when you are free to connect with me for 15 minutes on the phone so I can give you an overview of our interview process.
- 4) We currently only have open positions in San Bruno and Mountain View, CA - is this okay with you?
- 5) What are your technical areas of expertise (eg. Distributed systems, Web applications & multi-tiered systems, Advanced Algorithms, Machine learning, etc...)?
- 6) What are your strongest programming languages (in order of strength, strongest first)?

Graphs as ADTs

Graph operations

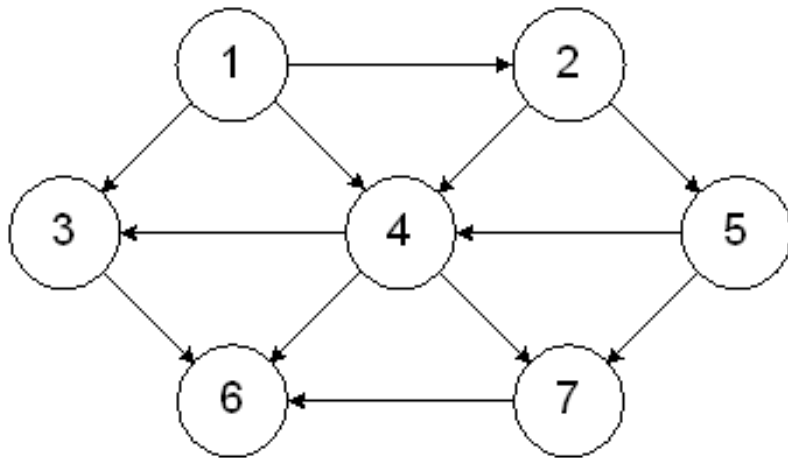
- Test whether graph is **empty**.
- Get **number of vertices** in a graph.
- Get **number of edges** in a graph.
- See whether **edge exists** between two given vertices.
- **Insert vertex** in a graph whose vertices have distinct values that differ from new vertex's value.
- **Insert edge** between two given vertices in a graph.
- **Remove specified vertex** from graph and any edges between the vertex and other vertices.
- **Remove edge** between two vertices in a graph.
- **Retrieve** from graph vertex that contains a given value.

Graph implementation

(1) Array-based implementation

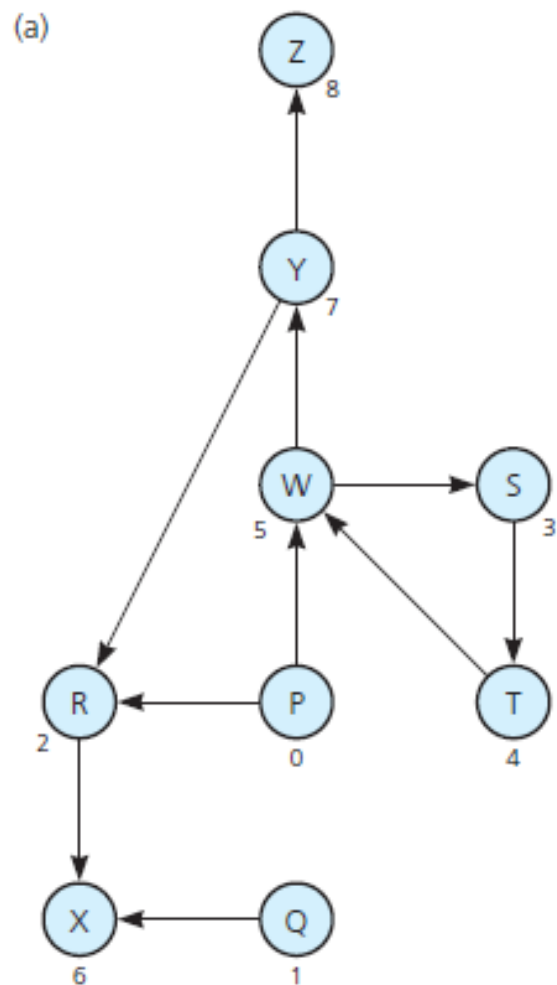
- To construct the **adjacency matrix**:

1. Number the vertices of the graph 1, 2, ..., n
2. Construct a matrix that is $n \times n$
3. For each entry in row i and column j , insert a 1 (or value) if there is an edge from vertex i to vertex j ; otherwise insert a 0

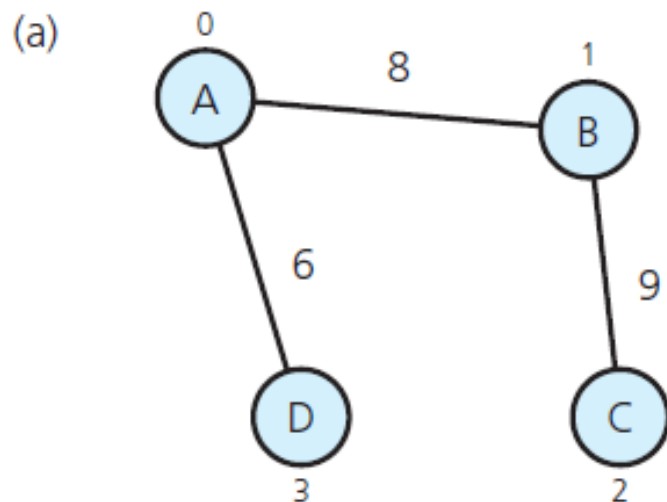


	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[1]	0	1	1	1	0	0	0
[2]	0	0	0	1	1	0	0
[3]	0	0	0	0	0	1	0
[4]	0	0	1	0	0	1	1
[5]	0	0	0	1	0	0	1
[6]	0	0	0	0	0	0	0
[7]	0	0	0	0	0	1	0

A directed graph (a) and its adjacency matrix (b)

[illegible]

A **weighted** undirected graph (a) and its adjacency matrix (b)

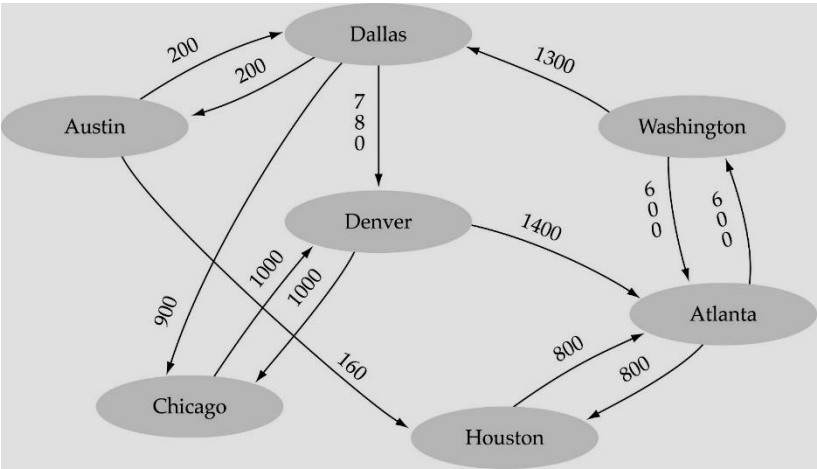


(b)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

2 arrays are required:

- 1. Vertex Array (1D array that holds vertices details)
- 2. Adjacency matrix (2D array that hold details of edges)



This representation is suitable for small graphs. **One drawback** to this type of representation is that it is often **sparse**, that is, it has a lot of **zero entries**, and thus considerable **space is wasted**. (e.g., the 2 billion of Facebook users are not friends to all users)

[0]	"Atlanta"	"
[1]	"Austin"	"
[2]	"Chicago"	"
[3]	"Dallas"	"
[4]	"Denver"	"
[5]	"Houston"	"
[6]	"Washington"	"
[7]		
[8]		
[9]		

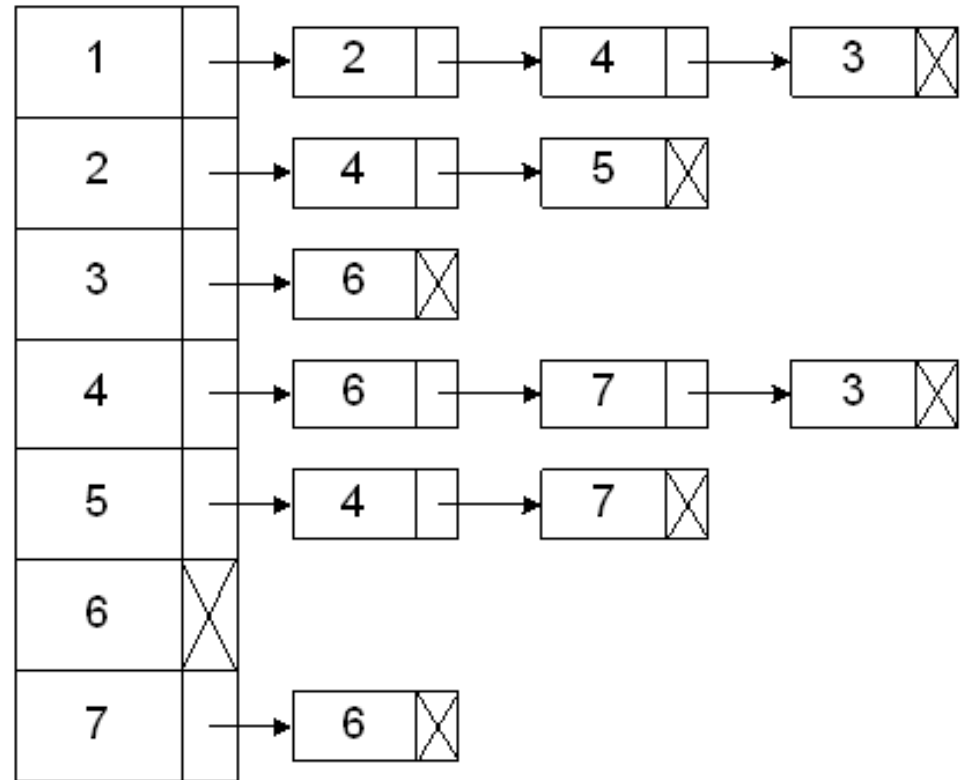
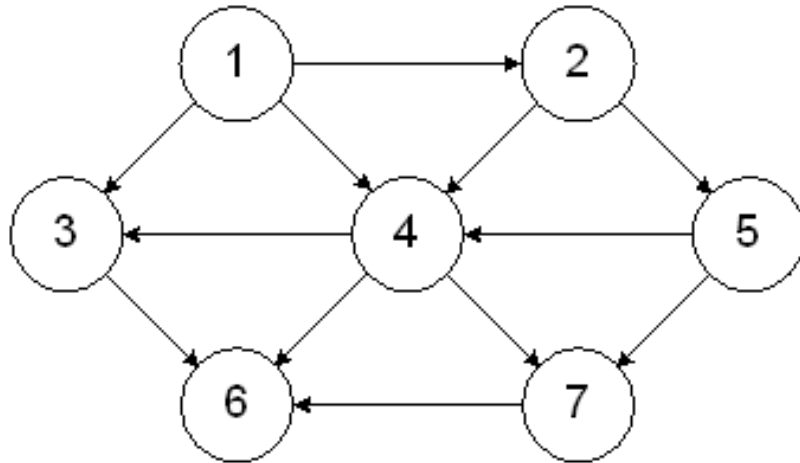
.edges

[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

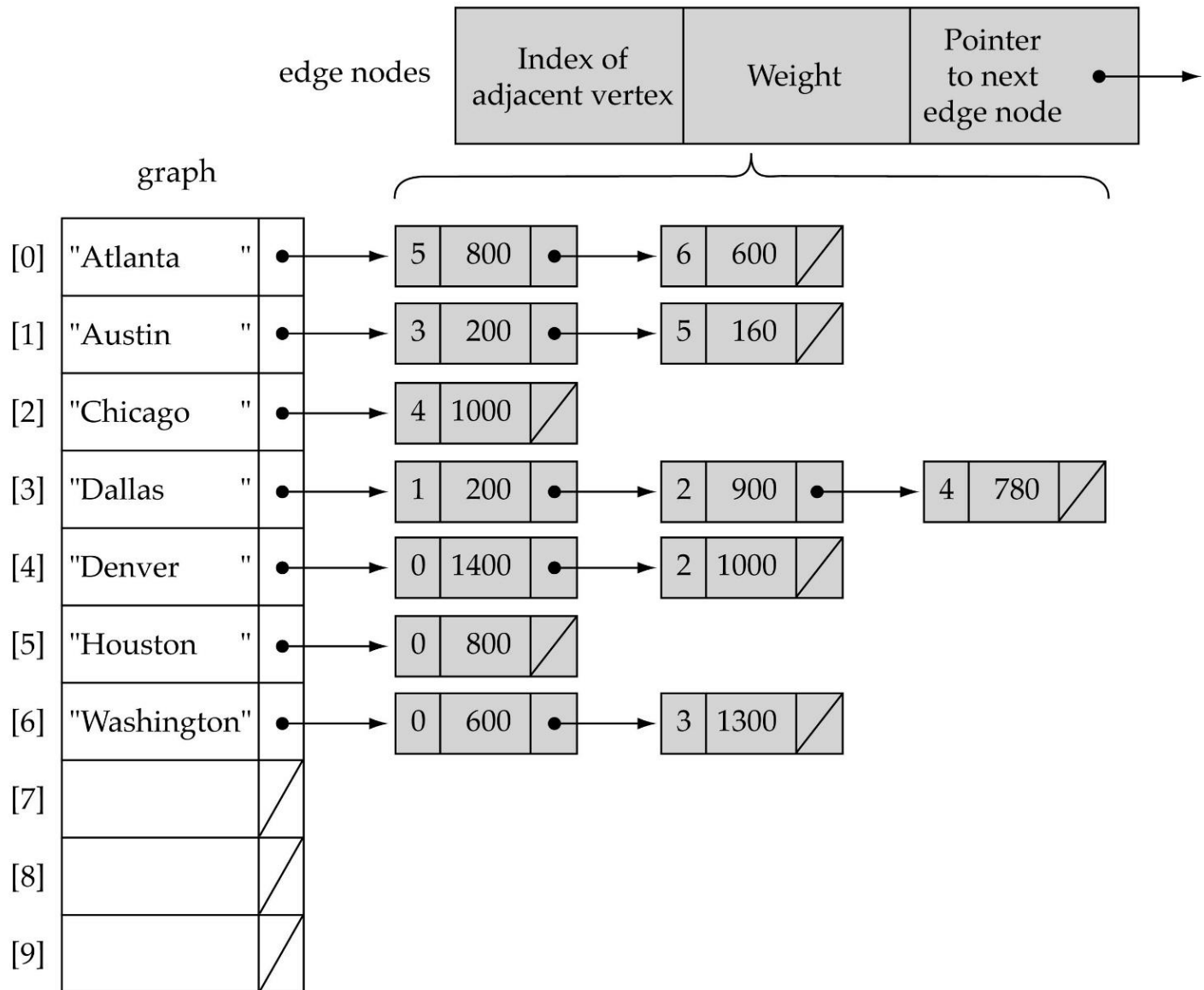
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
(Array positions marked '•' are undefined)

(2) Linked-list implementation

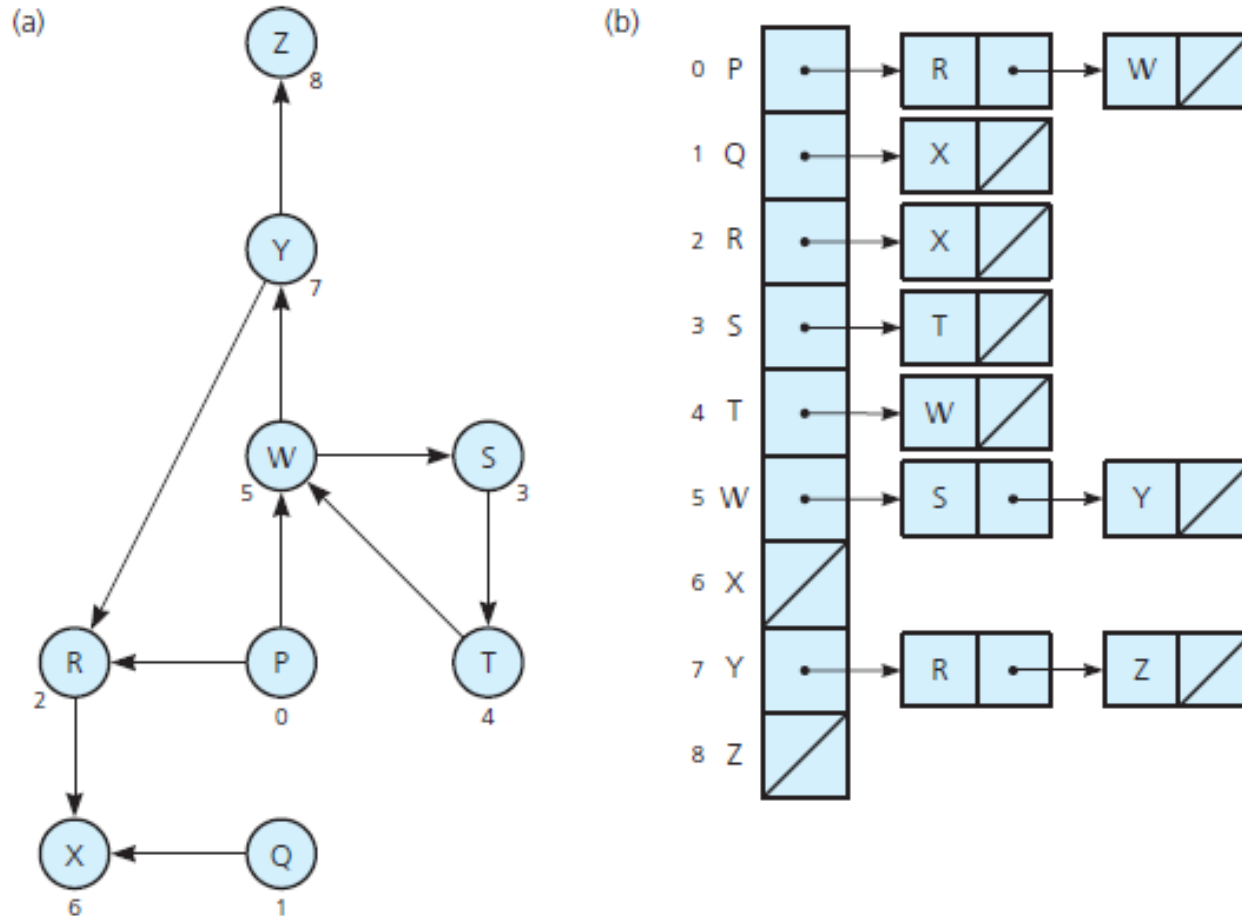
- A 1D array is used to hold the data for each vertex in the graph.
- For each vertex, there is also a pointer to a linked list of all vertices that are adjacent to the vertex (**adjacency list**).



