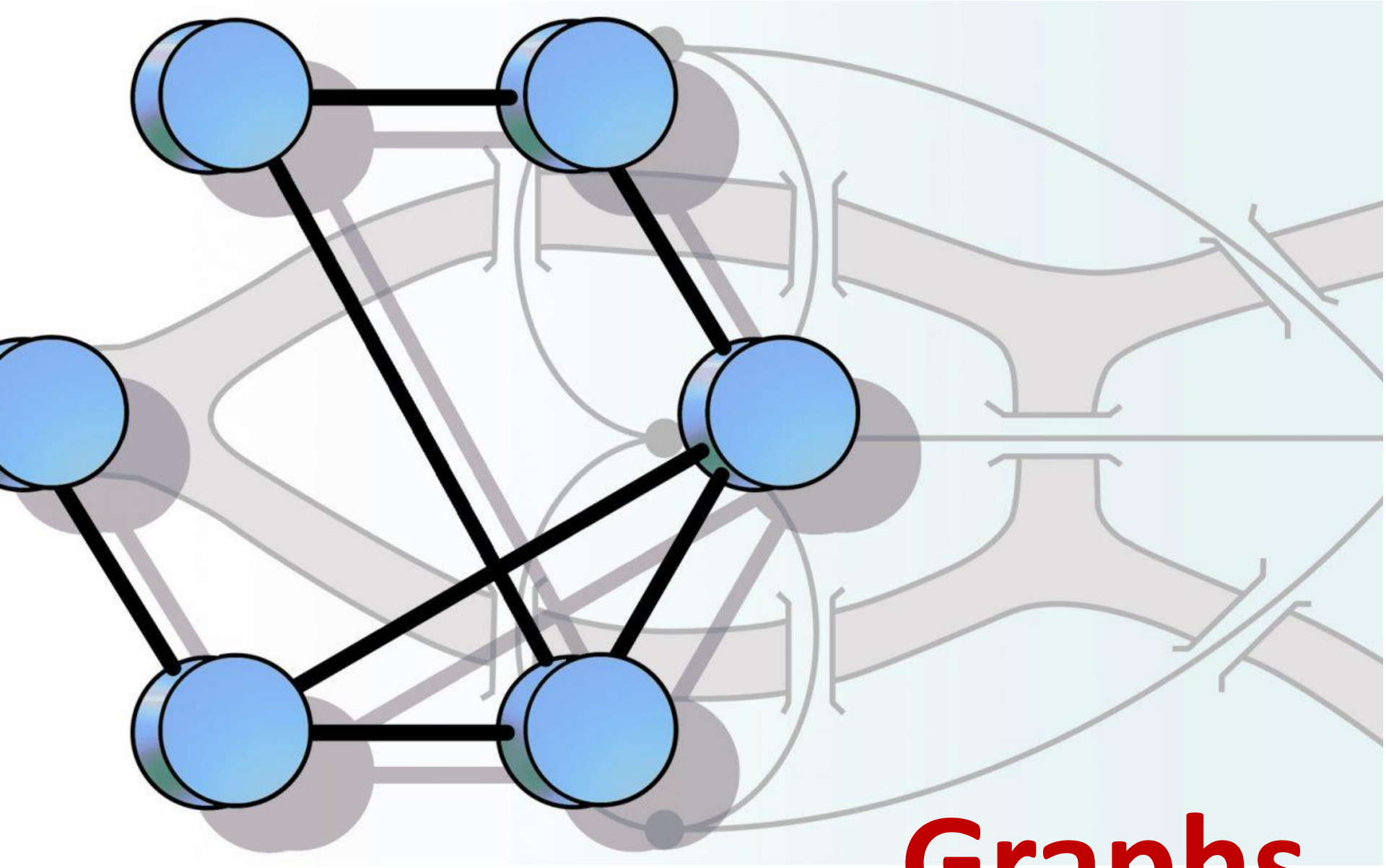
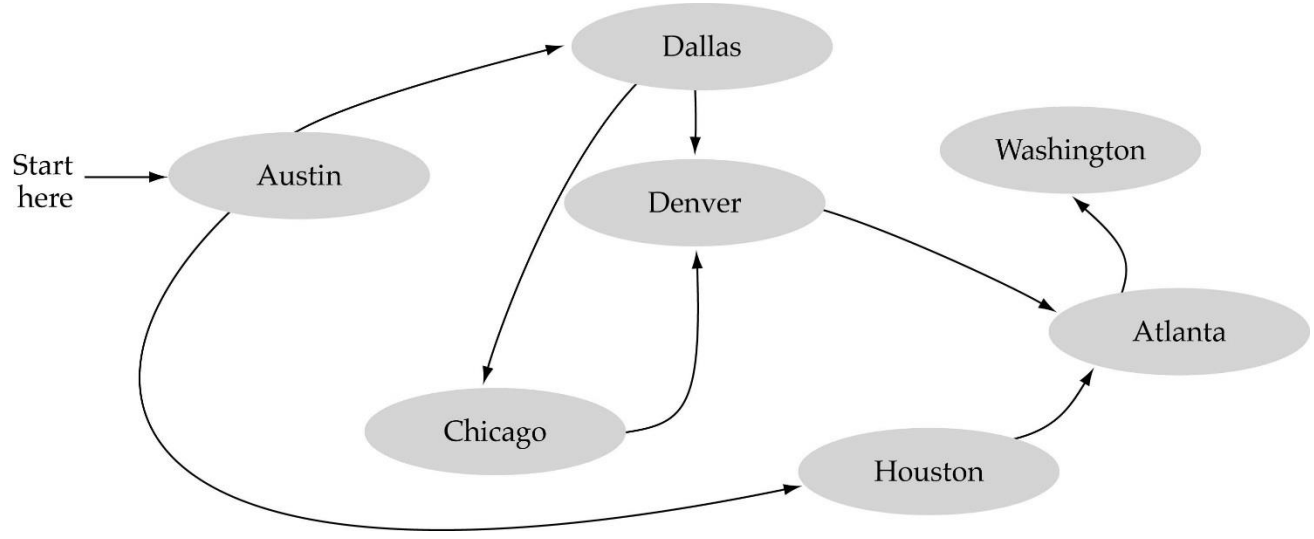