- Adjacency list can store vertices and edges

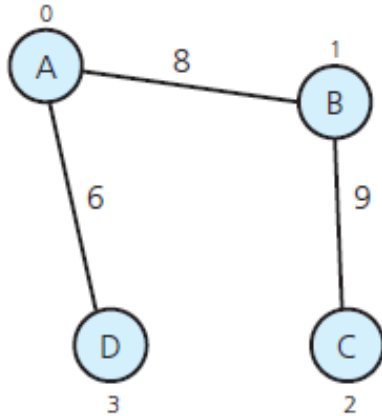


A directed graph (a) and its adjacency list (b)

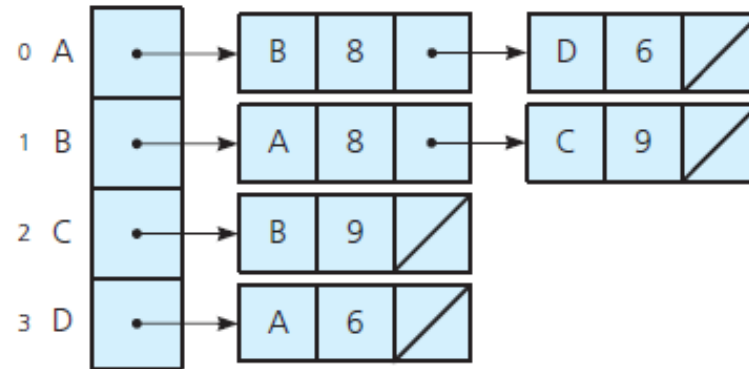


A weighted undirected graph (a) and its adjacency list (b)

(a)



(b)



Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**

- Good for **dense** graphs -- $|E| \sim O(|V|^2)$
- Connectivity between two vertices can be tested quickly

- **Adjacency list**

- Good for **sparse** graphs -- $|E| \sim O(|V|)$
- Vertices adjacent to another vertex can be found quickly

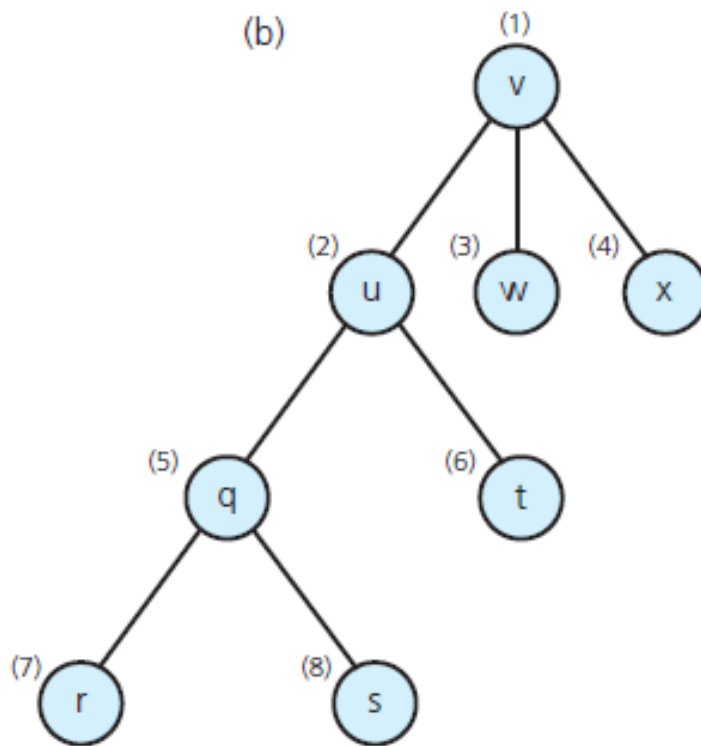
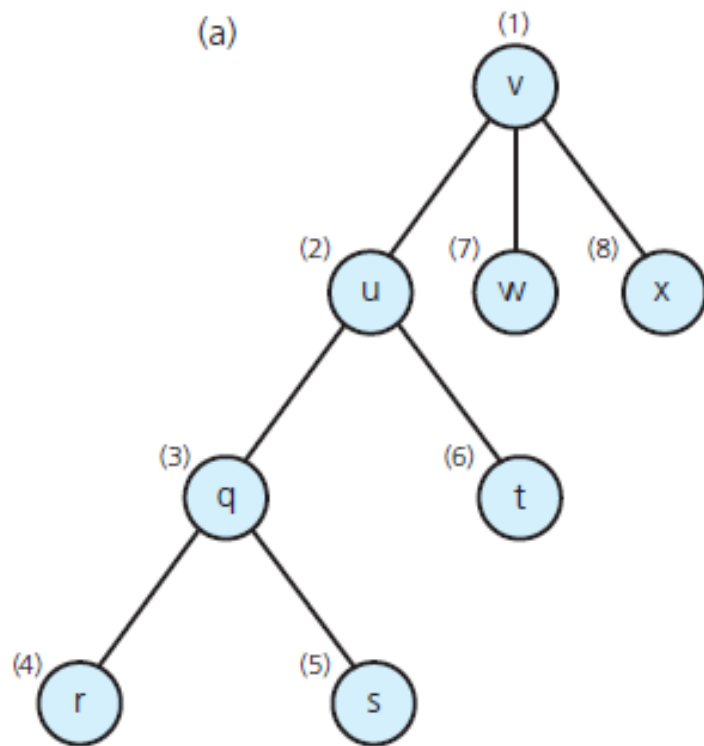
Graph searching/traversal

- **Methods:** Depth-First-Search (DFS) or Breadth-First-Search (BFS)
- Recall that a binary **tree has no cycles**. Also, starting at the root node, we can traverse the entire tree. On the other hand, **a graph might have cycles** and we might not be able to traverse the entire graph from a single vertex (for example, if the graph is not connected). Therefore, we must **keep track of the vertices that have been visited**.

(1) Depth-First-Search (DFS)

- DFS is similar to traversing a **binary tree preorder traversal** with some differences.
- What is the idea behind DFS?
 - Given a starting vertex v
 - Pick an arbitrary adjacent vertex, visit it.
 - Travel as far as you can down a path
 - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- Depth-first search is useful for **testing** a number of properties of graphs, including whether there is a **path** from one vertex to another and whether or not a graph is **connected**.
- DFS can be implemented efficiently using a *stack*
- <https://www.cs.usfca.edu/~galles/visualization/DFS.html>

Visitation order for (a) depth-first search; (b) breadth-first search



Depth-First-Search – version 1 - Recursive

Assume all vertices initially unmarked (not visited)

Outer loop to call the depth-first traversal // to visit all vertices

for each vertex, v , in the graph

if v is not visited

DFS(g , v)

DFS(graph g , vertex v)

mark v as visited

for each vertex w adjacent to v

if w is not visited

DFS(g , w)

```
void depthFirstTraversal() //In C++
{
    bool *visited; // pointer to create the array to keep
    // track of the visited vertices
    visited = new bool[gSize]; //gSize = graph size
    for (int index = 0; index < gSize; index++)
        visited[index] = false;
    // For each vertex that is not visited, do a depth first traversal
    for (int index = 0; index < gSize; index++)
        if (!visited[index])
            DFS(index,visited);
    delete [] visited;
} //end depthFirstTraversal
```



```
void DFS(int v, bool visited[]) //In C++
{
    visited[v] = true;
    cout << " " << v << " "; //visit the vertex
    linkedListIterator<int> graphIt;
    //for each vertex adjacent to v
    for (graphIt = graph[v].begin(); graphIt != graph[v].end(); ++graphIt)
    {
        int w = *graphIt;
        if (!visited[w])
            DFS(w, visited);
    }
}
```

Depth-First-Search (DFS) – version 2 uses stack

- Use non-recursive version if recursion depth is too big (over a few thousands).

DFS(graph g, vertex v)

s = a new empty stack

s.push(v)

Mark v as visited

while(! s.isEmpty())

{

x = s.top() // The vertex on the top of the stack

if (no unvisited vertices are adjacent to x)

 s.pop() // Backtrack

else

{

 select an unvisited vertex u adjacent to x

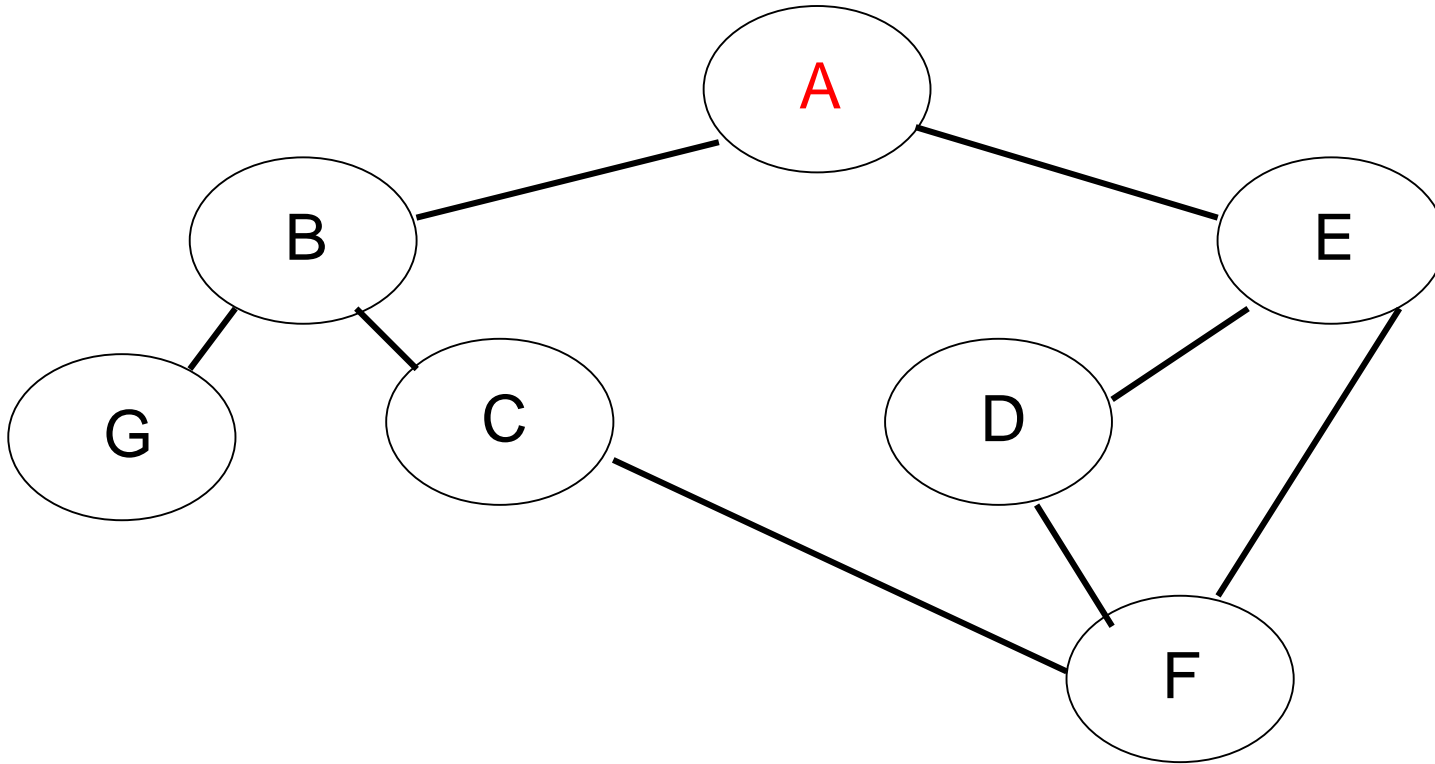
 s.push(u)

 Mark u as visited

}

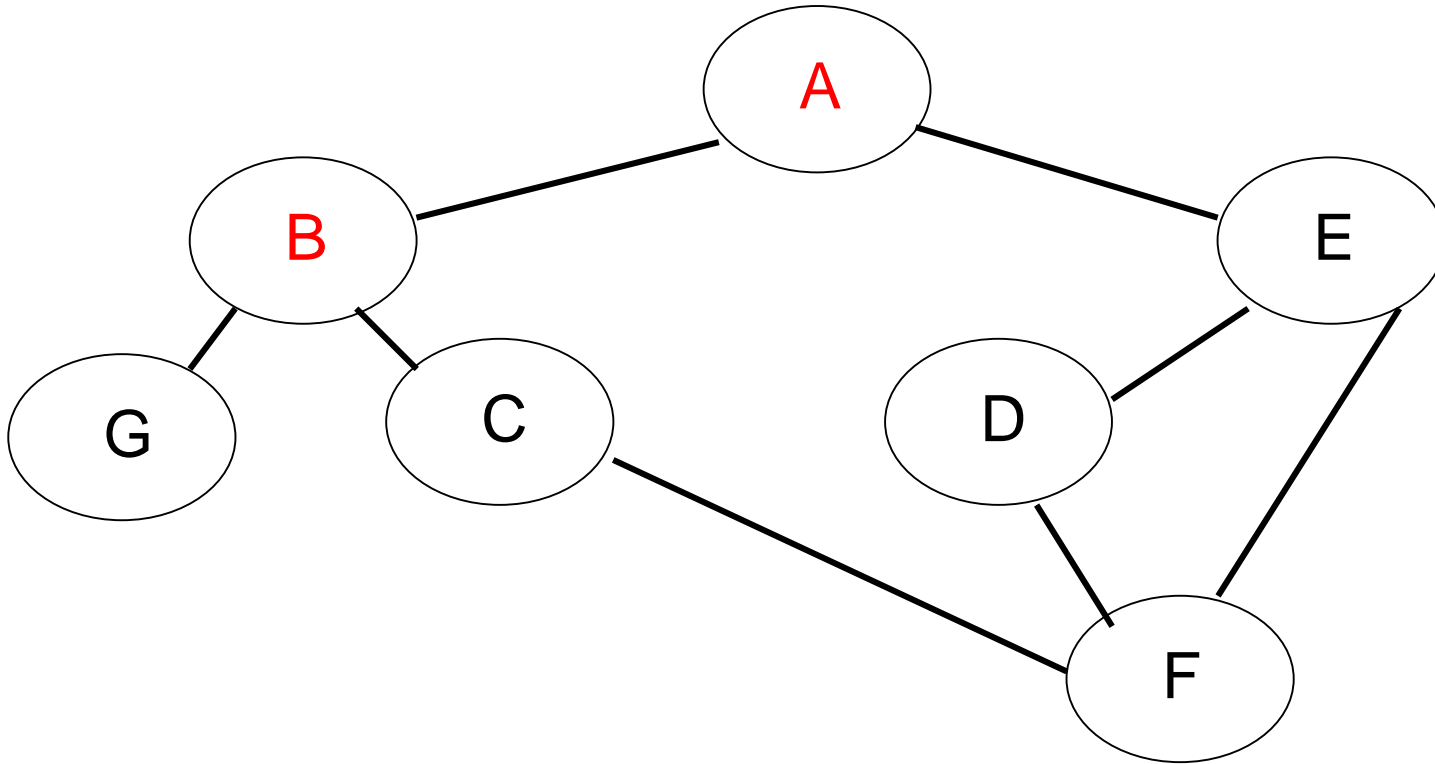
}

Current vertex: A



Start with A. Mark it.

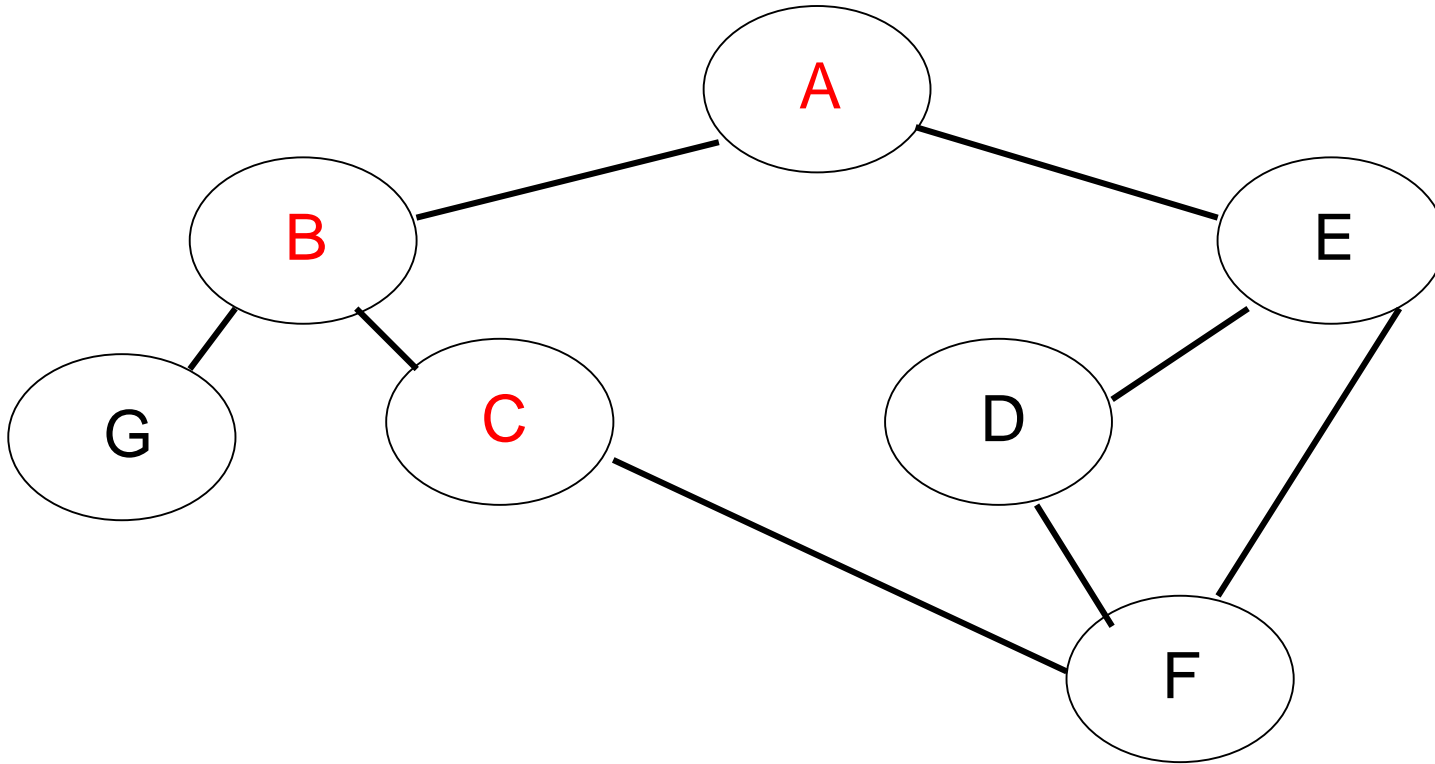
Current: B



Expand A's adjacent vertices. Pick one (B).

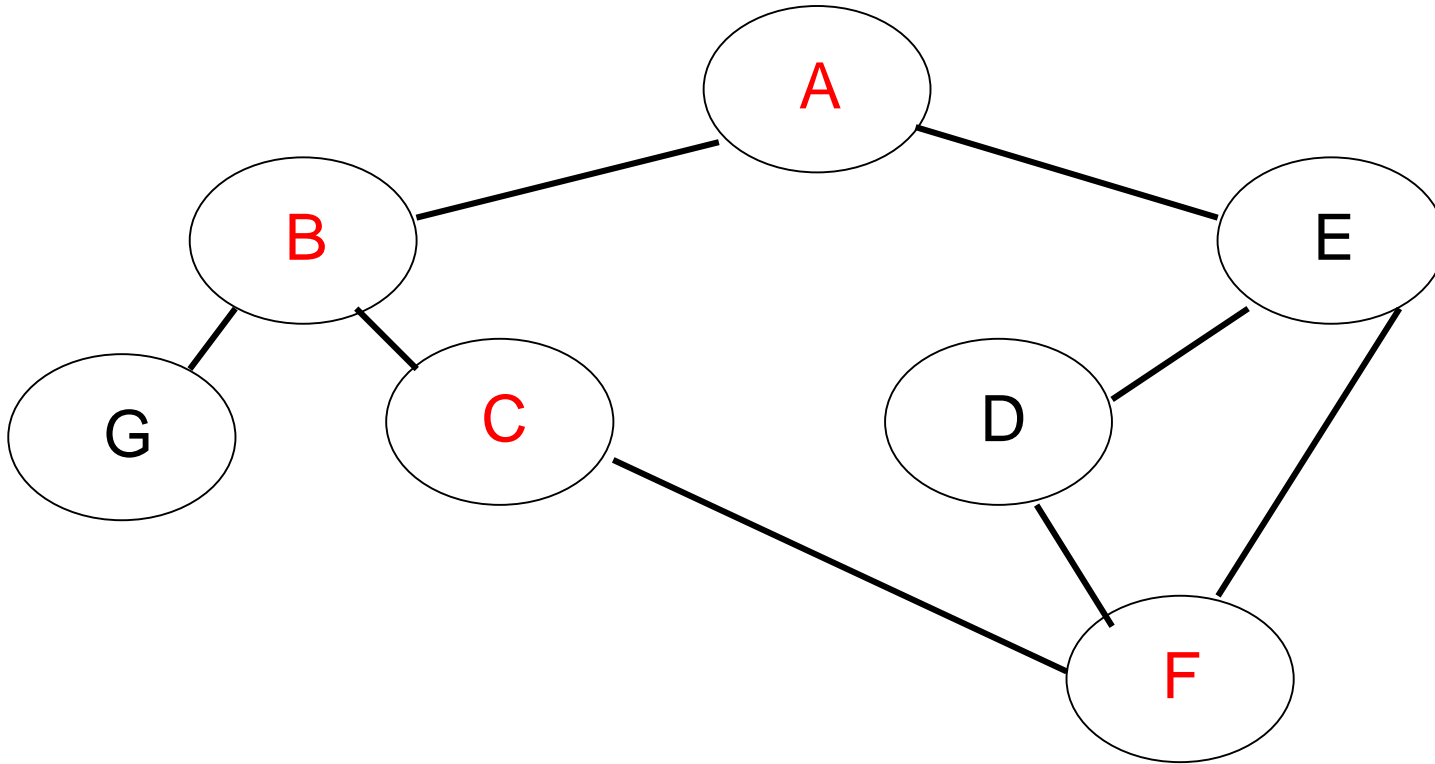
Mark it and re-visit.

Current: C



Now expand B, and visit its neighbor, C.

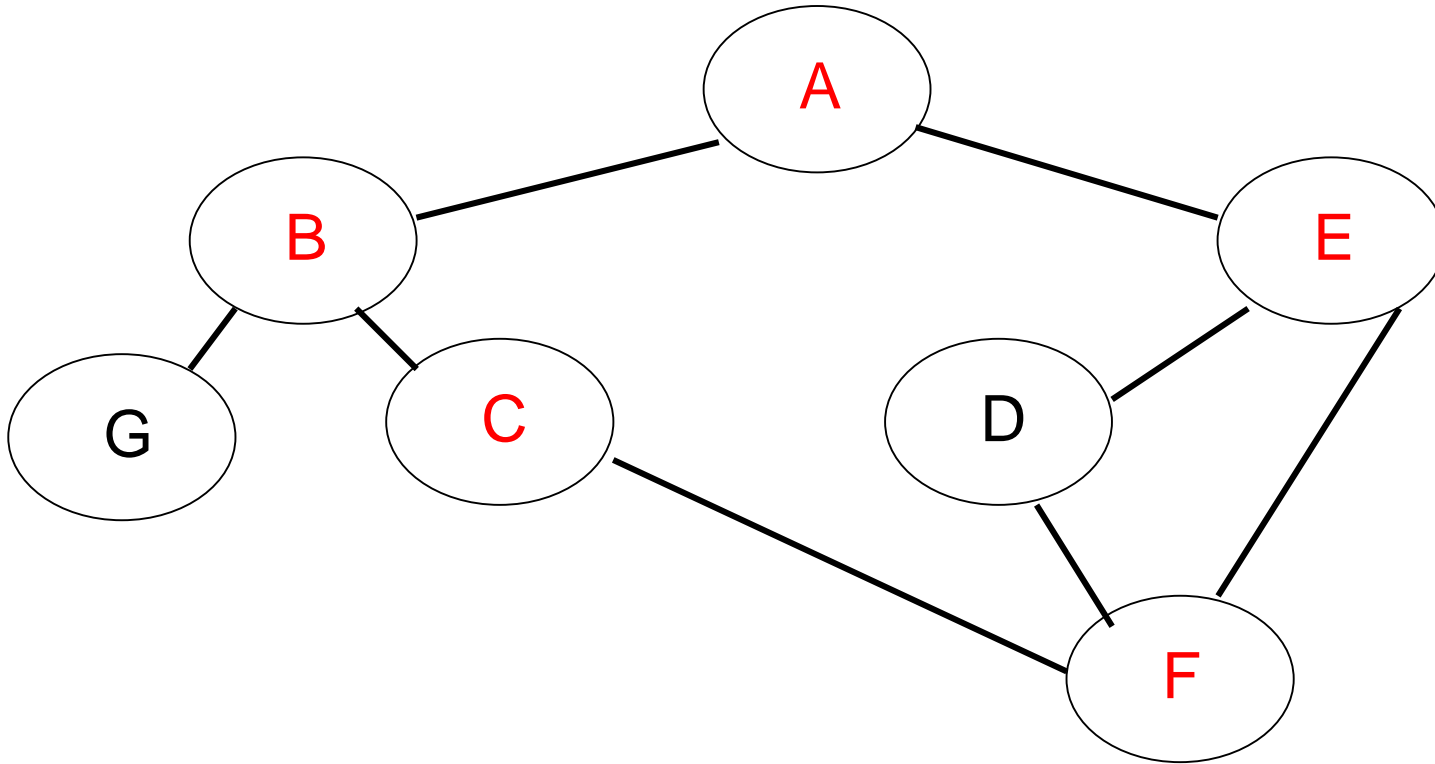
Current: F



Visit F.

Pick one of its neighbors, E.

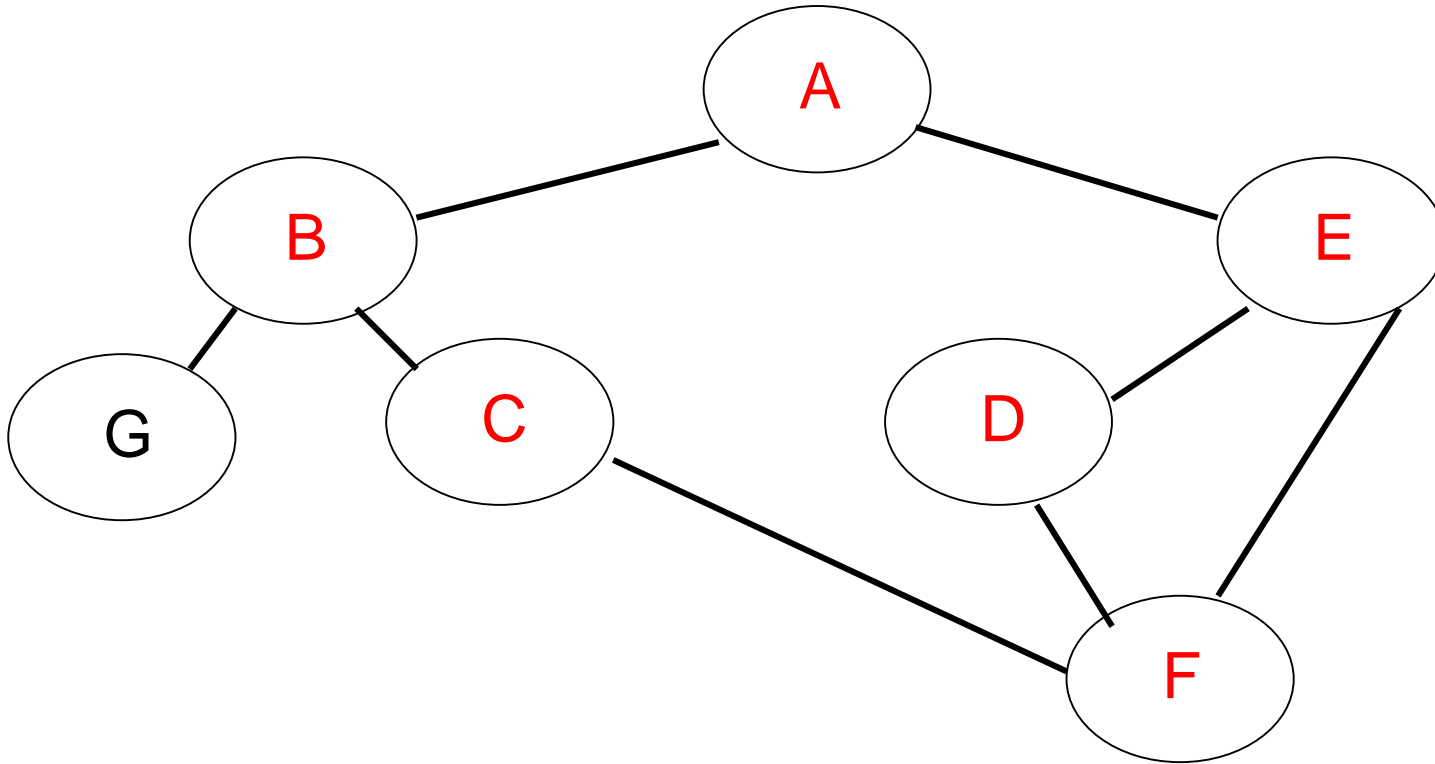
Current: E



E's adjacent vertices are A, D and F.

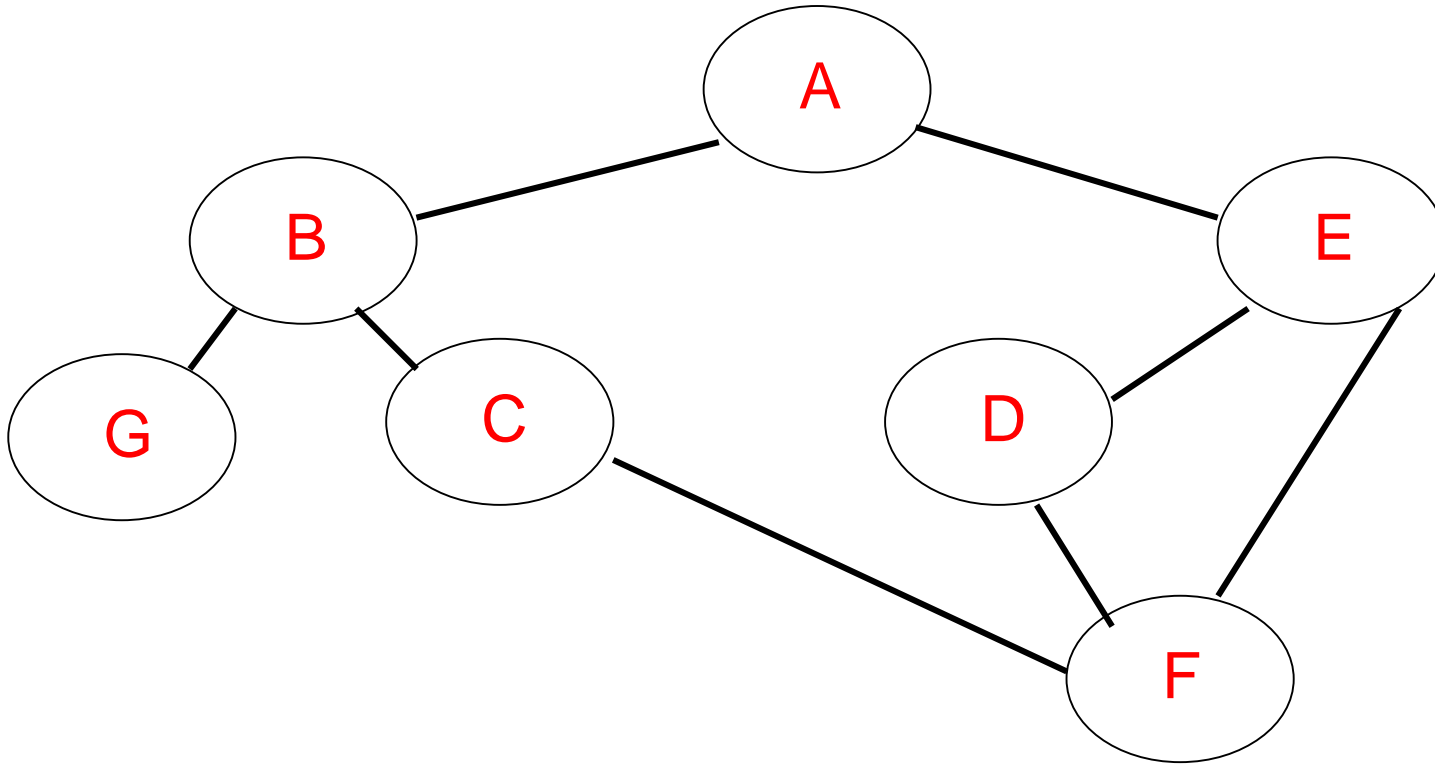
A and F are marked, so pick D.

Current: D



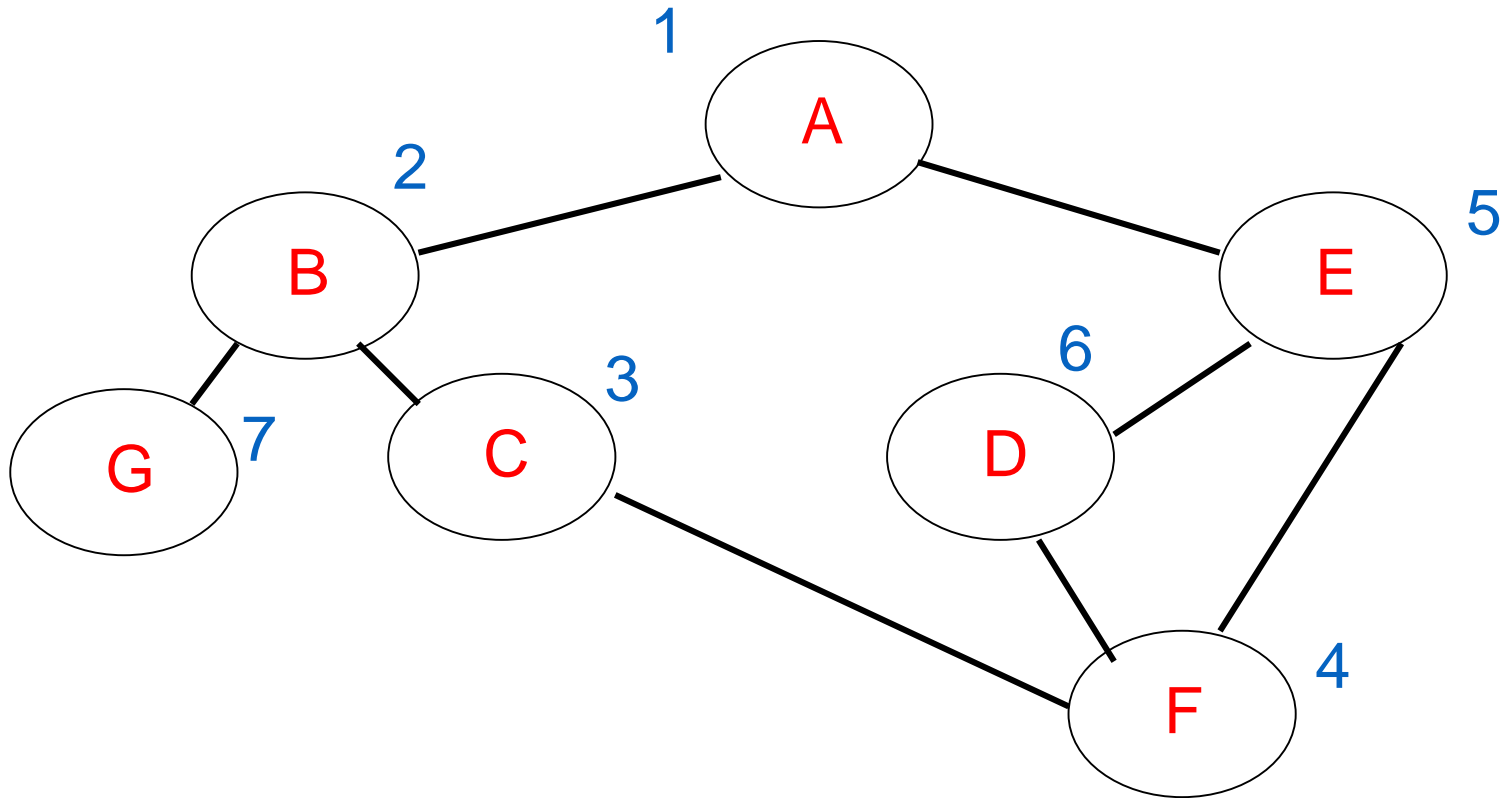
Visit D. No new vertices available. Backtrack to E. Backtrack to F. Backtrack to C. Backtrack to B

Current: G



Visit G. No new vertices from here. Backtrack to B. Backtrack to A. E already marked so no new.

Current:



Done. We have explored the graph in order:

A B C F E D G

(2) Breadth First Search (BFS)

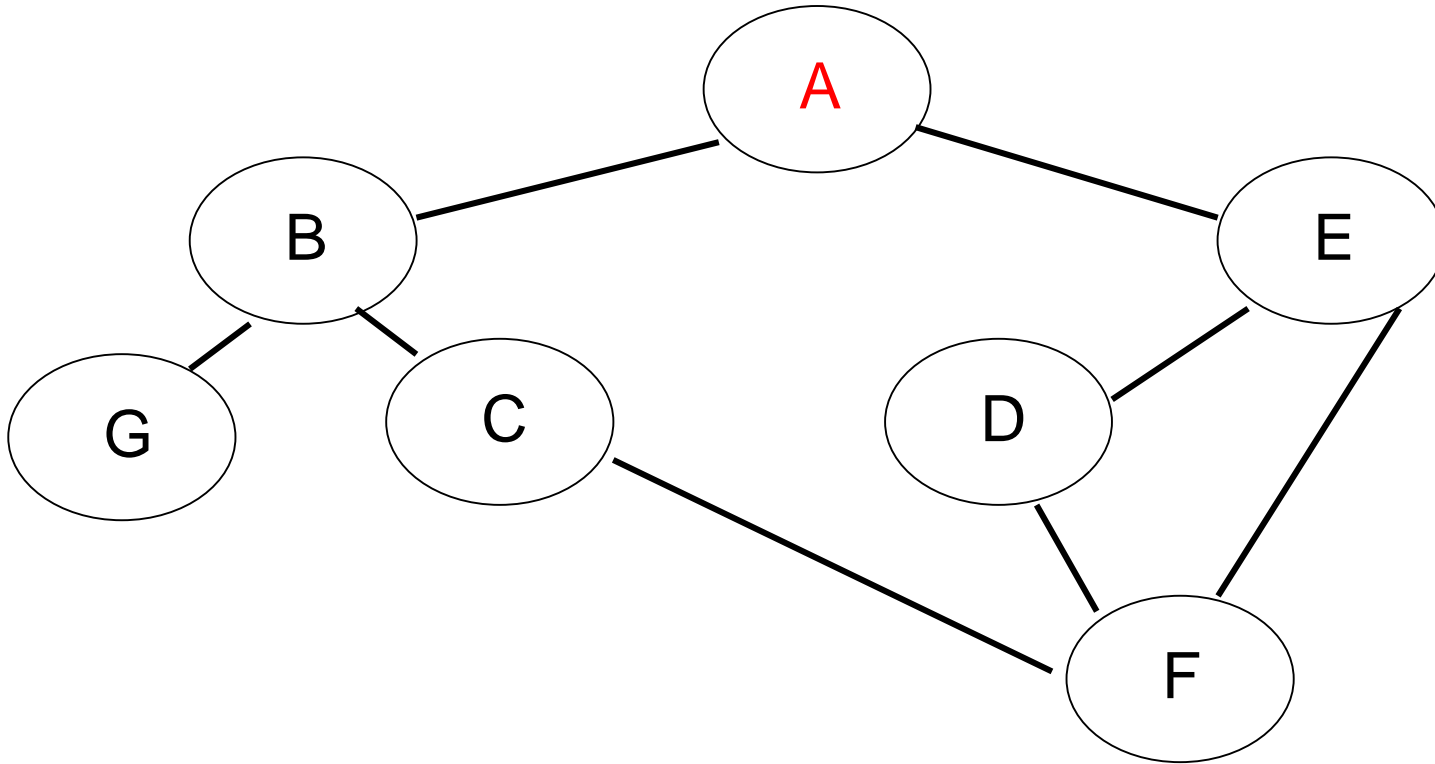
- What is the idea behind BFS?

- Visit **all vertices adjacent** to vertex before going forward
- Back up *as **far as possible*** when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- The breadth first traversal of a graph is similar to traversing a **binary tree level by level** (the nodes at each level are visited from left to right). All nodes at any level, i , are visited before visiting the nodes at level $i + 1$.
- As in the case of depth first traversal, because it might not be possible to traverse the entire graph from a single vertex, the breadth first traversal also traverses the graph from each vertex that is not visited.
- To implement the breadth first search algorithm, we use a **queue**.

BFS for general graphs

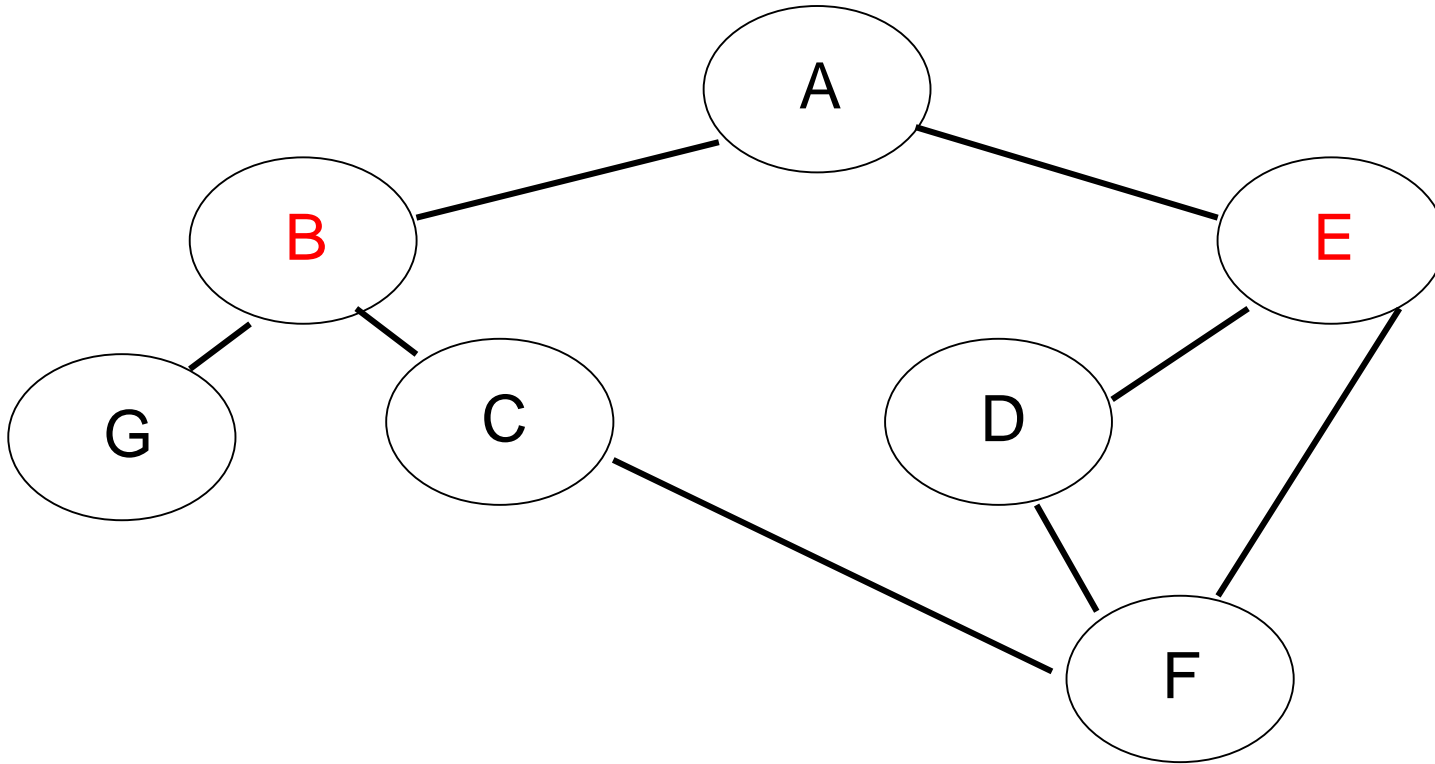
- This version assumes vertices have two children
 - left, right
 - This is trivial to fix
- But still not good for general graphs
- It does not handle cycles