Graphs

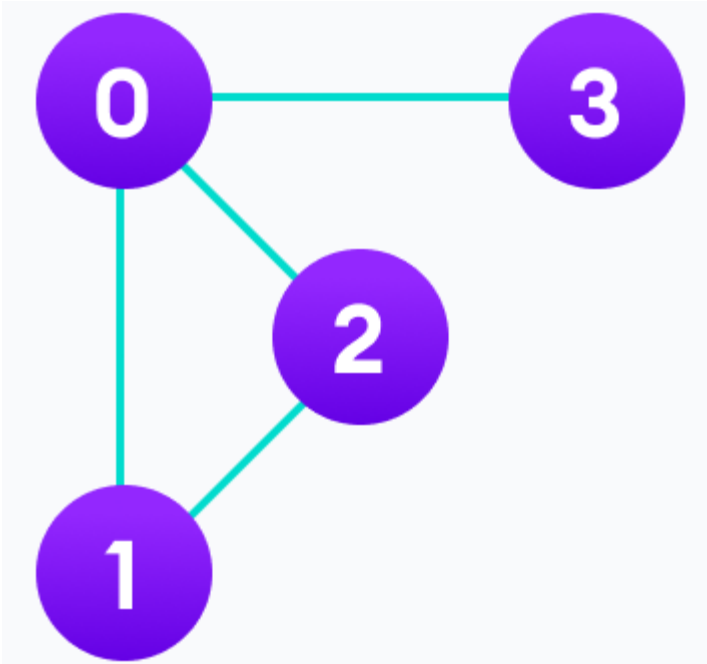
What is a graph?

- A data structure that consists of a set of nodes (**vertices**) and a set of **edges between the vertices.**
- The **set of edges describes relationships among the vertices**.



A graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices **(u,v)**



Vertices and edges

In the graph,

$V = \{0, 1, 2, 3\}$

$E = \{(0,1), (0,2), (0,3), (1,2)\}$

$G = \{V, E\}$

Terminology

- **Definition**:
  - A set of points that are joined by lines
- Graphs also represent the relationships among data items
- G = { V , E }
  - **a graph is a set of vertices and edges**
- A **subgraph** consists of a **subset of a graph's vertices and a subset of its edges**

Formally

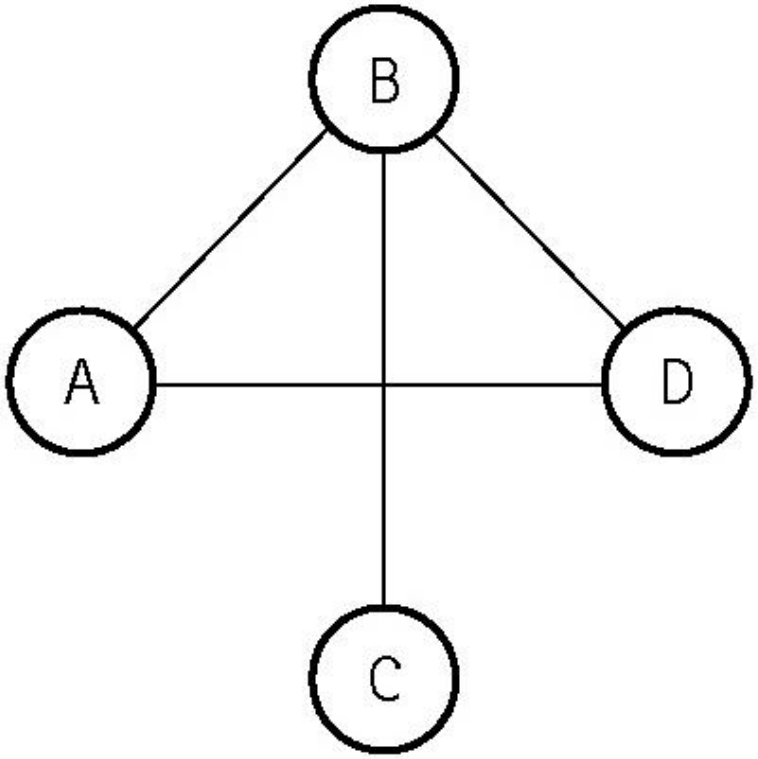
- a graph G is defined as follows:

$$G = (V,E)$$

- where
  - **V(G) is a finite, nonempty set of vertices**
  - **E(G) is a set of edges**
    - written as pairs of vertices