Queue: A



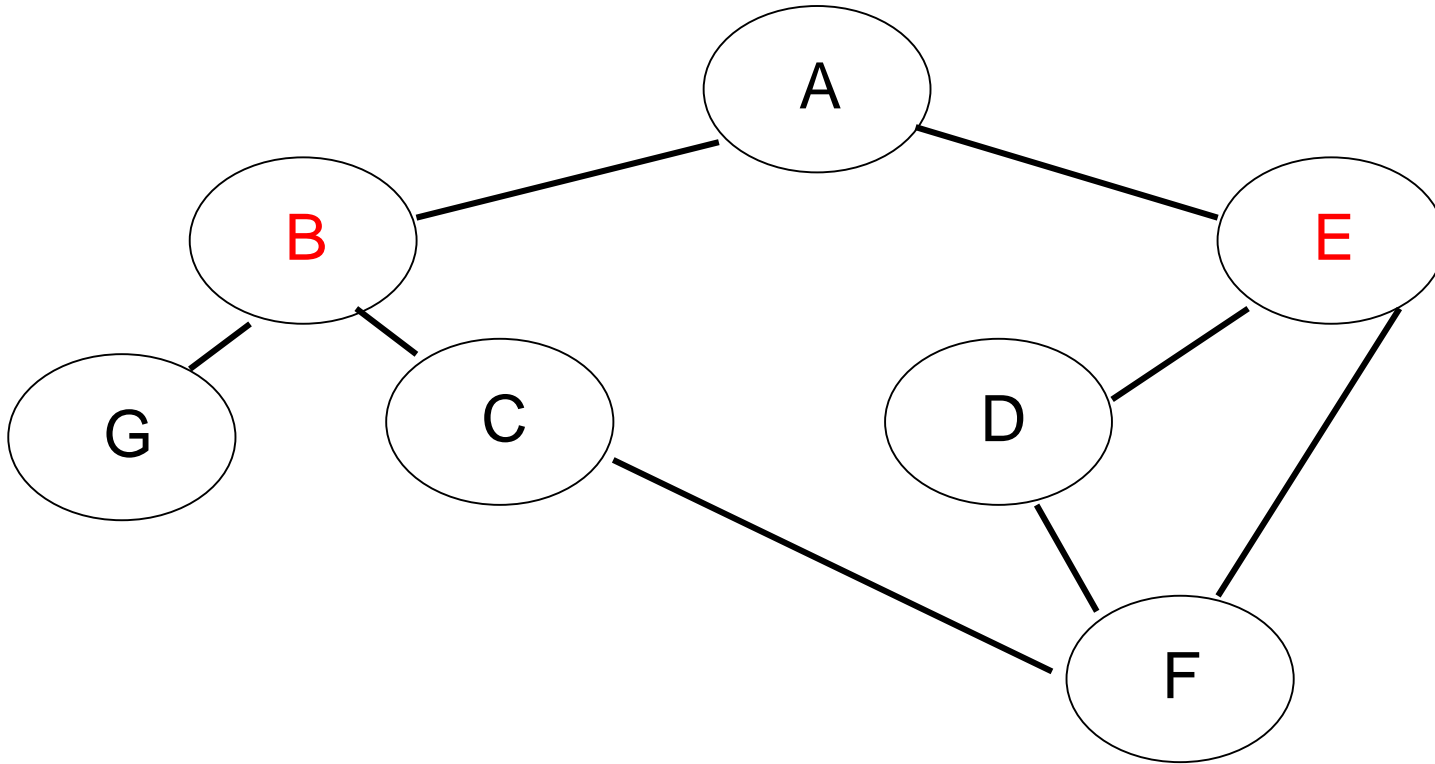
Start with A. Put in the queue (marked red)

Queue: A B E



B and E are next

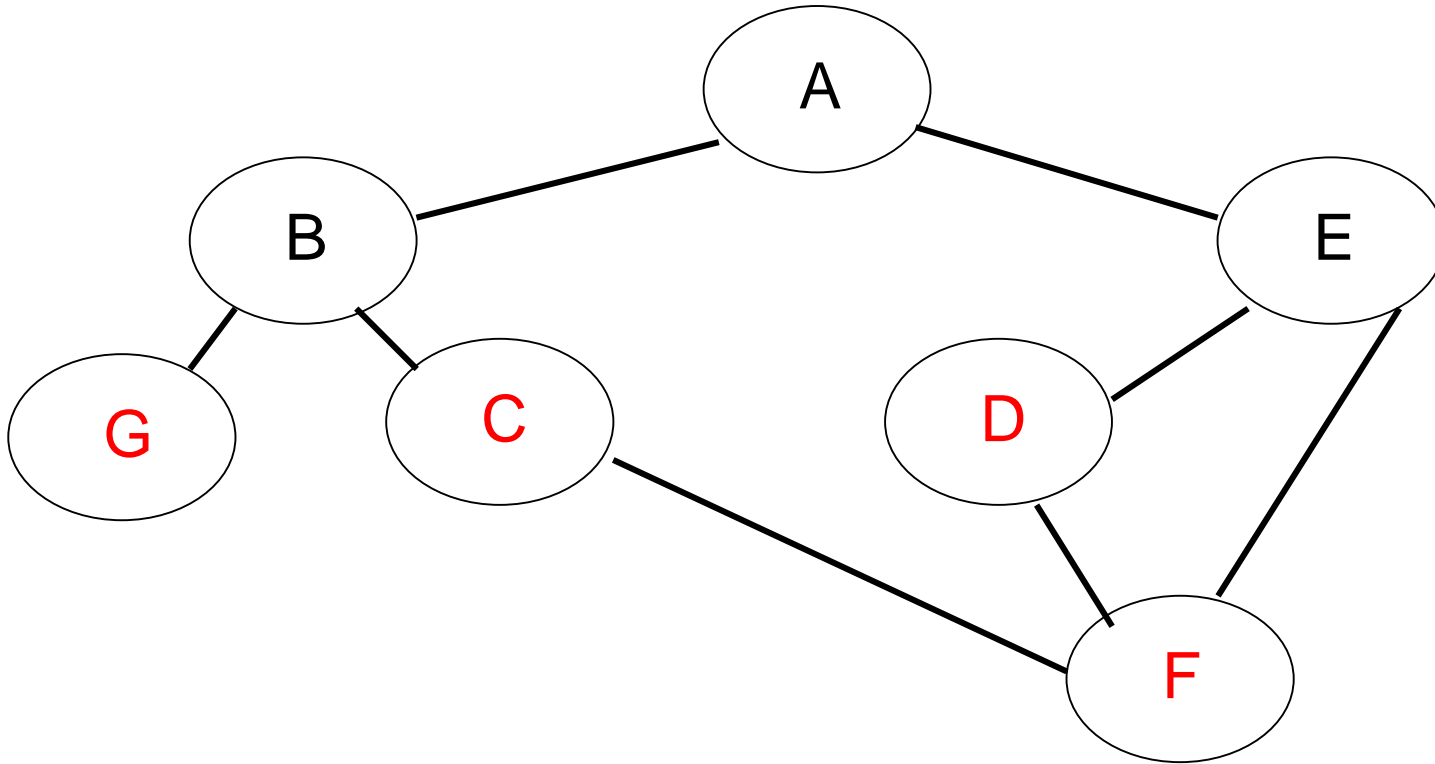
Queue: A B E C G D F



When we go to B, we put G and C in the queue

When we go to E, we put D and F in the queue

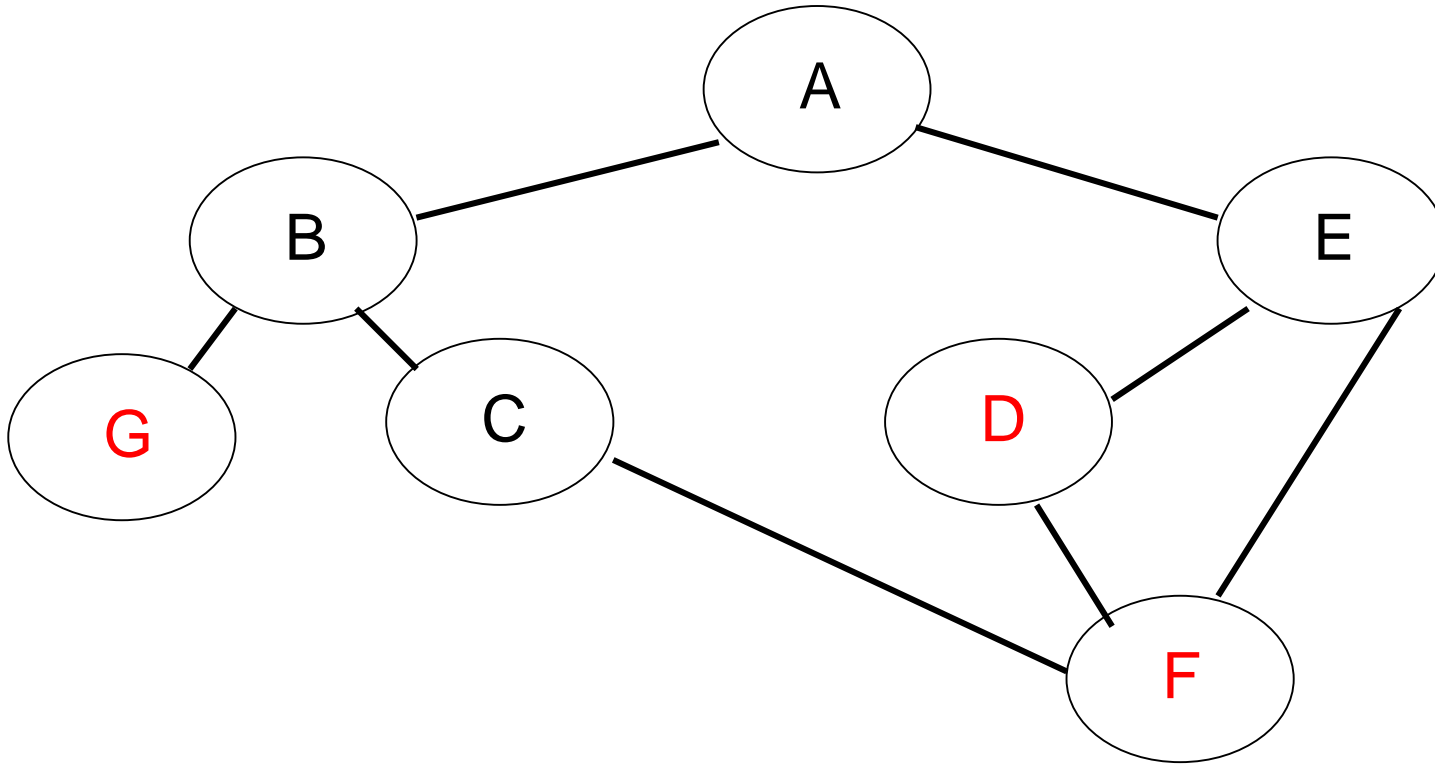
Queue: A B E C G D F



When we go to B, we put G and C in the queue

When we go to E, we put D and F in the queue

Queue: A B E C G D F F



Suppose we now want to expand C.

We put F in the queue again!

Generalizing BFS

- **Cycles:**
 - We need to save auxiliary information
 - Each **node** needs to be **marked**
 - **Visited:** No need to be put on queue
 - **Not visited:** Put on queue when found
- Need to put all adjacent vertices in queue

BFS(graph g, vertex v)

unmark all vertices in g

For each vertex v in the graph,

 If v is not visited

 add v to the queue

 mark v as visited

 while (not empty(q))

 remove vertex u from the queue

 for each vertex w adjacent to u

 if w is not visited

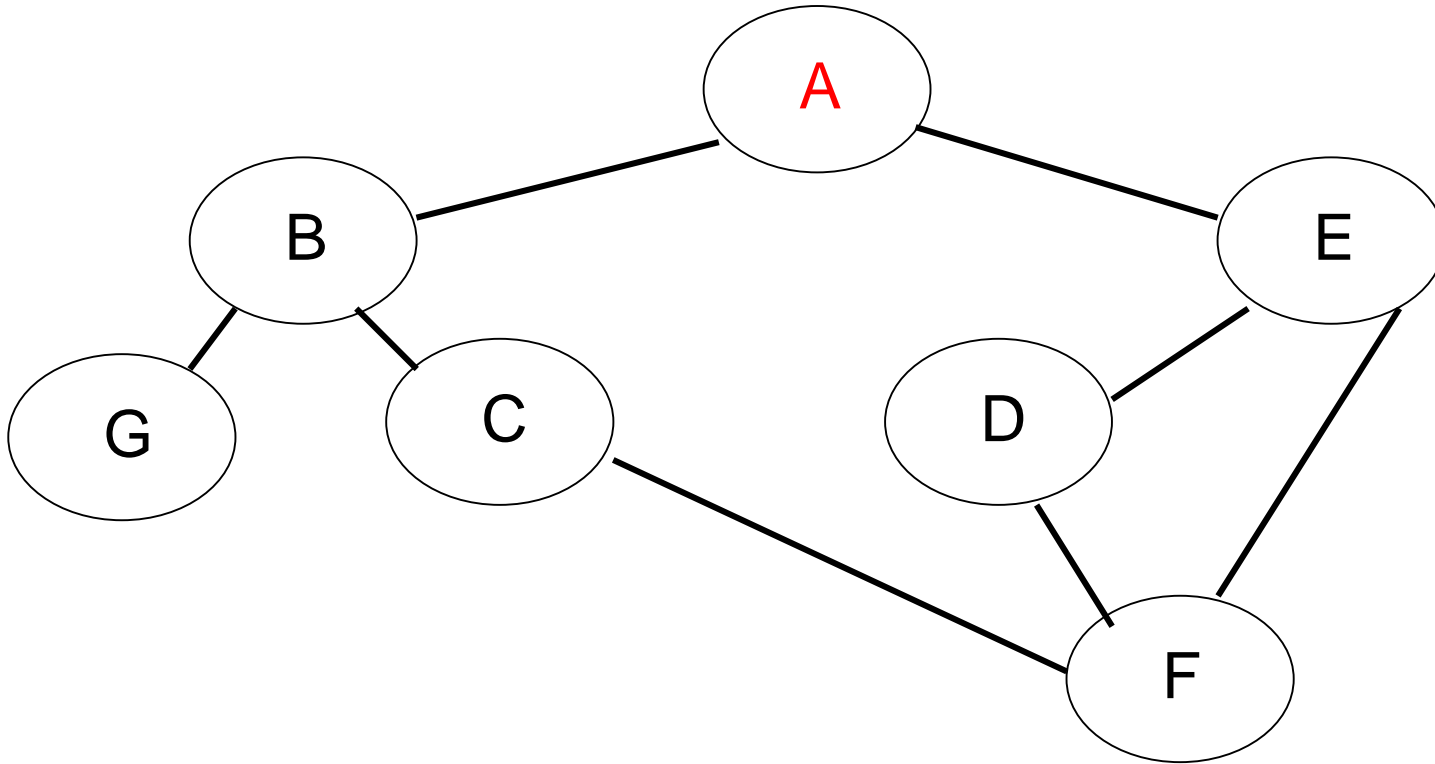
 add w to the queue

 mark w as visited

The general BFS algorithm

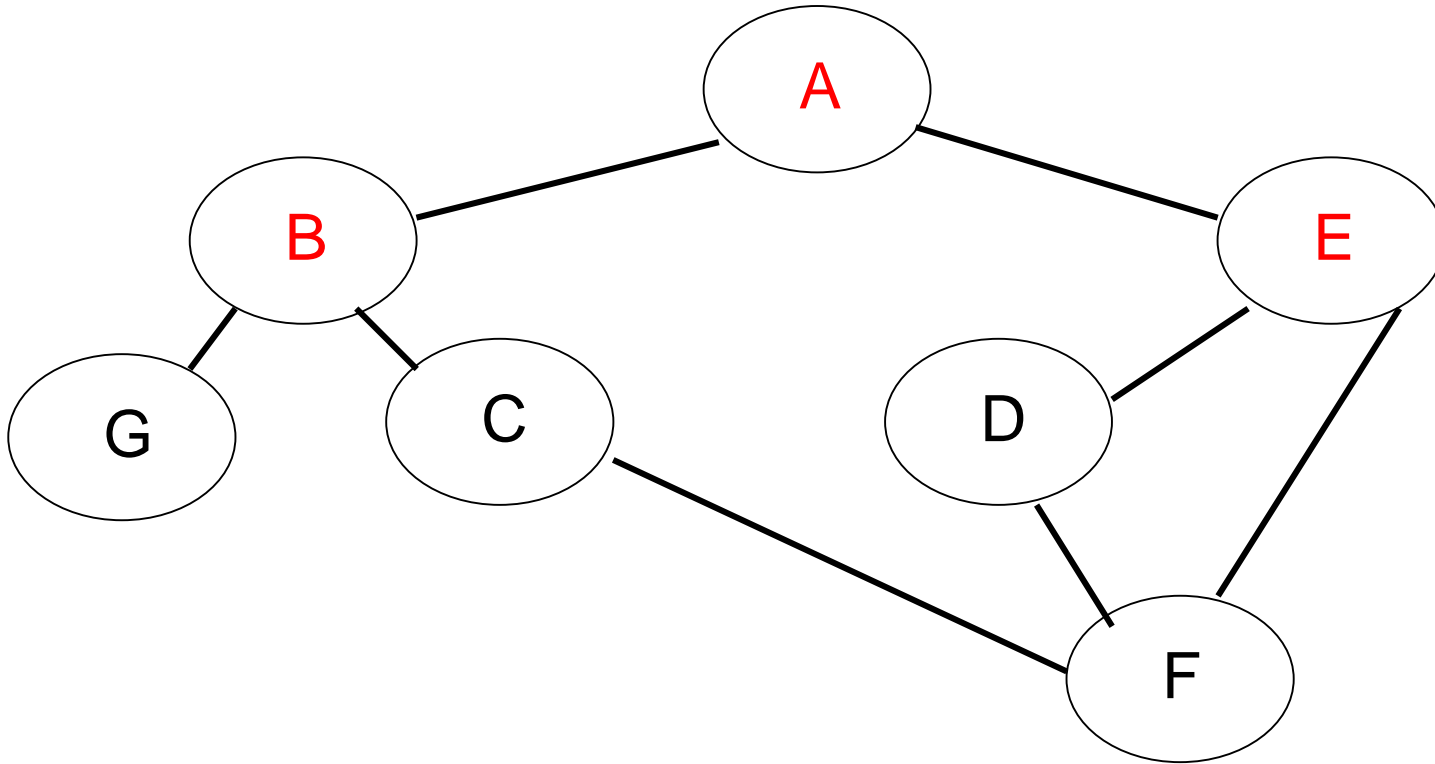
- Each vertex can be in one of three states:
 1. **Unmarked and not on queue:** Not reached yet
 2. **Marked and on queue:** Known, but adjacent vertices not visited yet (possibly)
 3. **Marked and off queue:** Known, all adjacent vertices on queue or done with
- The algorithm moves vertices between these states

Queue: A



Start with A. Mark it.