An undirected graph

A graph in which the edges have no direction



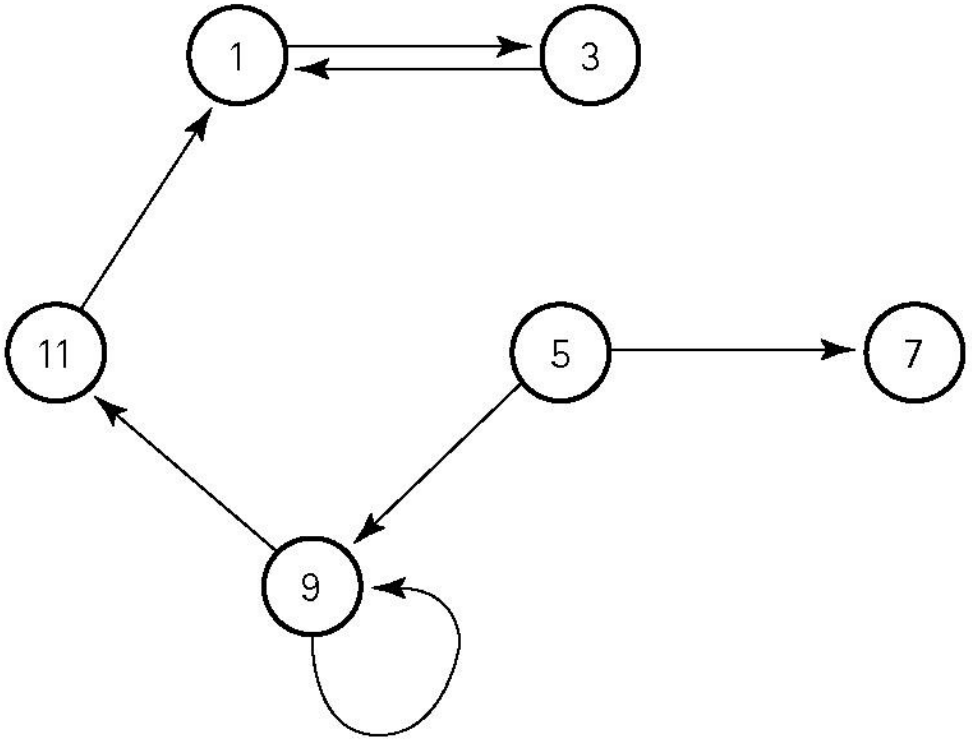
The order of vertices in E **is not** important for undirected graphs!!

V(Graph 1) = {A, B, C, D}
E(Graph 1) = {(A, B), (A, D), (B, C), (B, D)}

A Directed Graph

A graph in which each edge is directed from one vertex to another (or the same) vertex

(b) Graph2 is a directed graph.



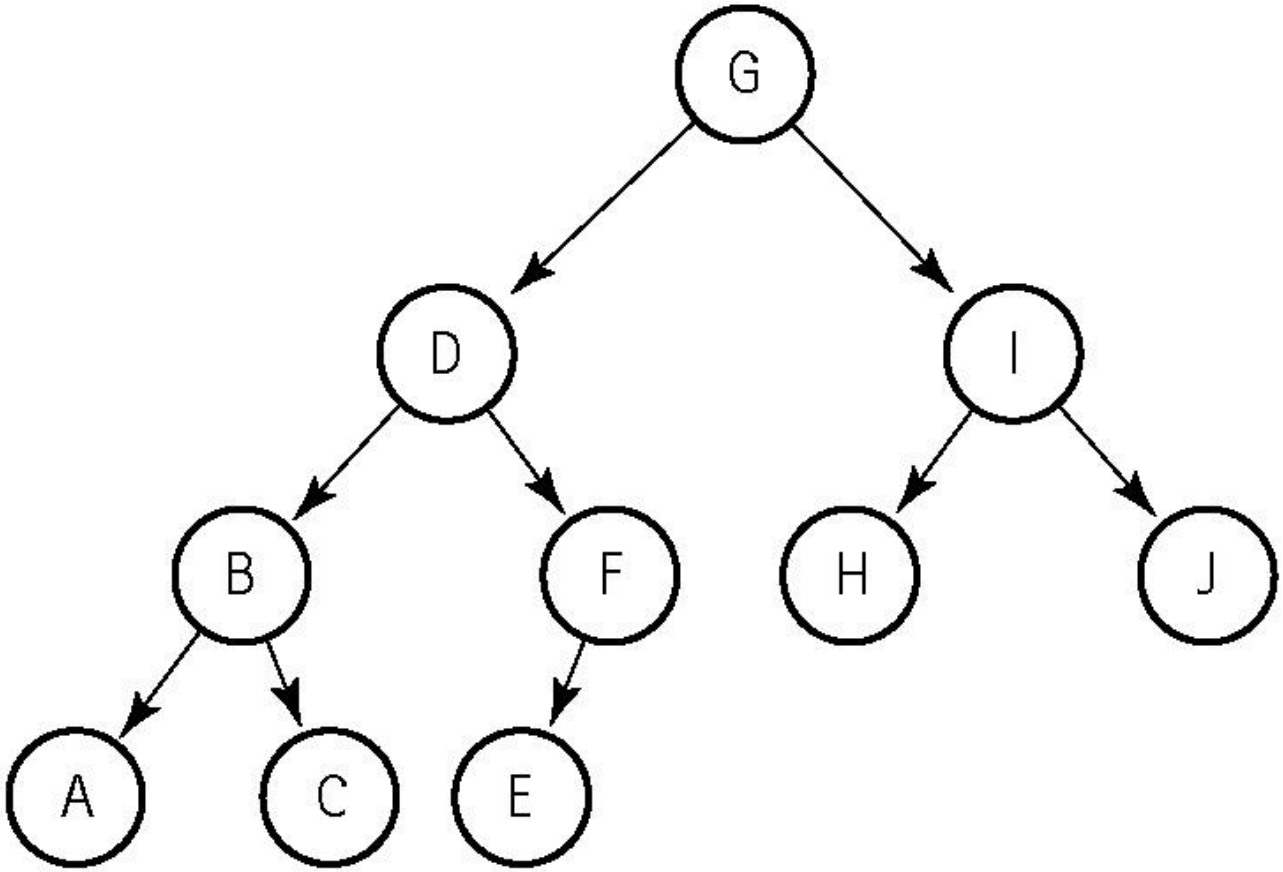
V(Graph2) = {1, 3, 5, 7, 9, 11}
E(Graph2) = {(1, 3), (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)}

The order of vertices in E
is important for directed graphs!!

A Directed Graph

Trees are special cases of graphs!

(c) Graph3 is a directed graph.