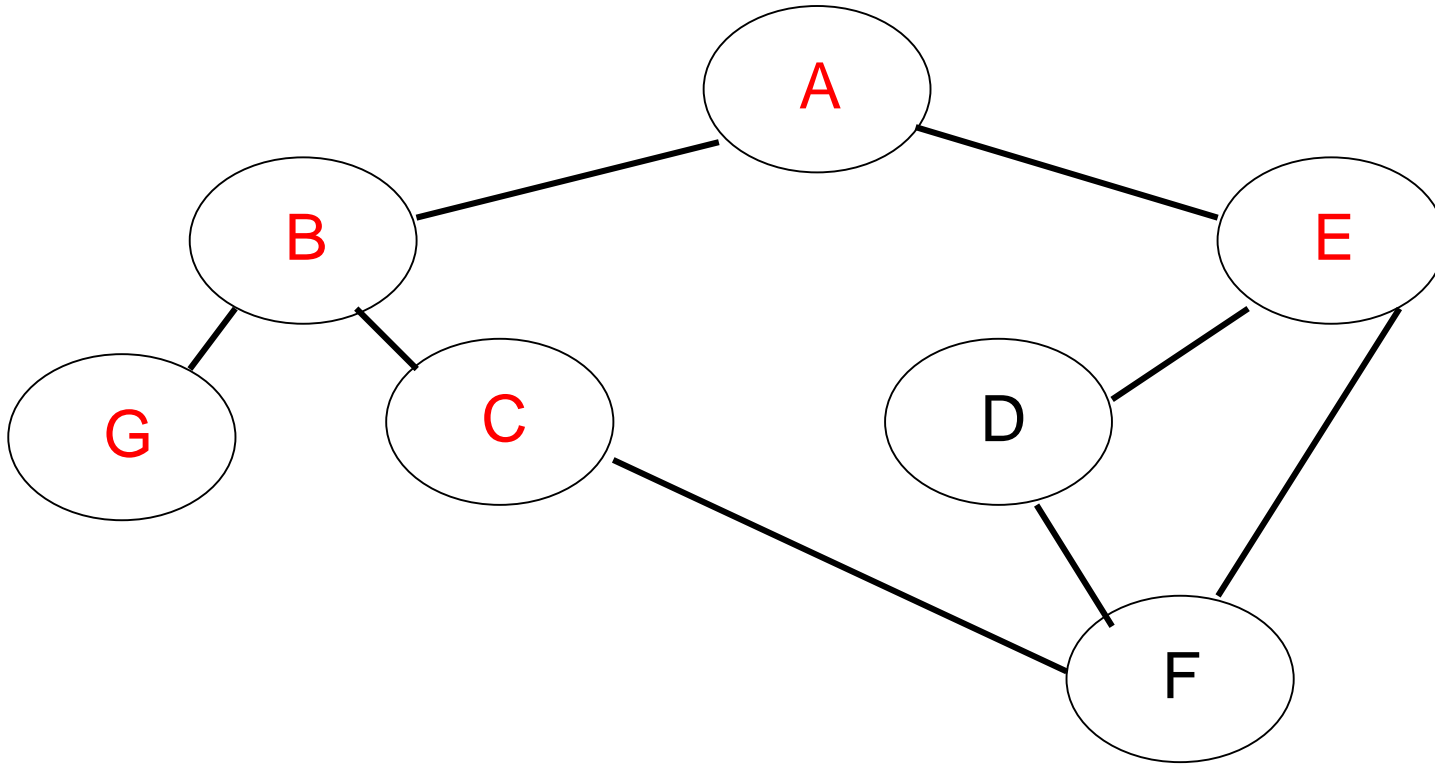
Queue: A B E



Expand A's adjacent vertices.

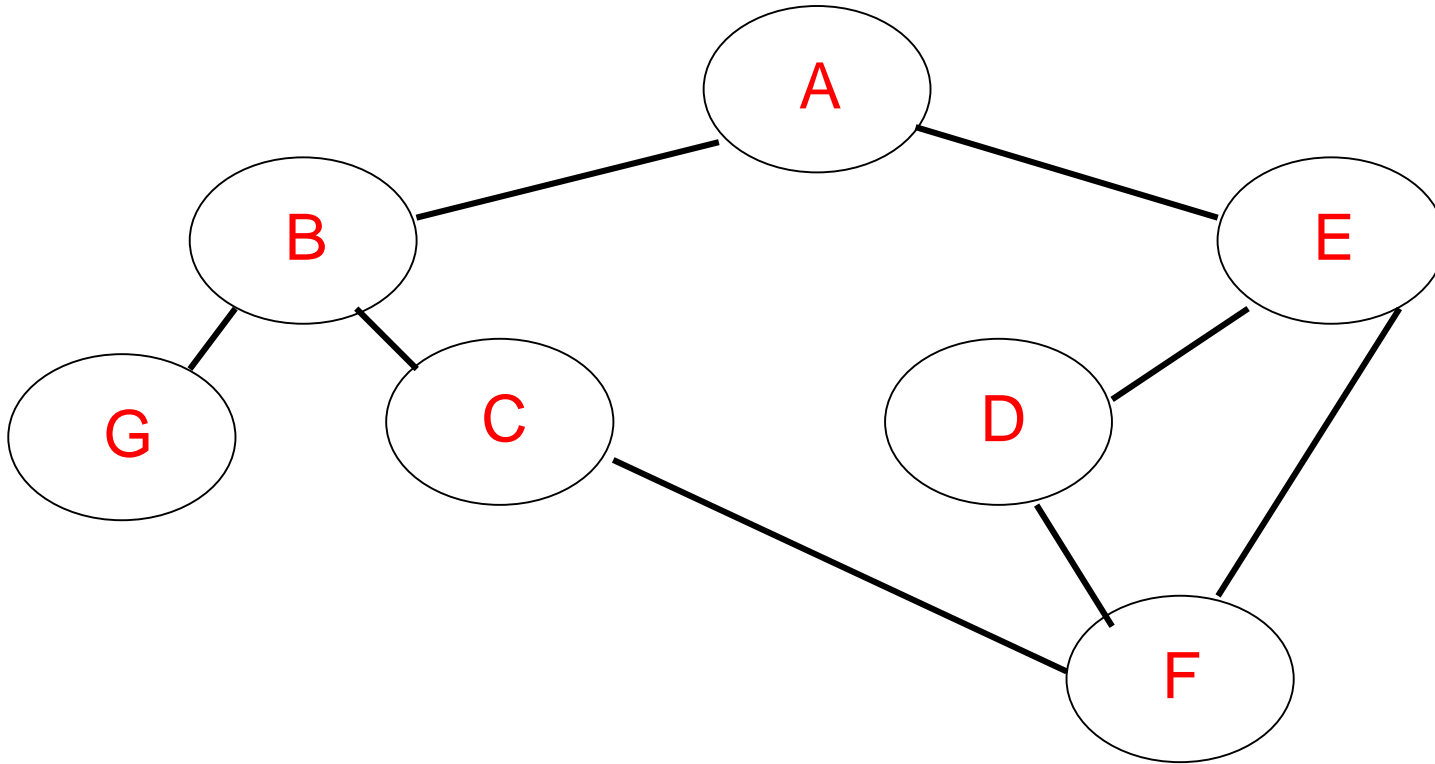
Mark them and put them in queue.

Queue: A B E C G



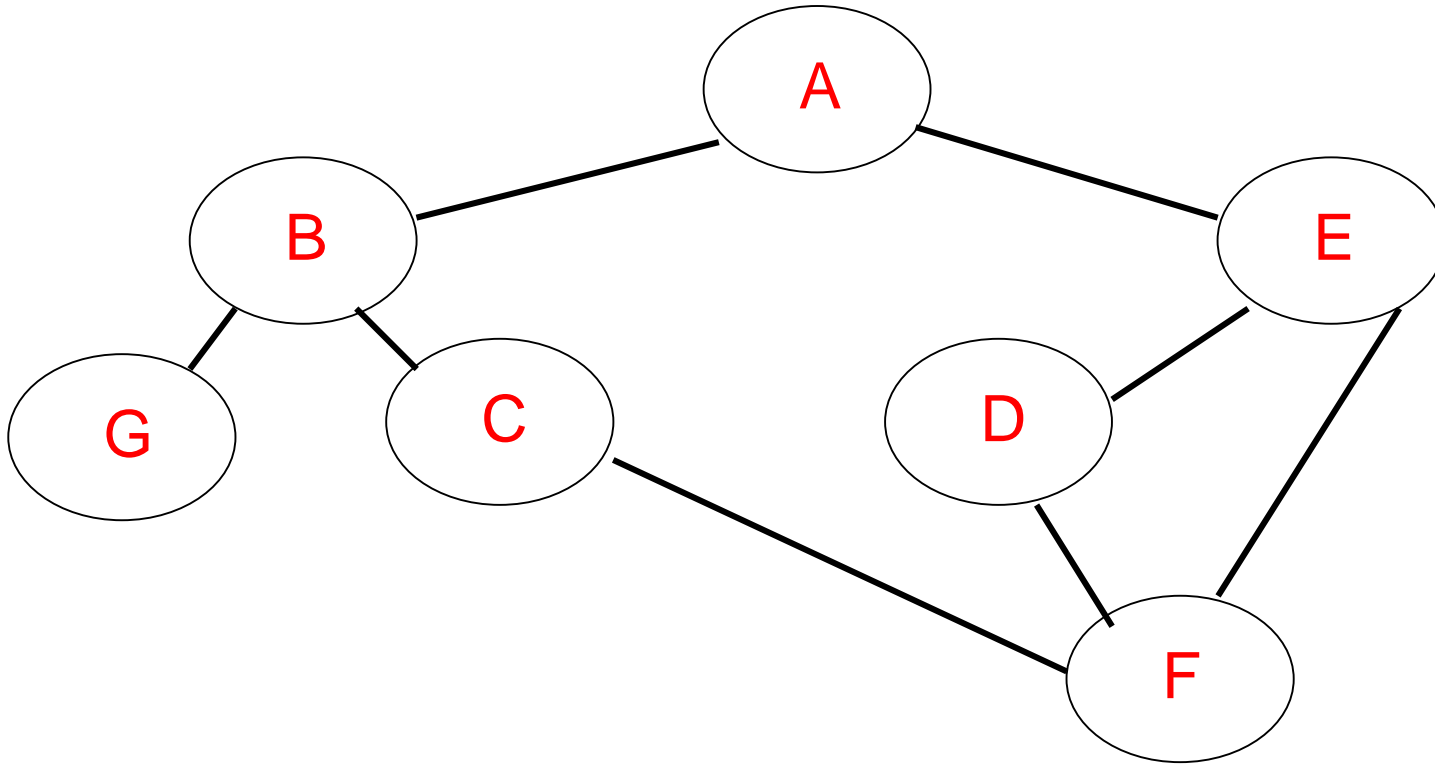
Now take B off queue, and queue its neighbors.

Queue: A B E C G D F



Do same with E.

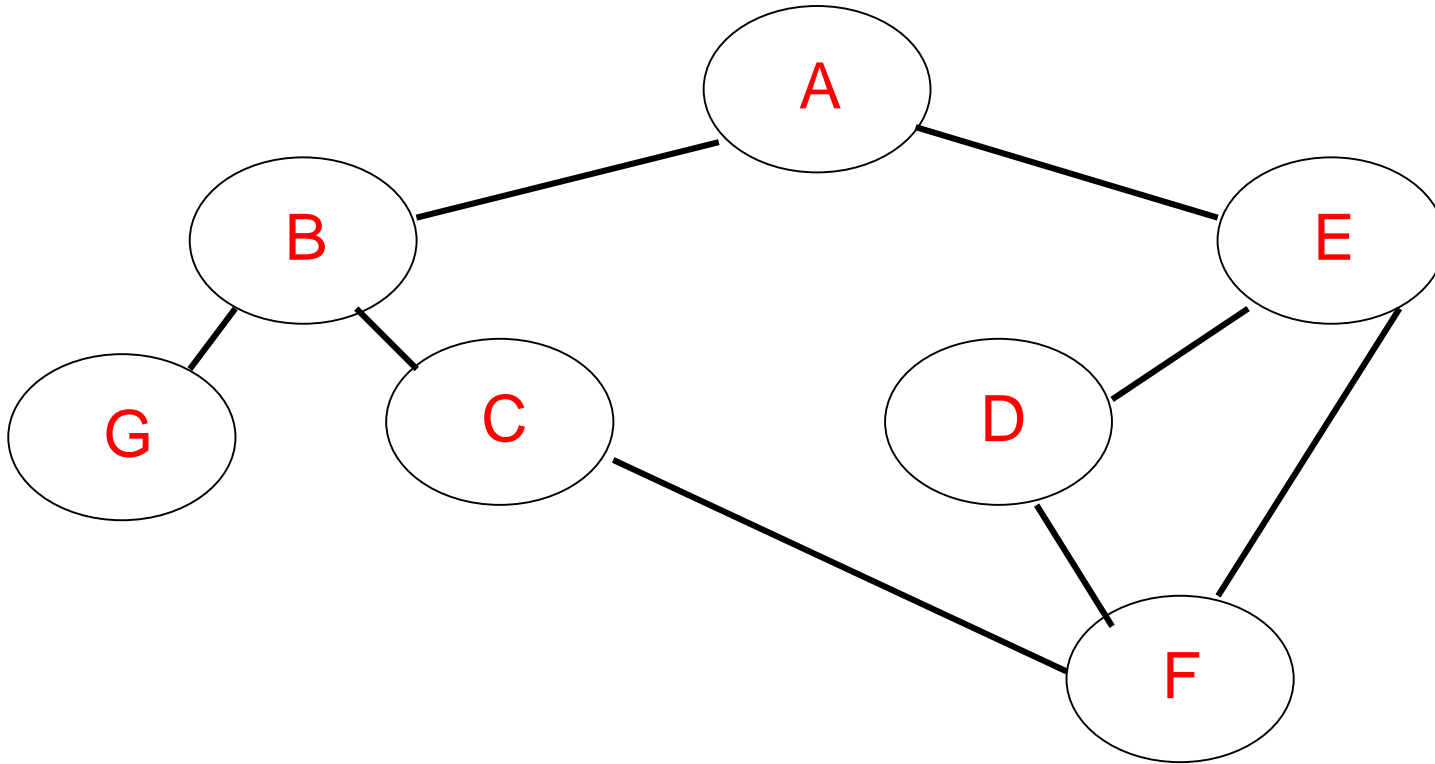
Queue: **A B E C** G D F



Visit C.

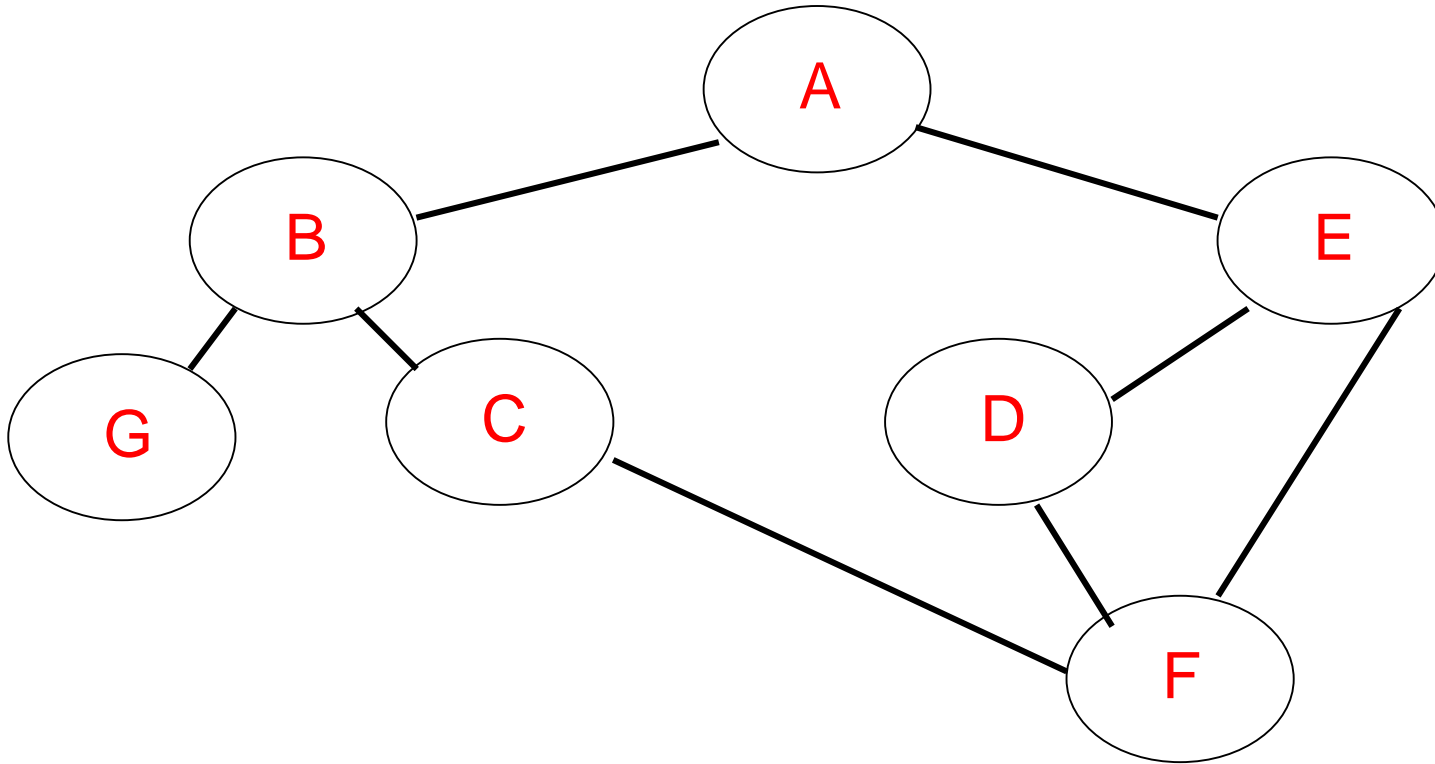
Its neighbor F is already marked, so not queued.

Queue: A B E C G D F



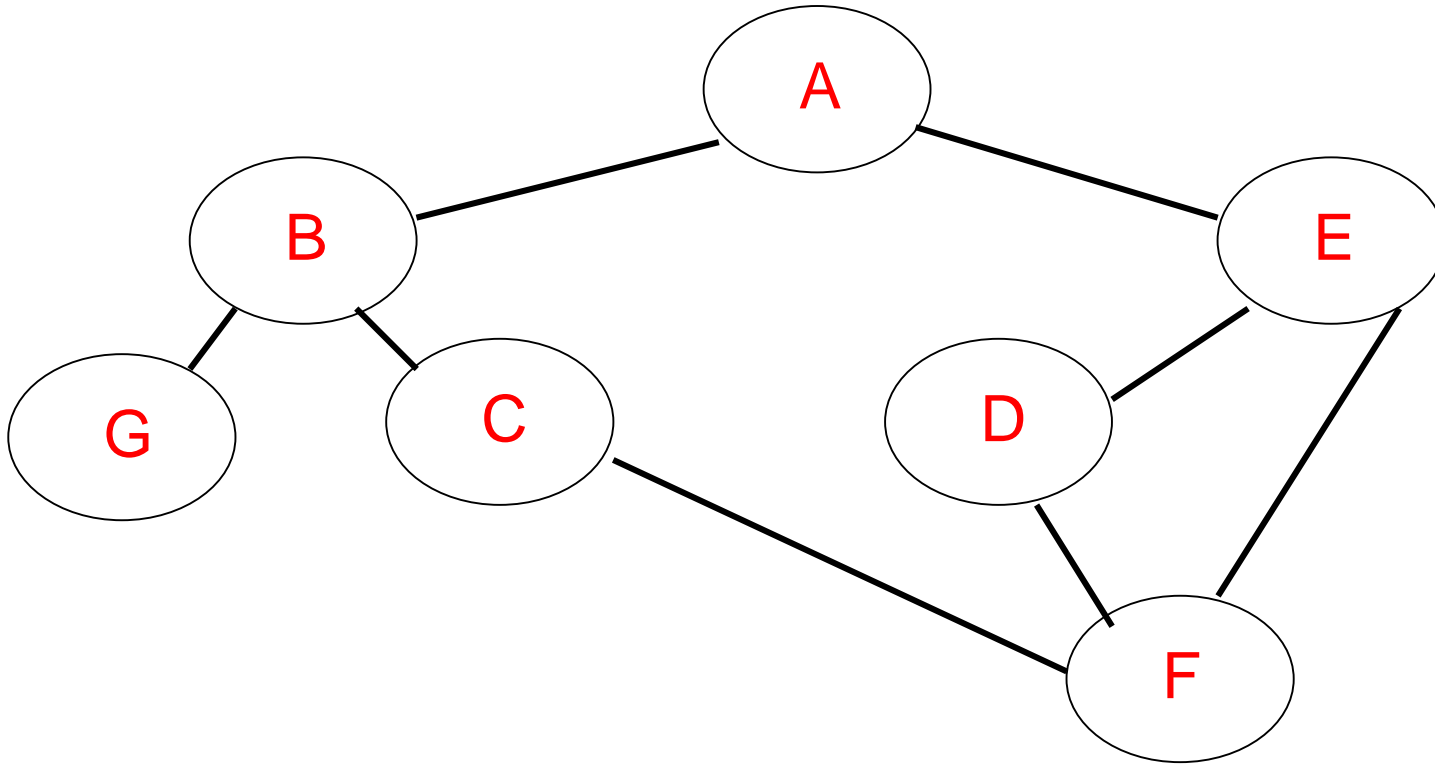
Visit G.

Queue: A B E C G D F



Visit D. F, E marked so not queued.