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

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

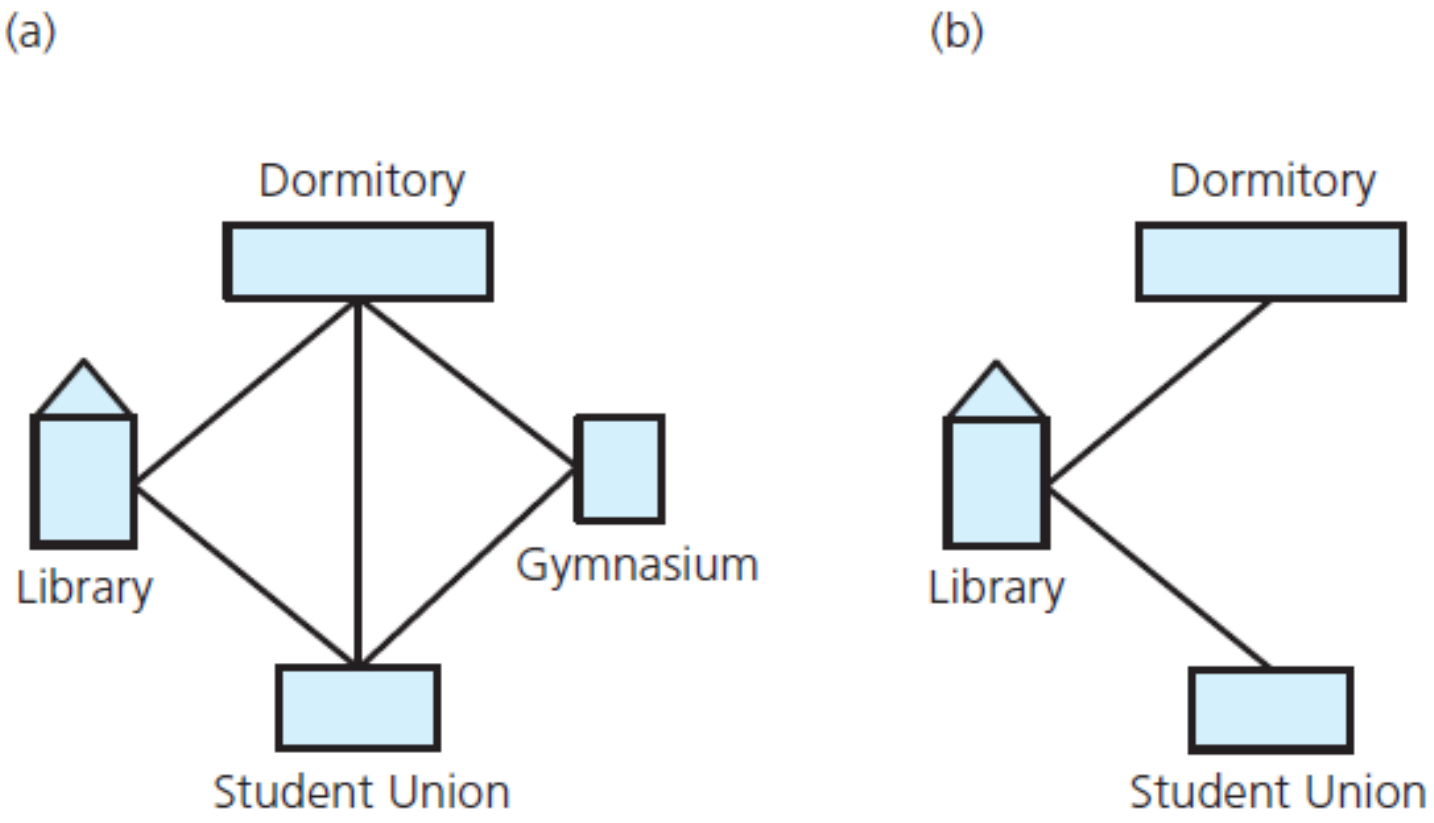
Terminology

- **Undirected graphs**: edges do not indicate a direction
- **Directed graph, or digraph**: each edge has a direction

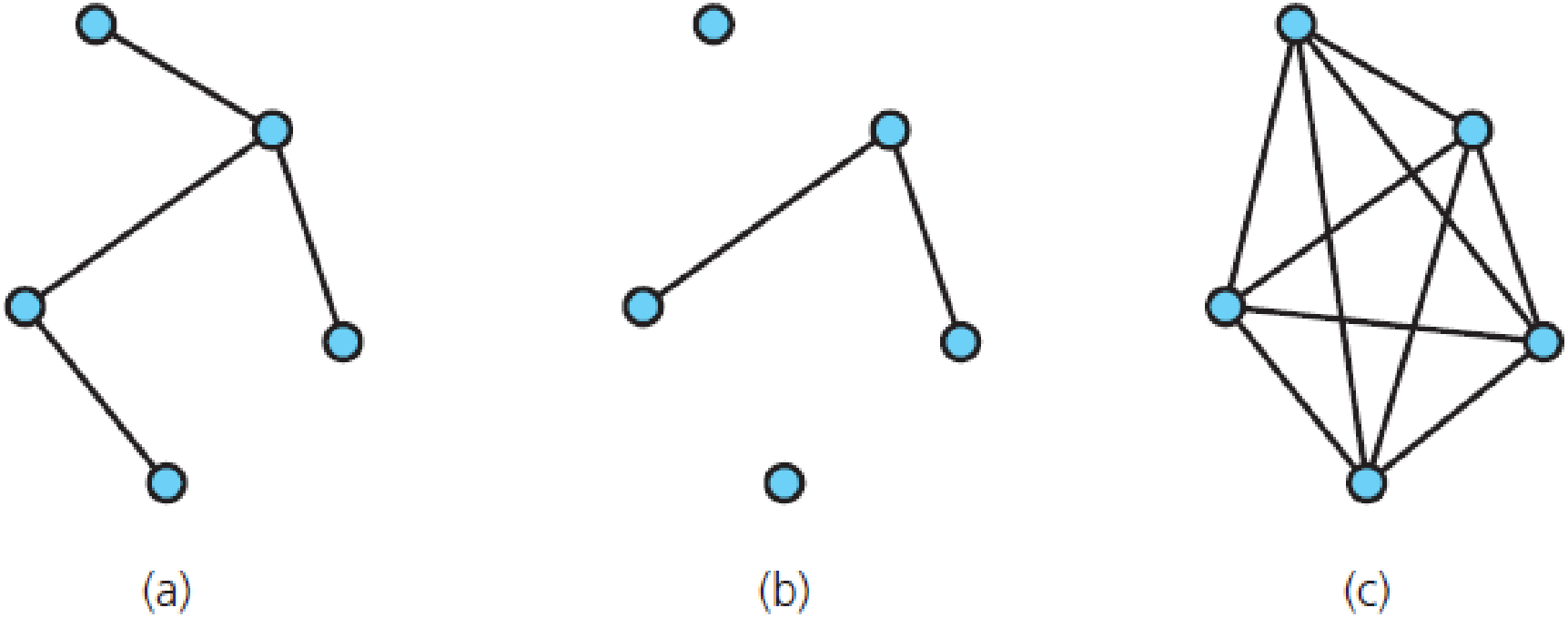
Terminology

(a) A campus map as a graph

(b) A subgraph



Terminology



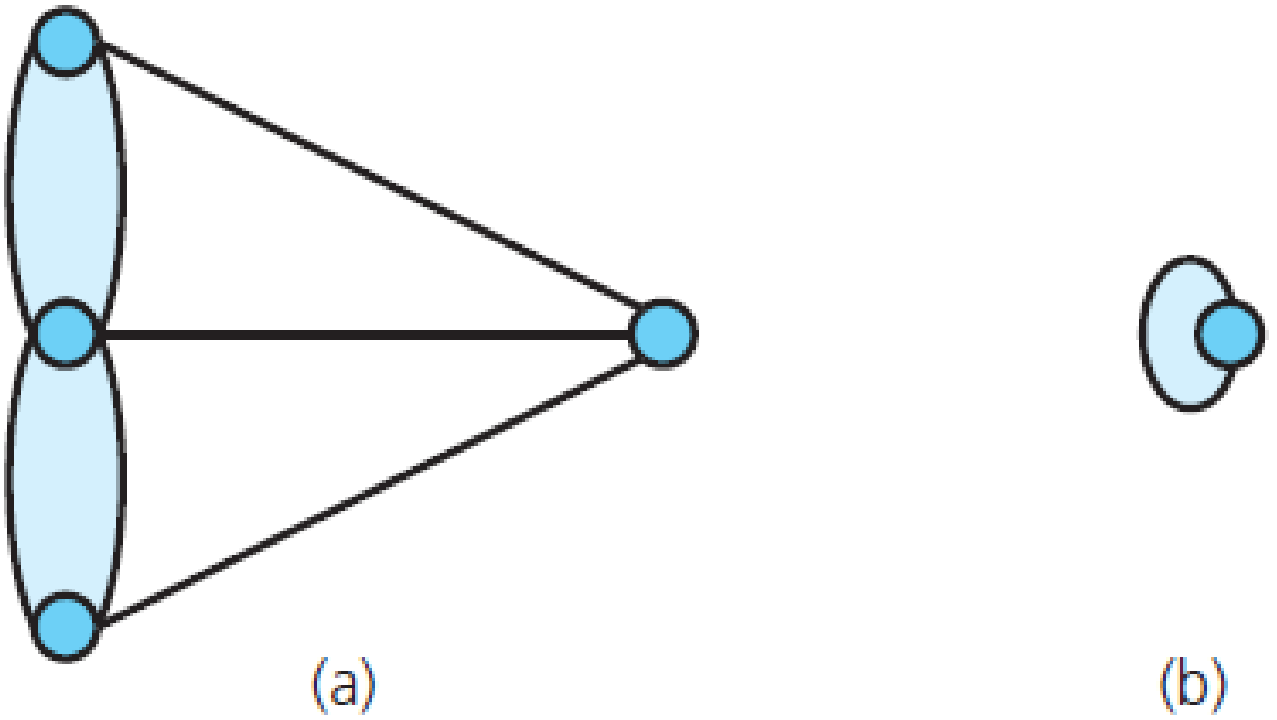
Graphs that are

(a) connected

(b) disconnected and

(c) complete

Terminology

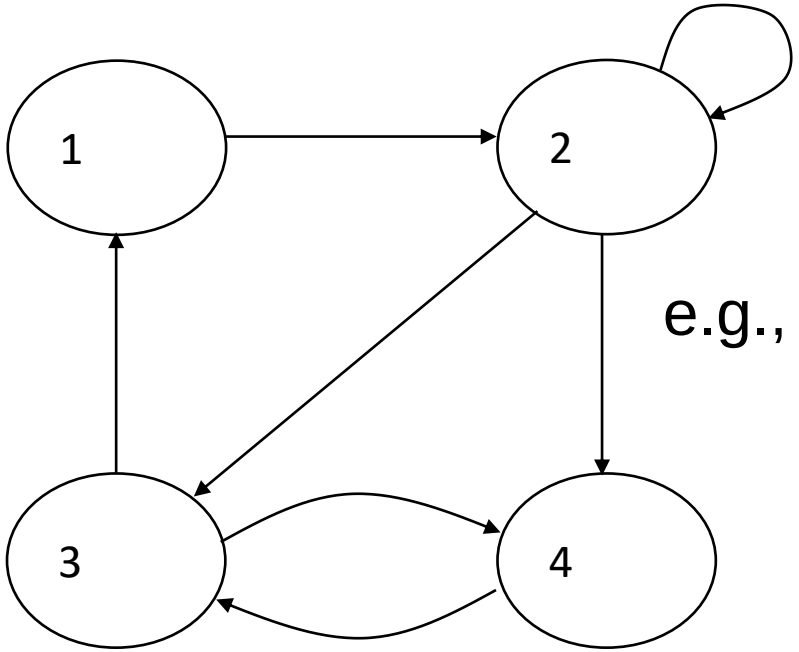


(a) A multigraph is not a simple graph;

(b) a self edge is not allowed in a simple graph

Terminology

- **Path**: A **sequence of vertices that connects two nodes in a graph**
- The **length** of a path is the number of edges in the path.