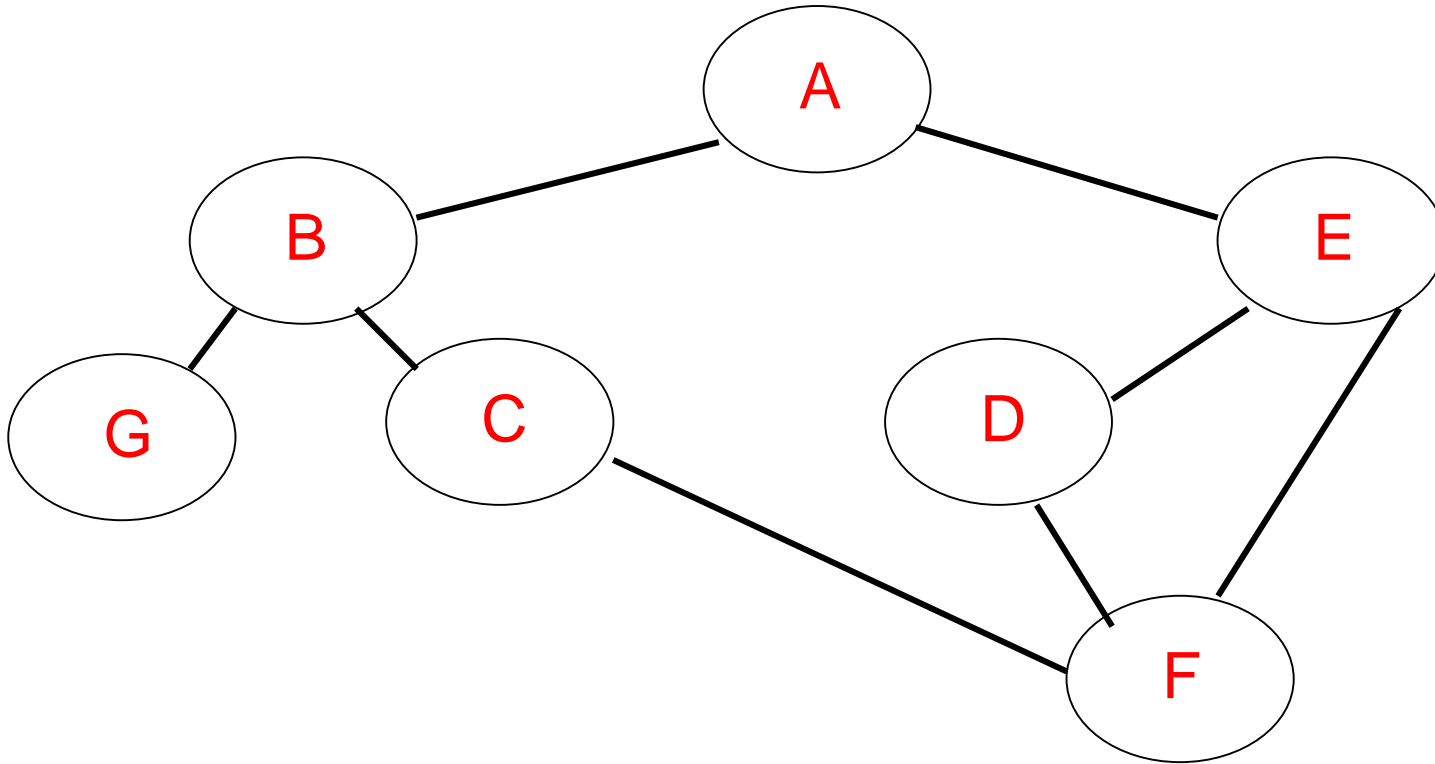
Queue: A B E C G D F



Visit F.

E, D, C marked, so not queued again.

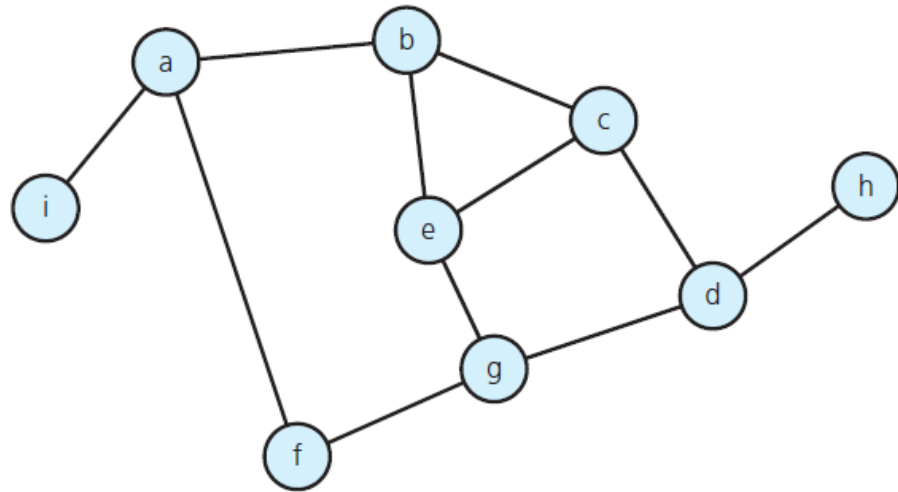
Queue: A B E C G D F



Done. We have explored the graph in order:

A B E C G D F.

Exercise: show the BFS traversal, beginning at vertex a



Node visited

a

b

f

i

c

e

g

d

h

Queue (front to back)

a

(empty)

b

b f

b f i

f i

f i c

f i c e

i c e

i c e g

c e g

e g

e g d

g d

d

(empty)

h

(empty)

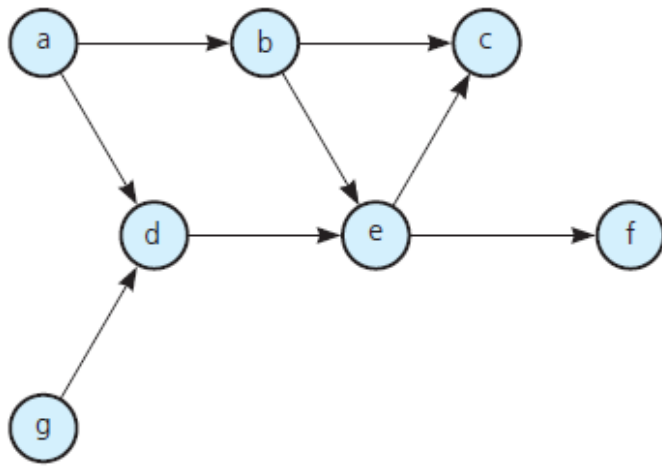
Topological Sort

- An **ordering of vertices** in a directed acyclic graph(DAG), such that for every directed edge uv from vertex u to vertex v , u comes before v in the ordering. This is not possible if the graph has a cycle.
- Any DAG has **at least one** topological ordering.
- **Examples:**
 - Course prerequisites
 - **Scheduling** a sequence of jobs or tasks based on their dependencies. The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started (for example, when washing clothes, the washing machine must finish before we put the clothes to dry). In this application, a topological ordering is just a **valid sequence for the tasks**.

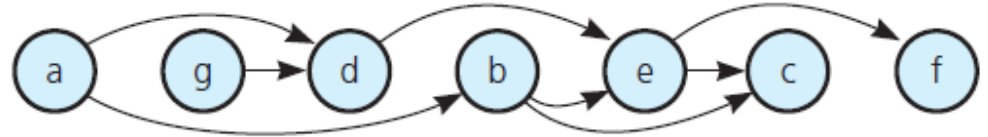
Topological Sort

- **Algorithm:**

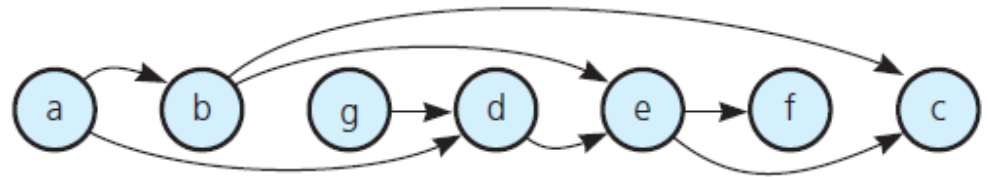
1. Find a vertex with no incoming edges
2. Process it (remove outgoing edges from it) and then remove it from the graph
3. Repeat



(a)



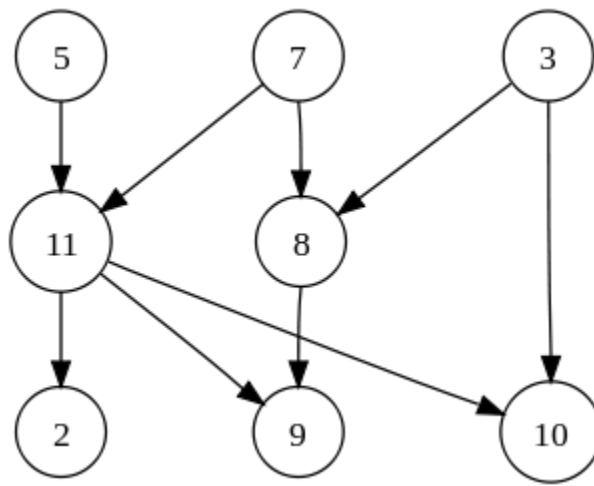
(b)



The graph above arranged according
to the topological orders

(a) a, g, d, b, e, c, f

(b) a, b, g, d, e, f, c



- The graph above has many valid topological sorts, including:
- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Topological sorting algorithm – version 1

The **indegree of vertex** v is the number of edges coming into v . Assuming that the indegree for each vertex is stored and that the graph is stored as an adjacency list, the following algorithm can be derived:

```
for( counter from 0 through the number of vertices )  
     $v$  = find a new vertex with indegree of 0  
    assign a topological number to  $v$  using the counter  
  
    for each vertex  $w$  that is adjacent to  $v$   
        decrement  $w$ 's indegree by 1  
    endfor  
endfor
```

- The finding of the vertex with indegree of 0 is accomplished by scanning the vertices looking for a vertex with indegree of 0 and no topological number. If no such vertex exists, then the graph contains a **cycle**.
- Finding the vertex with indegree of 0 has a running time of V . That call is made V times, therefore the algorithm has a running time of V^2 .
- The algorithm can be **improved** by simply storing all of the vertices with **indegree of 0 on a queue**. The algorithm now has a running time **$E+V$** since the for loop is executed at most once per edge, the queue operations are executed at most once per vertex, and the initialization is at most the size of the graph.
- The new algorithm resembles:

Topological sorting algorithm – version 2

for each vertex v

if v has indegree of 0

push v onto a **queue**

Initialize a counter to 0

while the queue is not empty

$v = \text{queue.top}()$

$\text{queue.pop}()$

increment the counter

assign the counter as the topological number for v

for each vertex w that is adjacent to v

decrement w 's indegree by 1

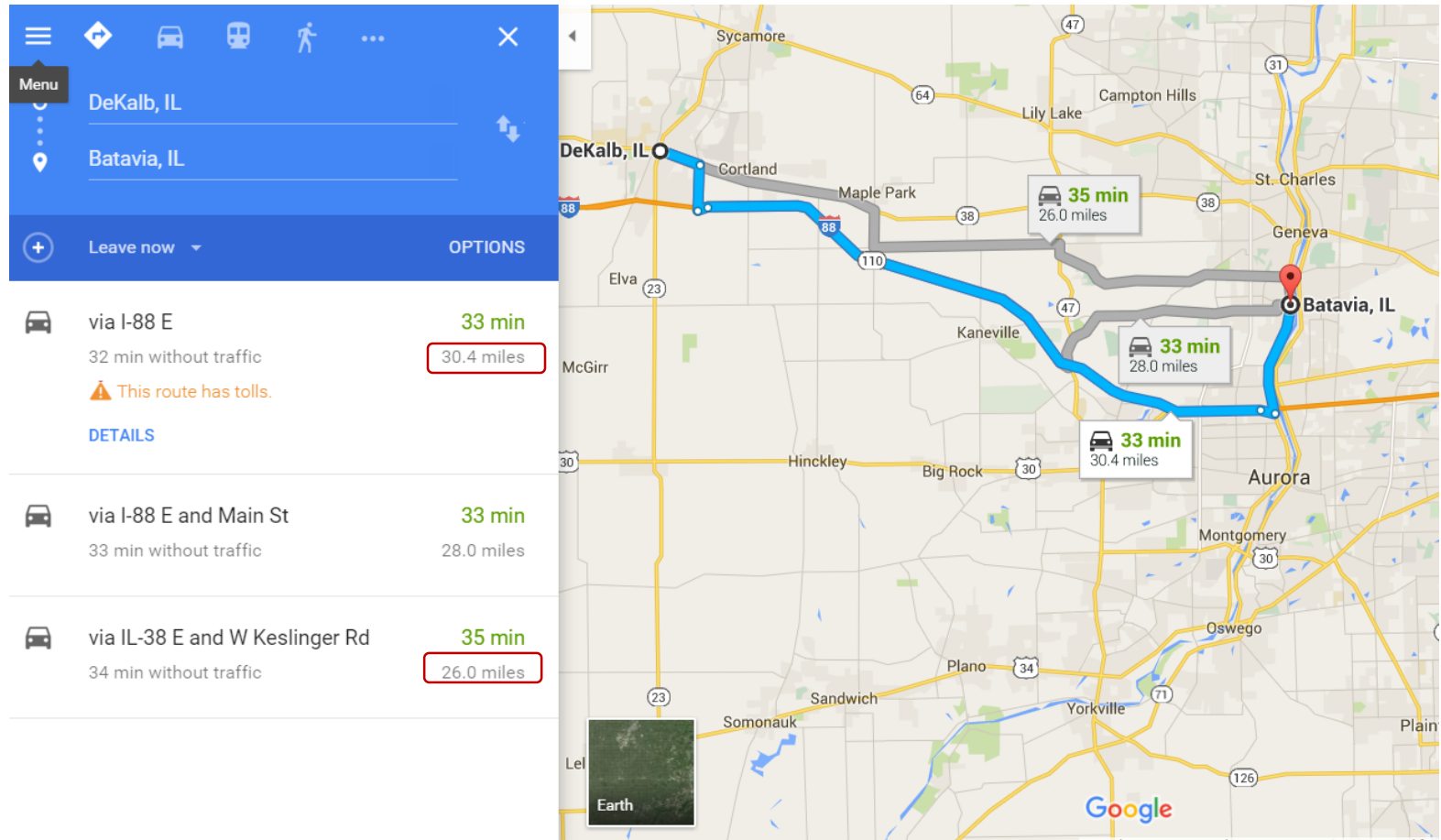
if w 's indegree is 0

push w onto the queue

Topological sorting algorithm – other versions

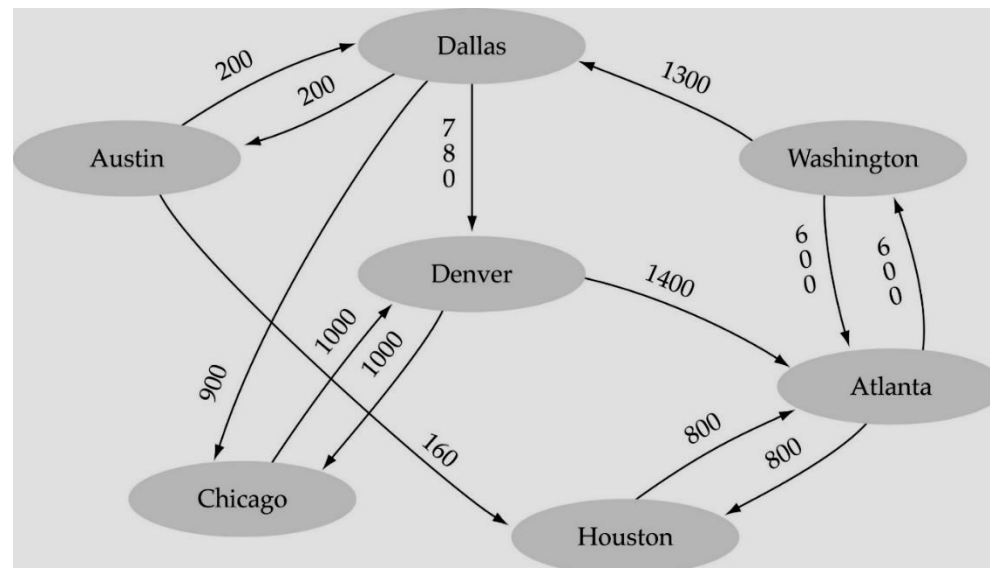
- Indegree and stack
- <https://www.cs.usfca.edu/~galles/visualization/TopoSortIndegree.html>
- Using DFS (when outdegree = 0)
- <https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html>

What is the best route from Dekalb to Batavia?



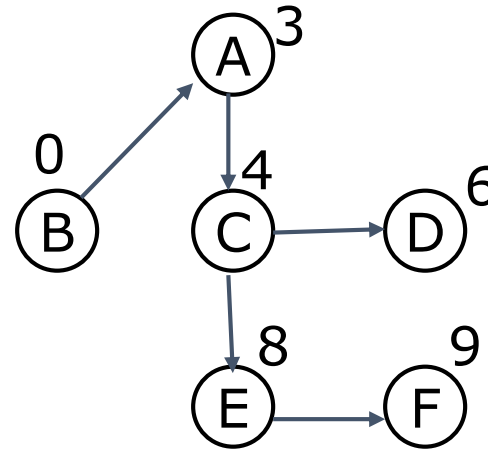
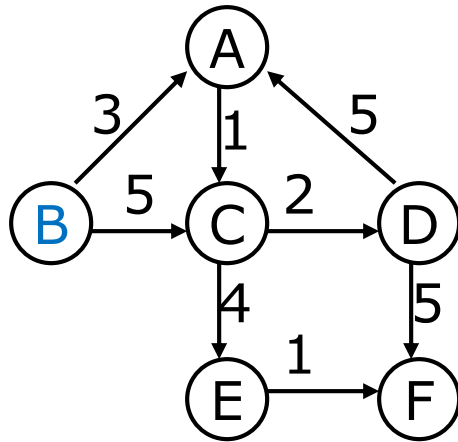
Shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- **Shortest path**: the path whose total weight (i.e., sum of edge weights) is minimum.
- **Dijkstra's Algorithm** finds the shortest path from a given node to **all** other reachable nodes in a weighted graph with **nonnegative** edge weights.
- **Examples:**
 - Austin->Houston->Atlanta->Washington: 1560 miles
 - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles



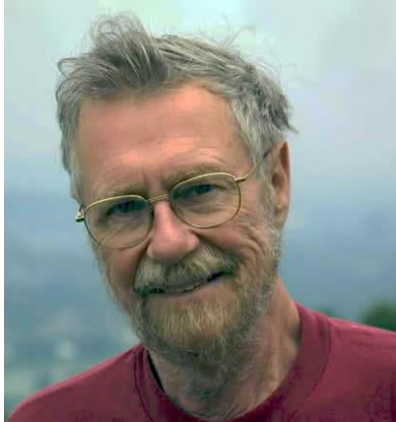
Dijkstra's algorithm

- Dijkstra's algorithm builds up a **tree**.
- It is a **greedy algorithm** that solves problems in stages by doing what appears to be best at each stage.
- For example, in the following graph, we want to find shortest paths from node **B**



- **Edge values** in the graph are **weights**
- **Node values** in the tree are **total weights**

Edsger W. Dijkstra



- “If debugging is the process of removing software bugs, then programming must be the process of putting them in” 😂
- “Program testing can be used to show the presence of bugs, but never to show their absence!”
- “Simplicity is prerequisite for reliability”
- Received the 1972 A. M. **Turing Award**, widely considered the most prestigious award in computer science.
- Made a strong case against use of the GOTO statement in programming languages and helped lead to its deprecation.
- Known for his many essays on programming.

Dijkstra's shortest-path algorithm

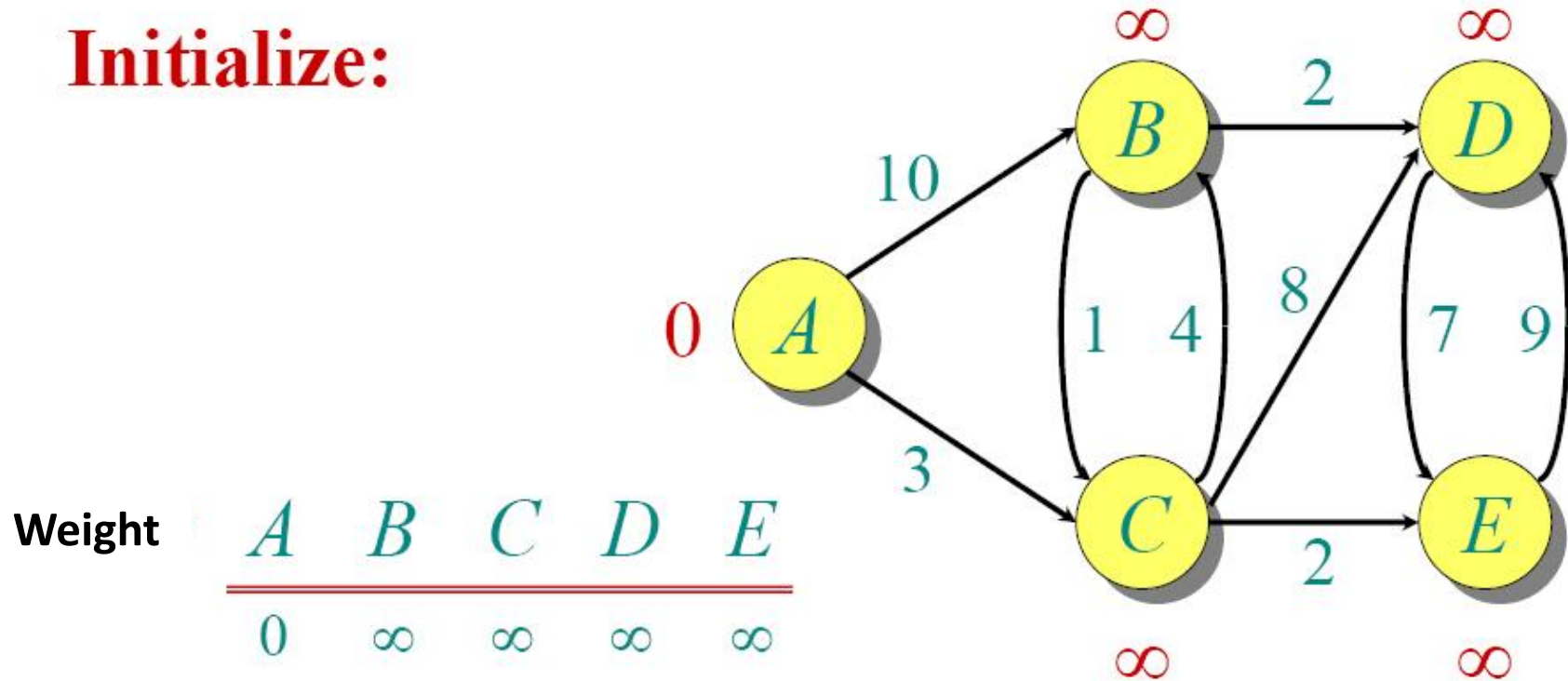
- The graph needs to be **initialized** with high values at the beginning.
- The algorithm stores all weights from the source vertex to all other vertices in the graph in an array called **weight**.
- It then step by step attempts to visit the vertex with minimum weight using the vertices stored in a set **visited**, and adding the newly visited vertex to it.