e.g., a path from 1 to 4

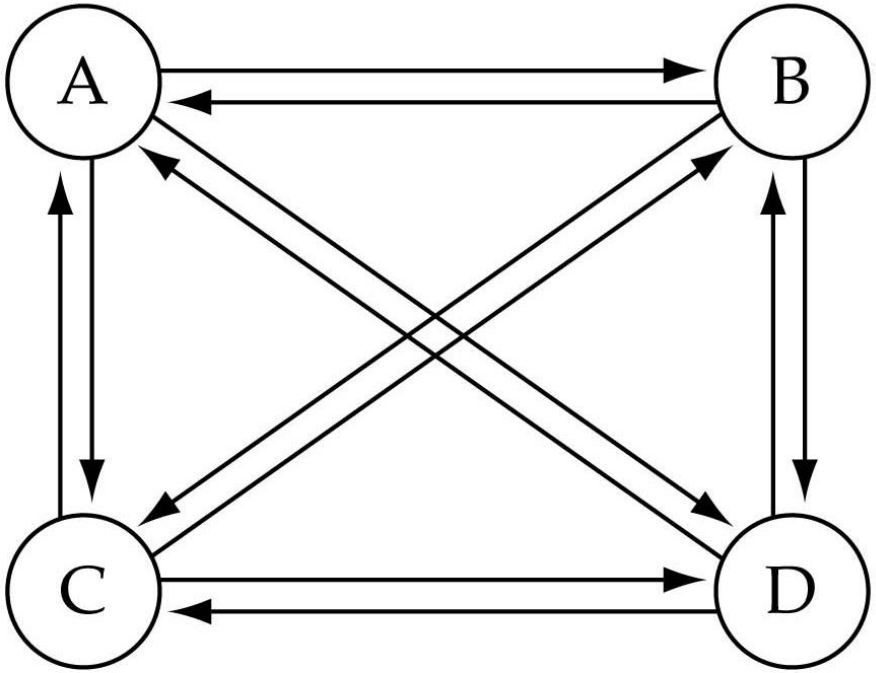
<1, 2, 3, 4>

Terminology

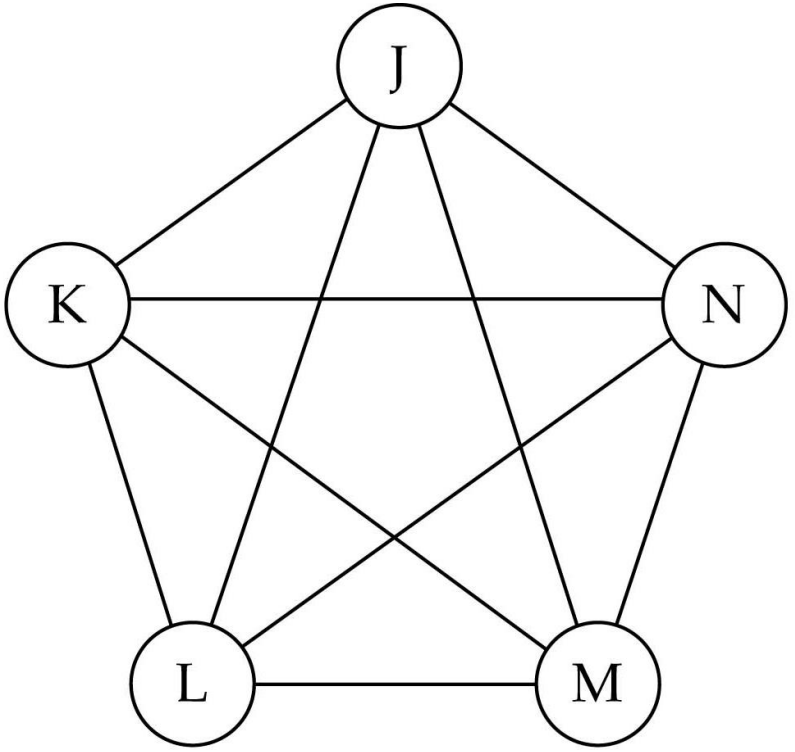
- **Simple path**: passes through vertex only once
- **Cycle**: a path that begins and ends at same vertex
- **Simple cycle**: cycle that does not pass through other vertices more than once
- **Connected graph**: each pair of distinct vertices has a path between them

Terminology

Complete graph: A graph in which every vertex is directly connected to every other vertex



(a) Complete directed graph.



(b) Complete undirected graph.