Dijkstra's shortest-path algorithm

```
shortestPath(in theGraph, in weight:WeightArray) {  
  // Finds the minimum-cost paths between an origin vertex (vertex 0) and all other  
  // vertices in a weighted directed graph.  
  visited = vertex 0  
  n = number of vertices in the Graph;  
  // Step 1  
  for (x=0 through n-1)  
    weight[v] = matrix[v][x];  
  // Steps 2 through n  
  for (step=2 through n) {  
    Find the smallest weight[v] such that v is not visited  
    Add v to visited;  
  
    for (each neighbor u of v not in visited)  
      if (weight[u] > weight[v]+matrix[v][u])  
        weight[u] = weight[v]+matrix[v][u];  
  }  
}
```

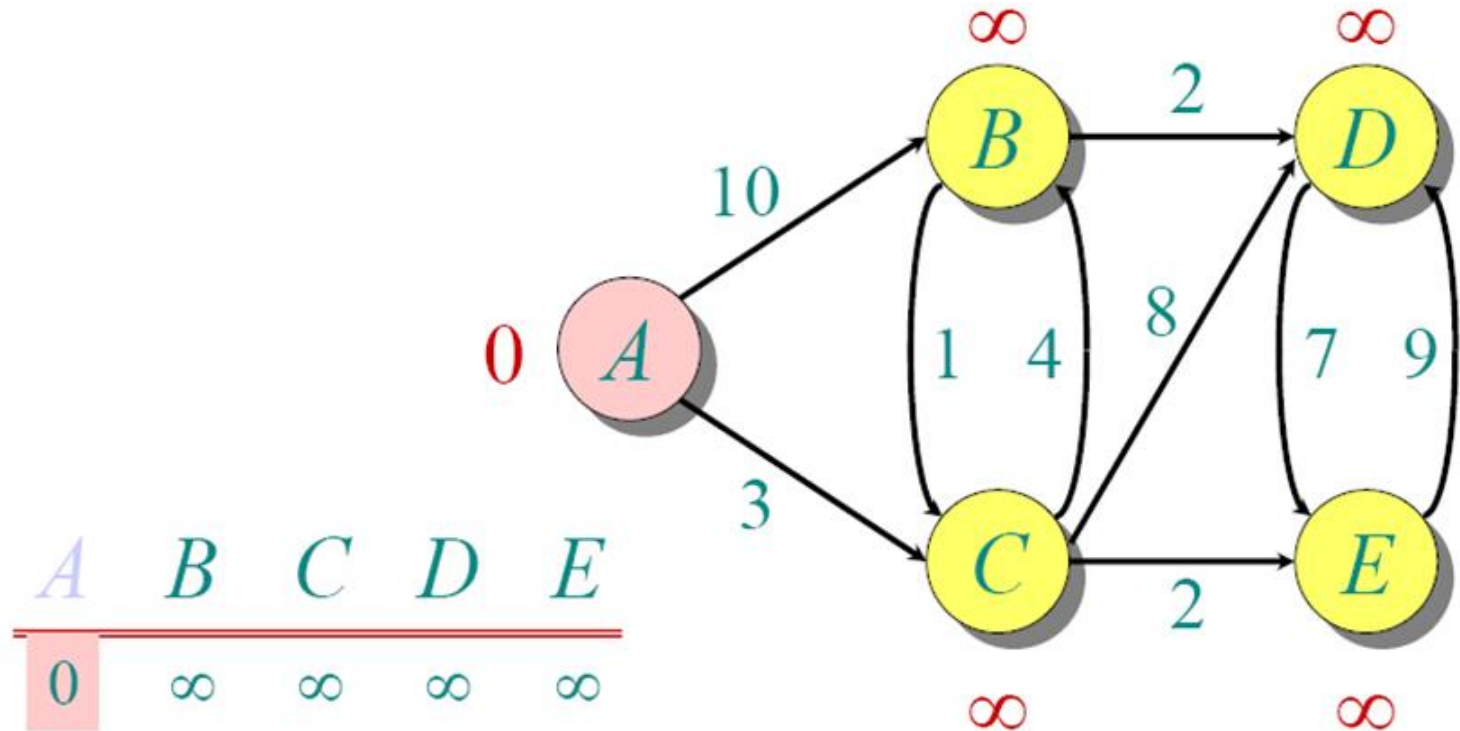
Dijkstra Animated Example

Initialize:

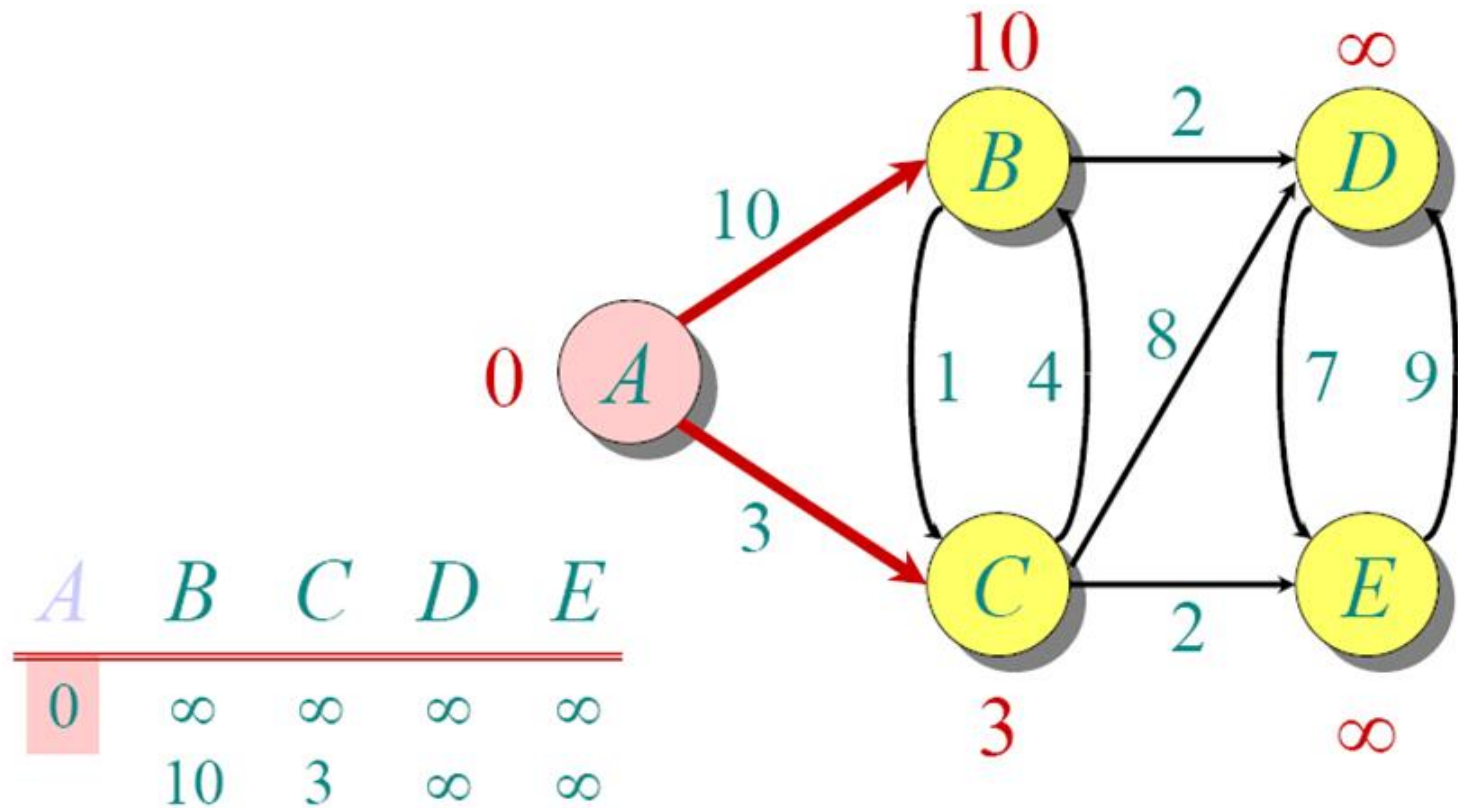


Visited *S*: {}

Dijkstra Animated Example

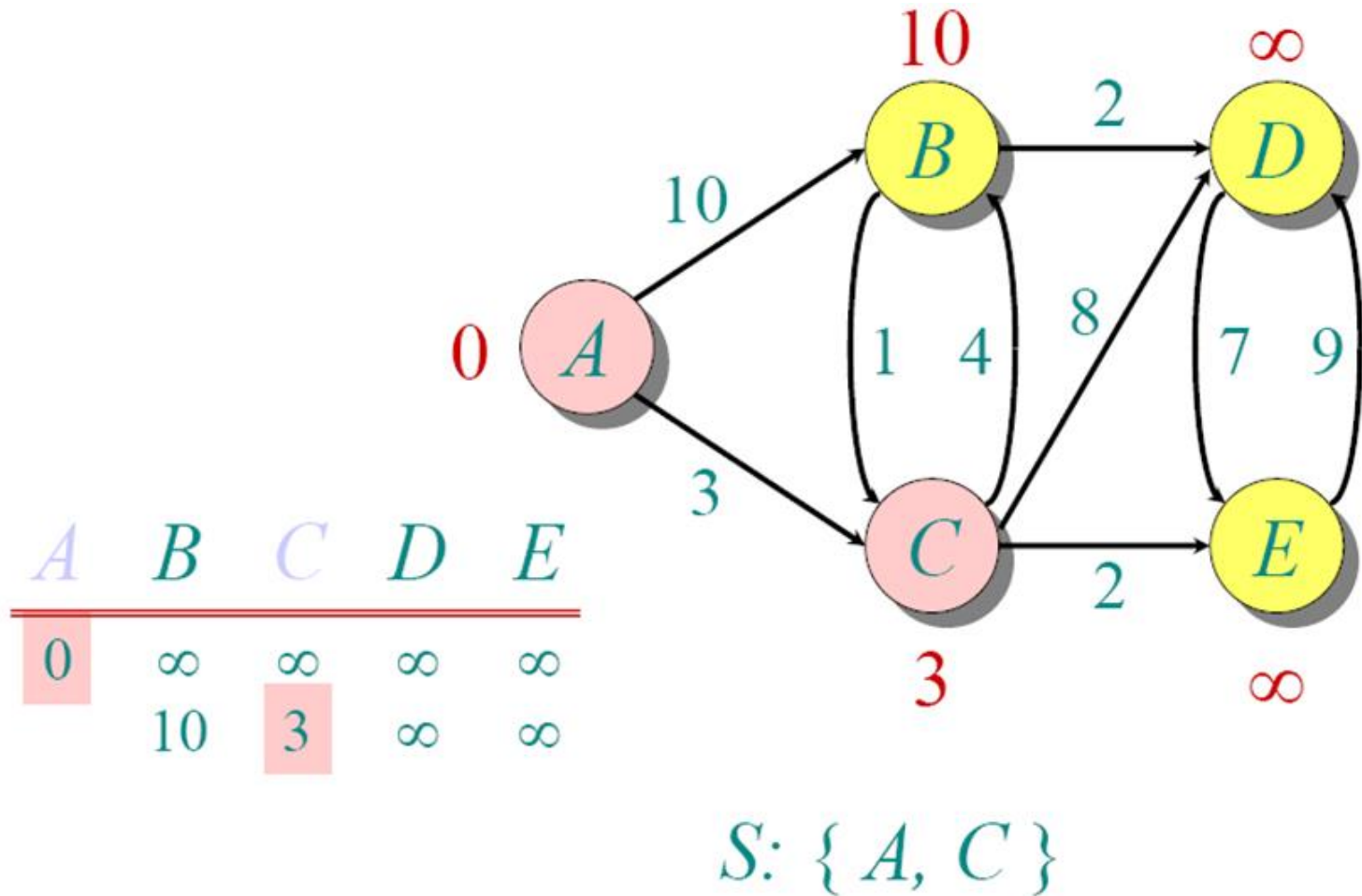


Dijkstra Animated Example

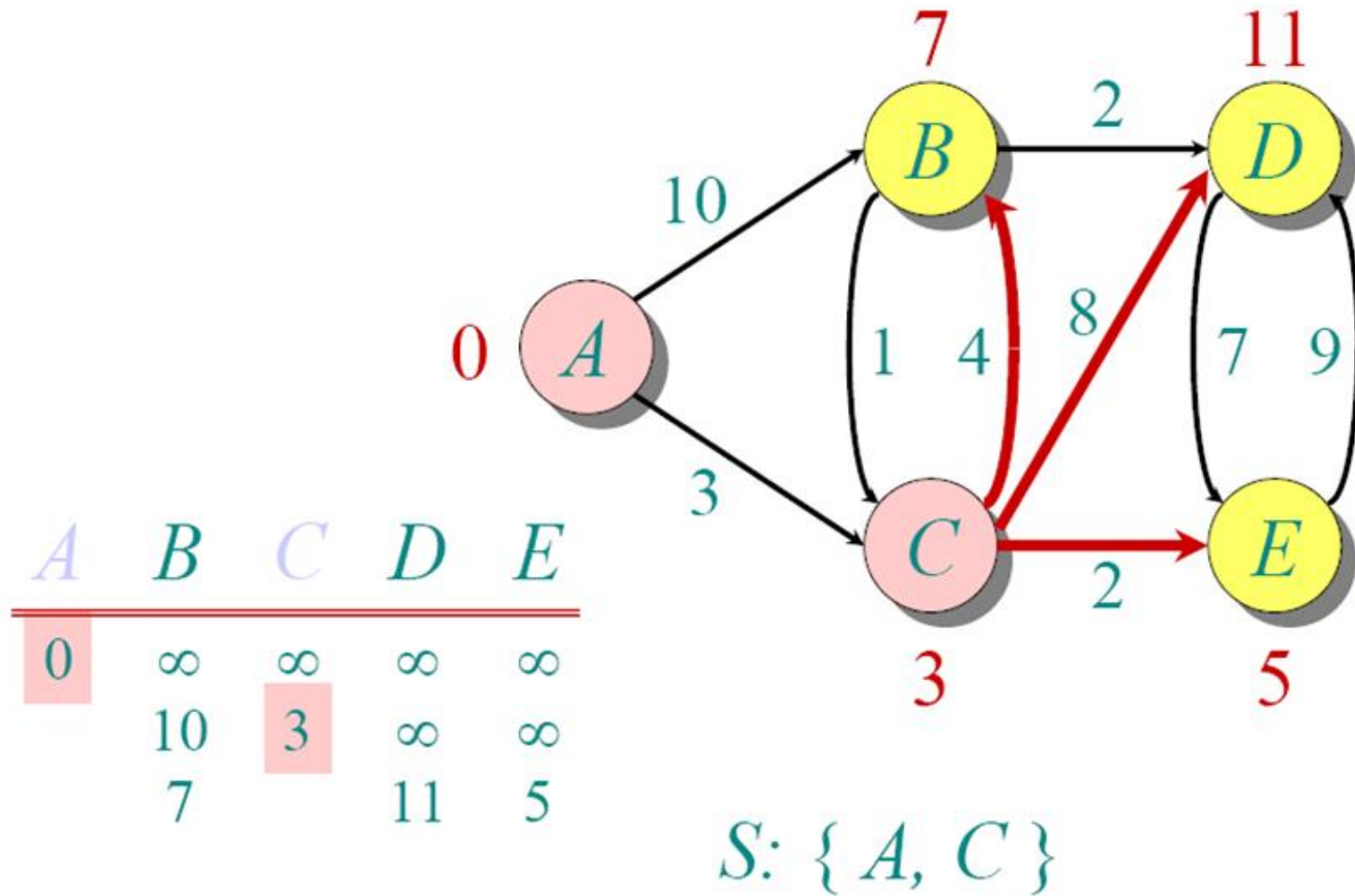


$S: \{A\}$

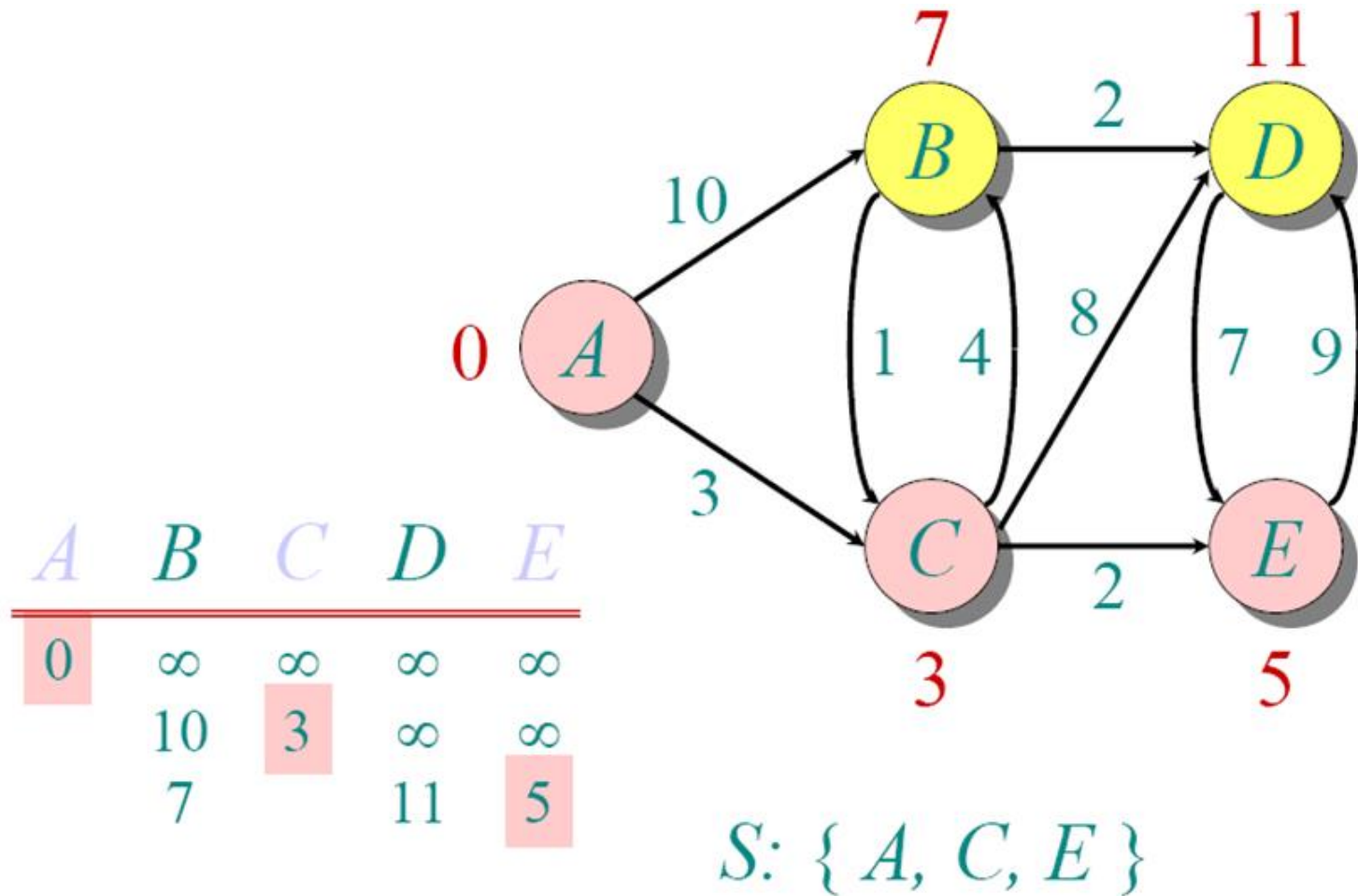
Dijkstra Animated Example



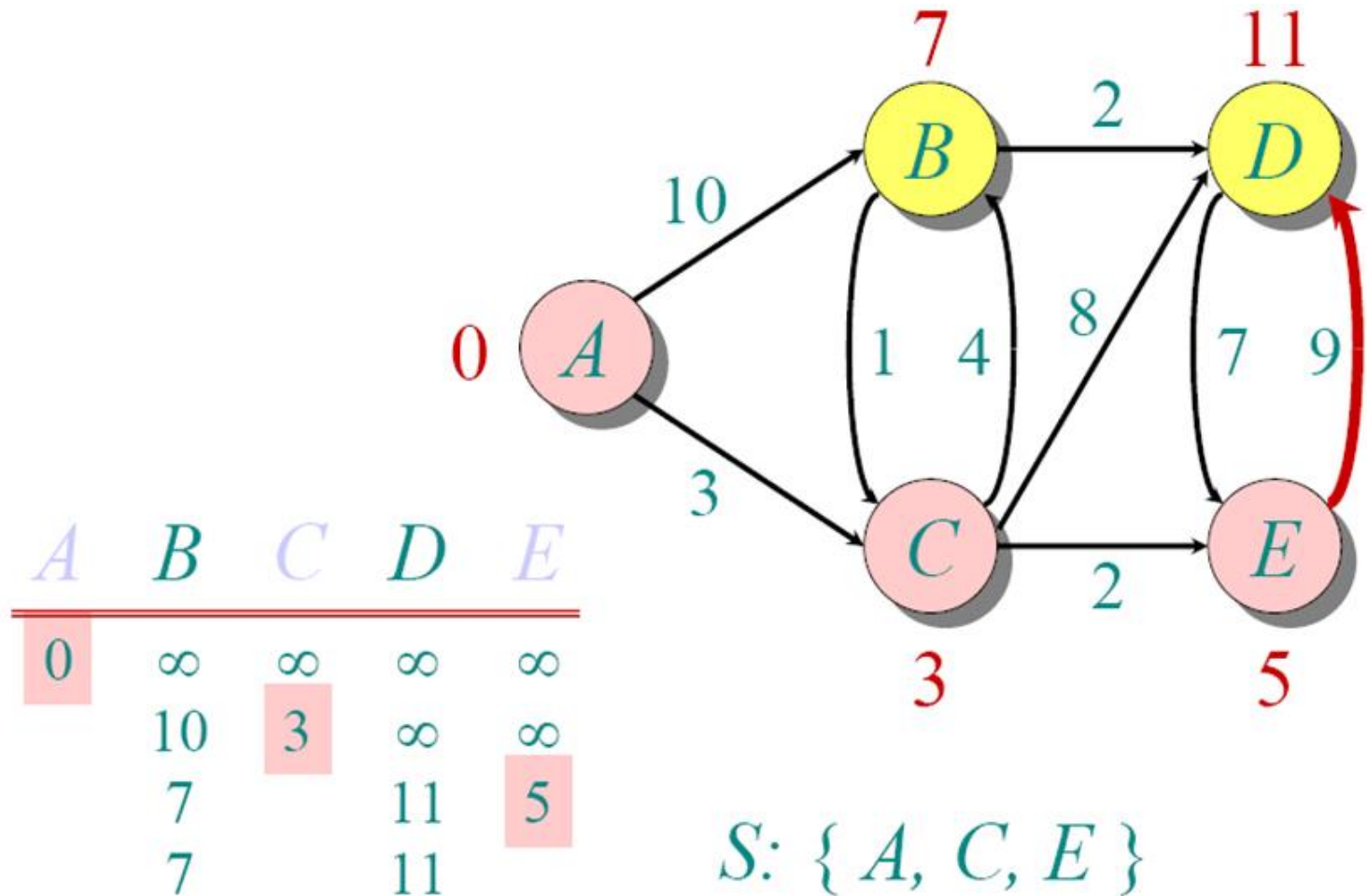
Dijkstra Animated Example



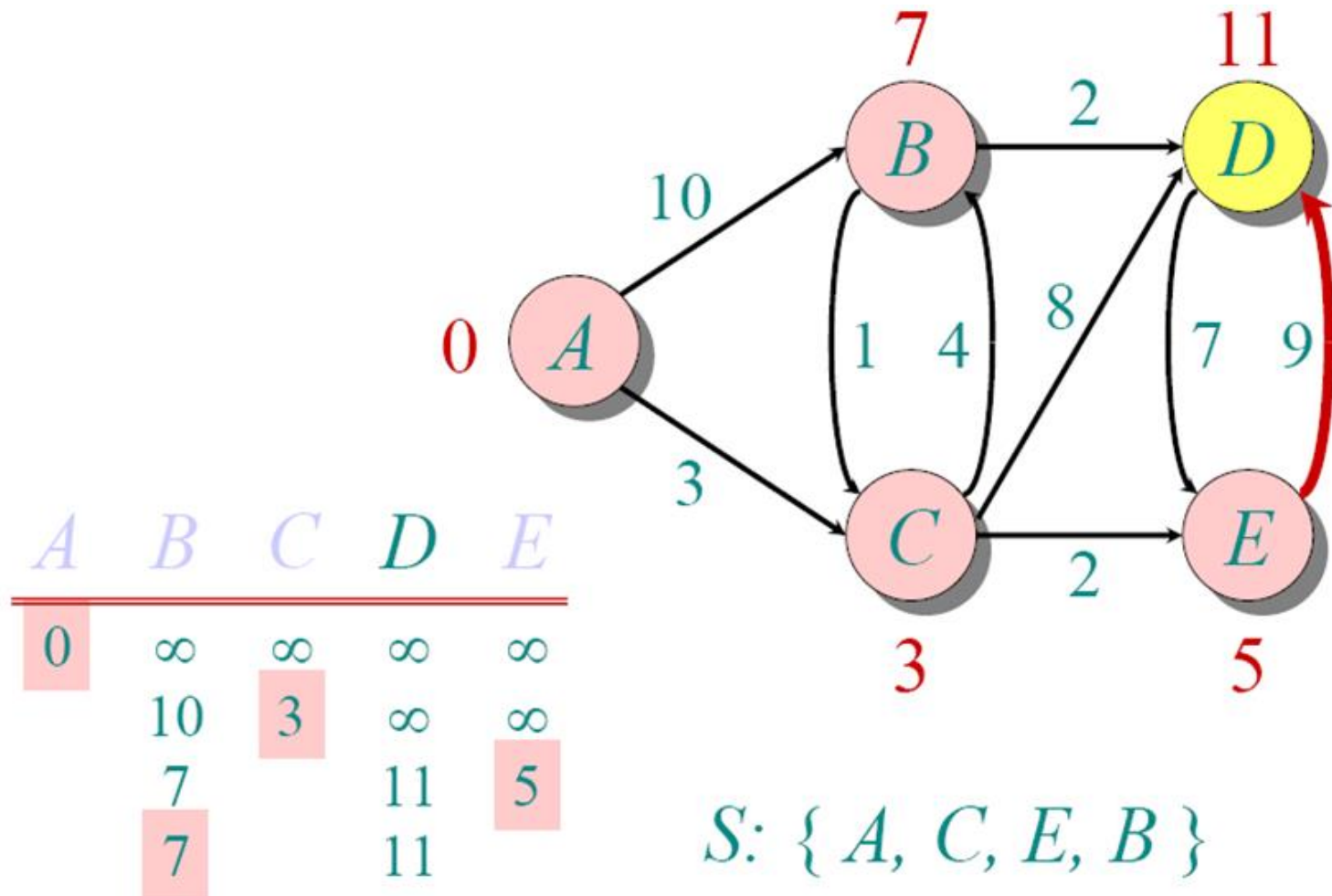
Dijkstra Animated Example



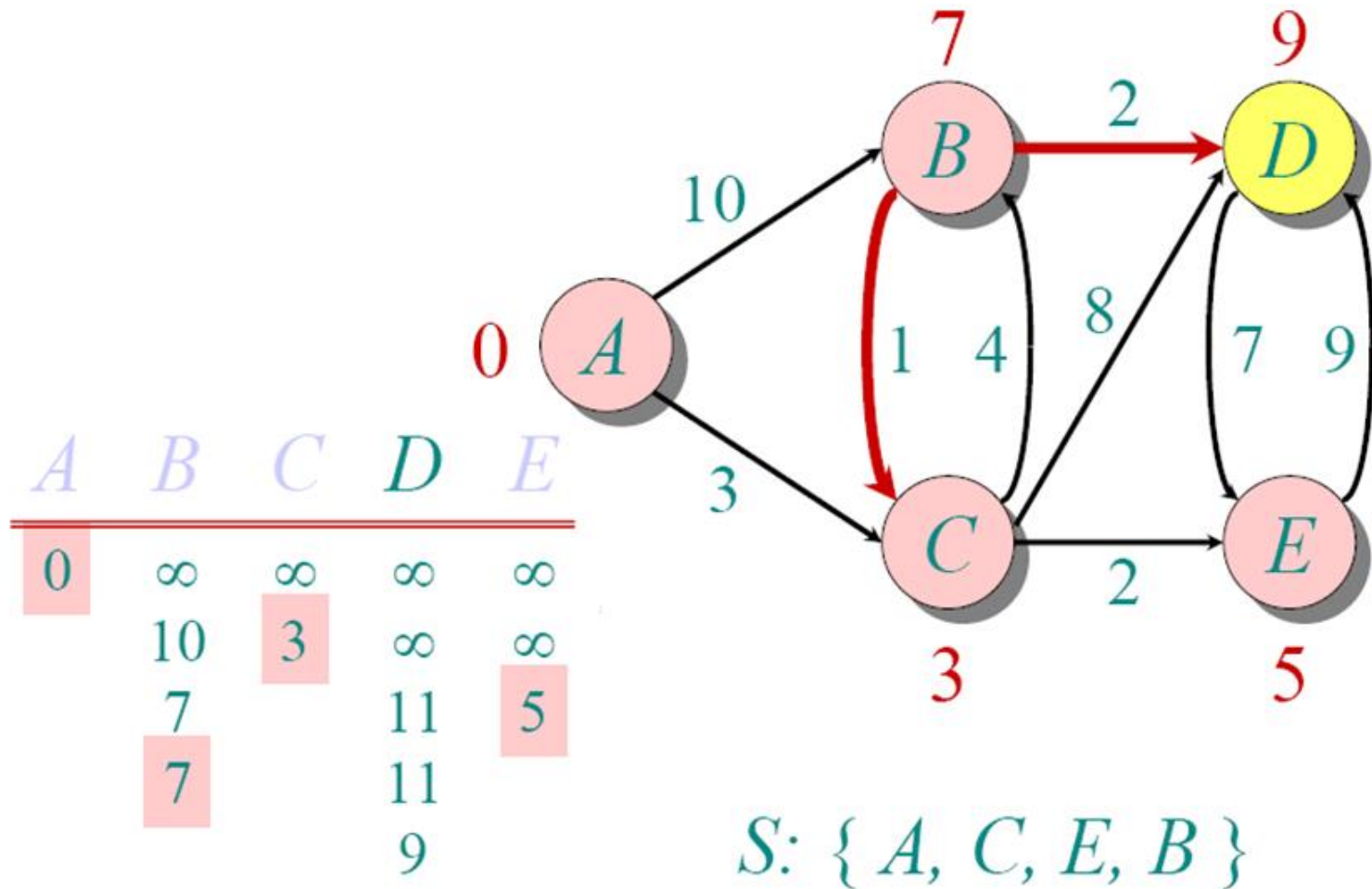
Dijkstra Animated Example



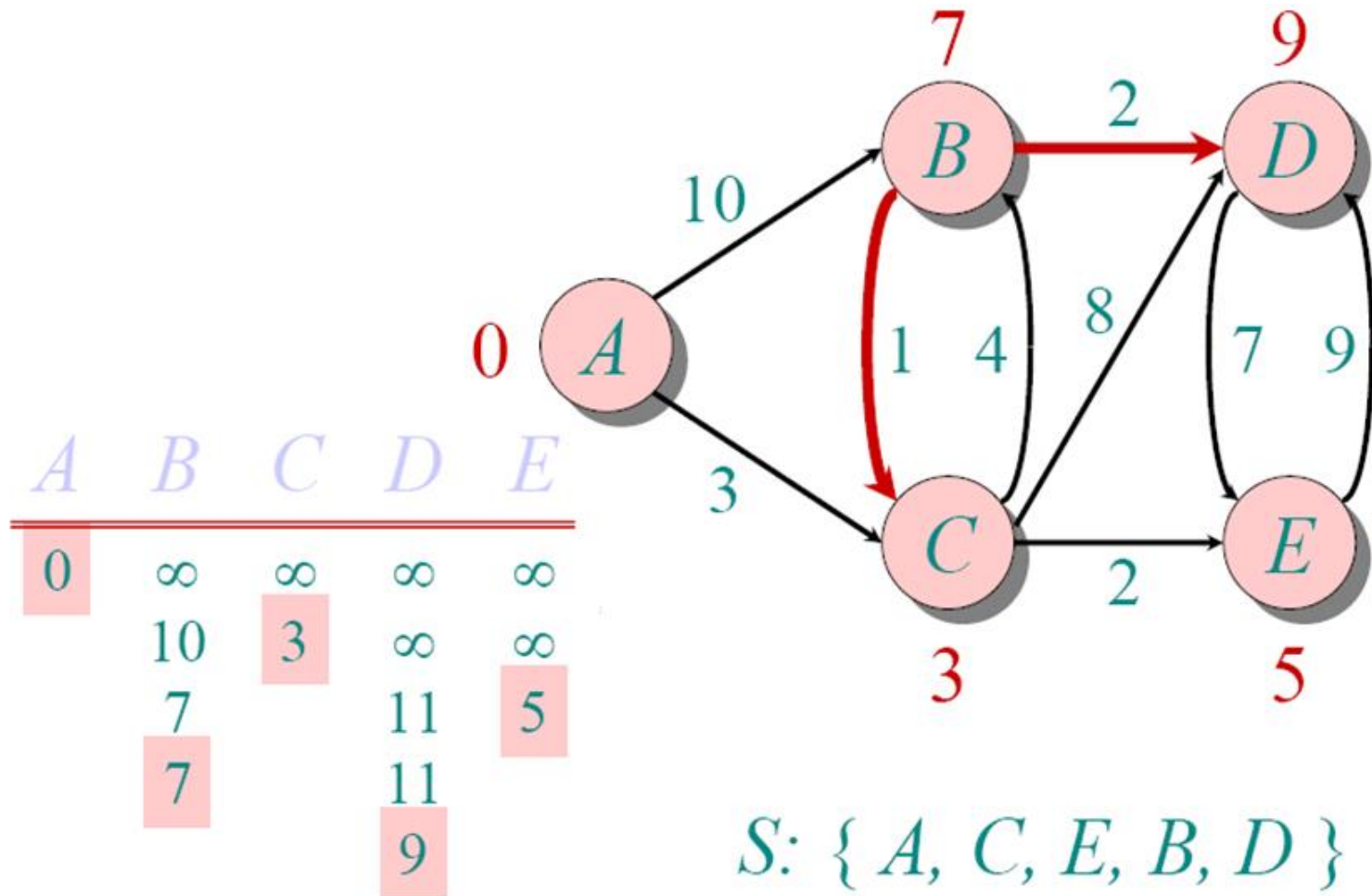
Dijkstra Animated Example



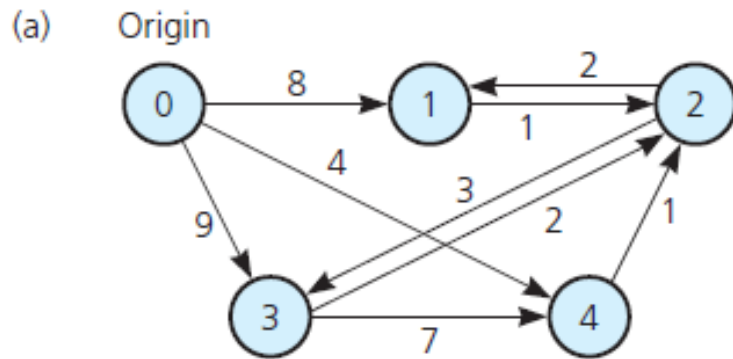
Dijkstra Animated Example



Dijkstra Animated Example



Find the shortest path from 0 to all other vertices



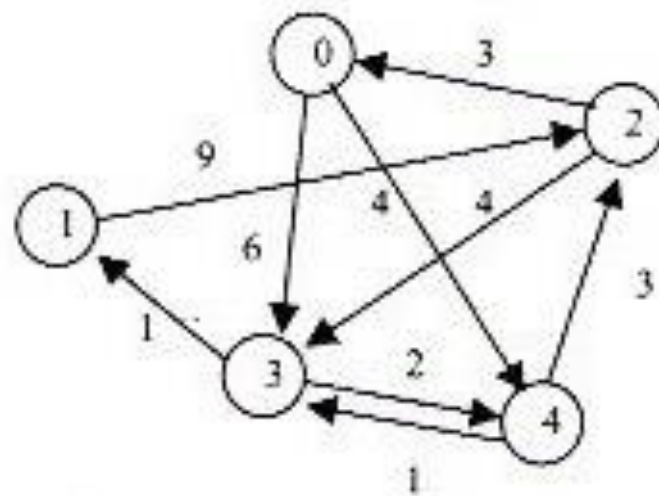
(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

<u>Step</u>	<u>v</u>	Visited	weight				
		<u>vertexSet</u>	<u>[0]</u>	<u>[1]</u>	<u>[2]</u>	<u>[3]</u>	<u>[4]</u>
1	—	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Find the shortest path from 0 to all other vertices

	0	1	2	3	4
0	∞	∞	∞	6	4
1	∞	∞	9	∞	∞
2	3	∞	∞	4	∞
3	∞	1	∞	∞	2
4	∞	∞	3	1	∞

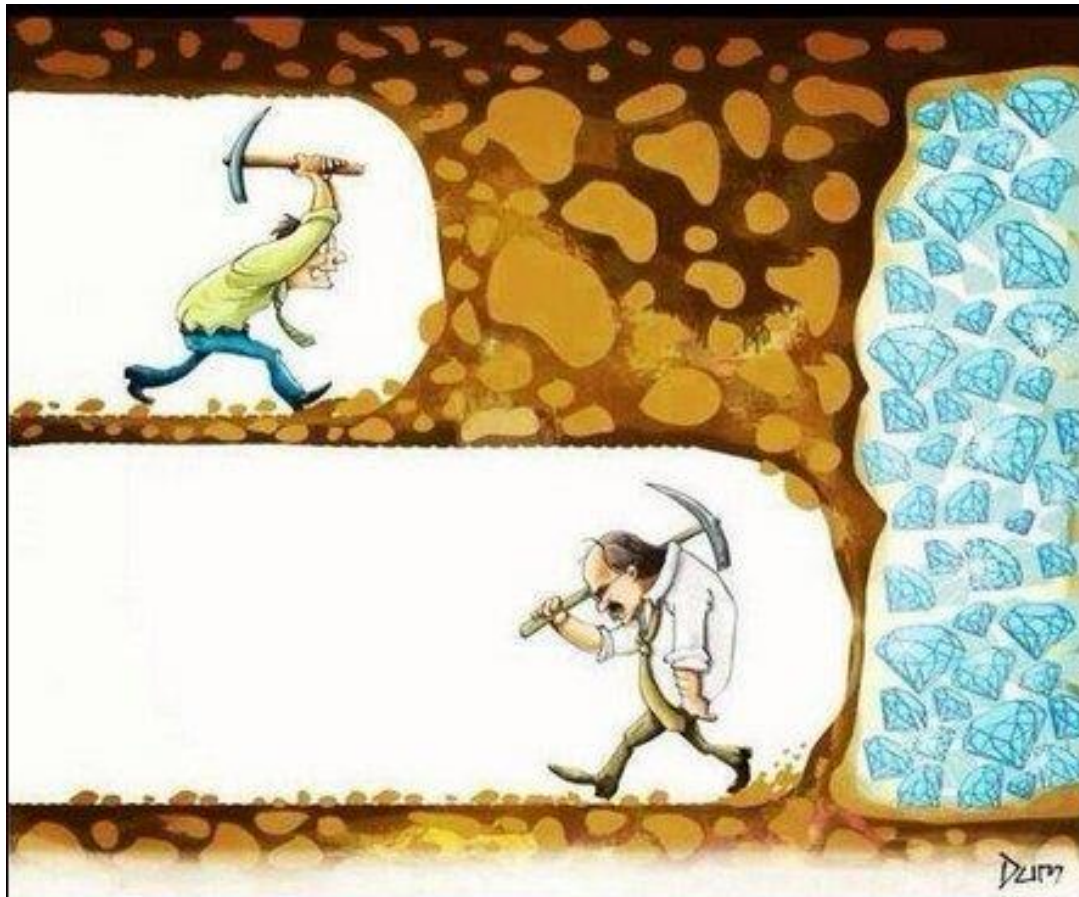


weights

Count	Vertex	Visited	0	1	2	3	4
1	–	0	0	∞	∞	6	4
2	4	0,4	0	∞	7	5	4
3	3	0,4,3	0	6	7	5	4
4	1	0,4,3,1	0	6	7	5	4
5	2	0,4,3,1,2	0	6	7	5	4

What if unweighted?

- All distances are 1.
- Equivalent to **breadth-first search**
- Look for all vertices that are distance 1 away from the start (or currently dequeued element), put in a queue, repeat until all nodes are marked.



DZIM

NEVER GIVE UP!!!