Terminology

- **Complete graph**: each pair of distinct vertices has an edge between them
- Graph **cannot have duplicate edges between vertices**
  - **Multigraph**: does allow multiple edges
- When labels represent numeric values, graph is called a **weighted graph**

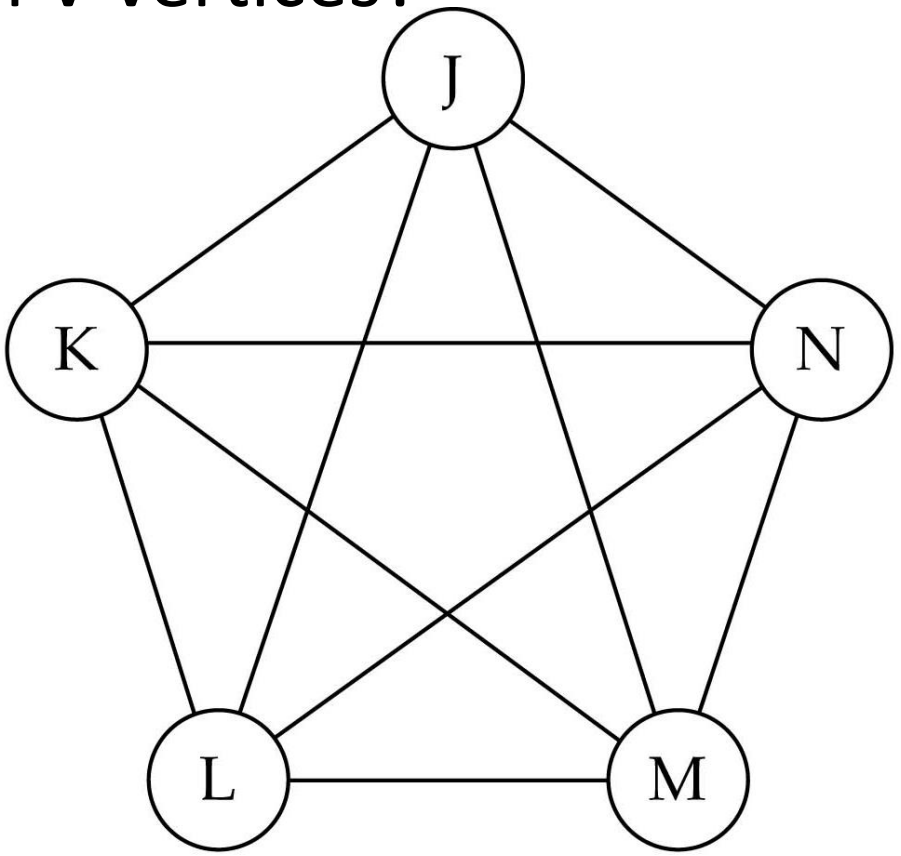
Terminology

What is the number of edges E in a **complete undirected graph** with V vertices?

E=V* (V-1) / 2

or

O(V^2)



(b) Complete undirected graph.

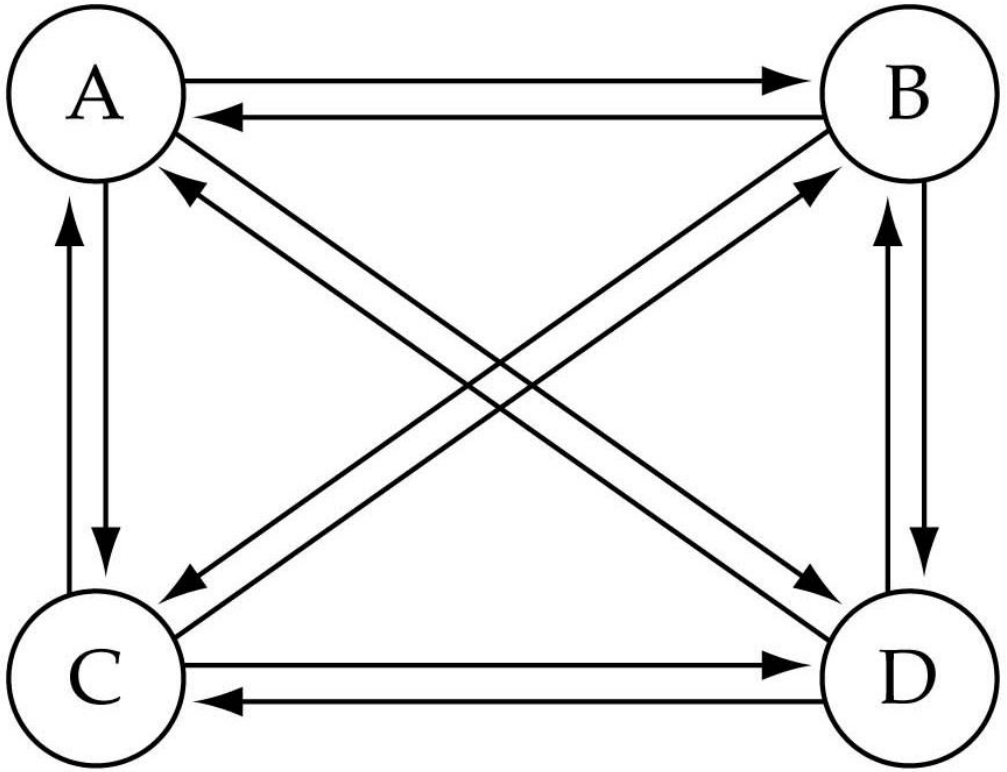
Terminology

What is the number of edges E in a **complete directed graph** with V vertices?

E=V * (V-1)

or

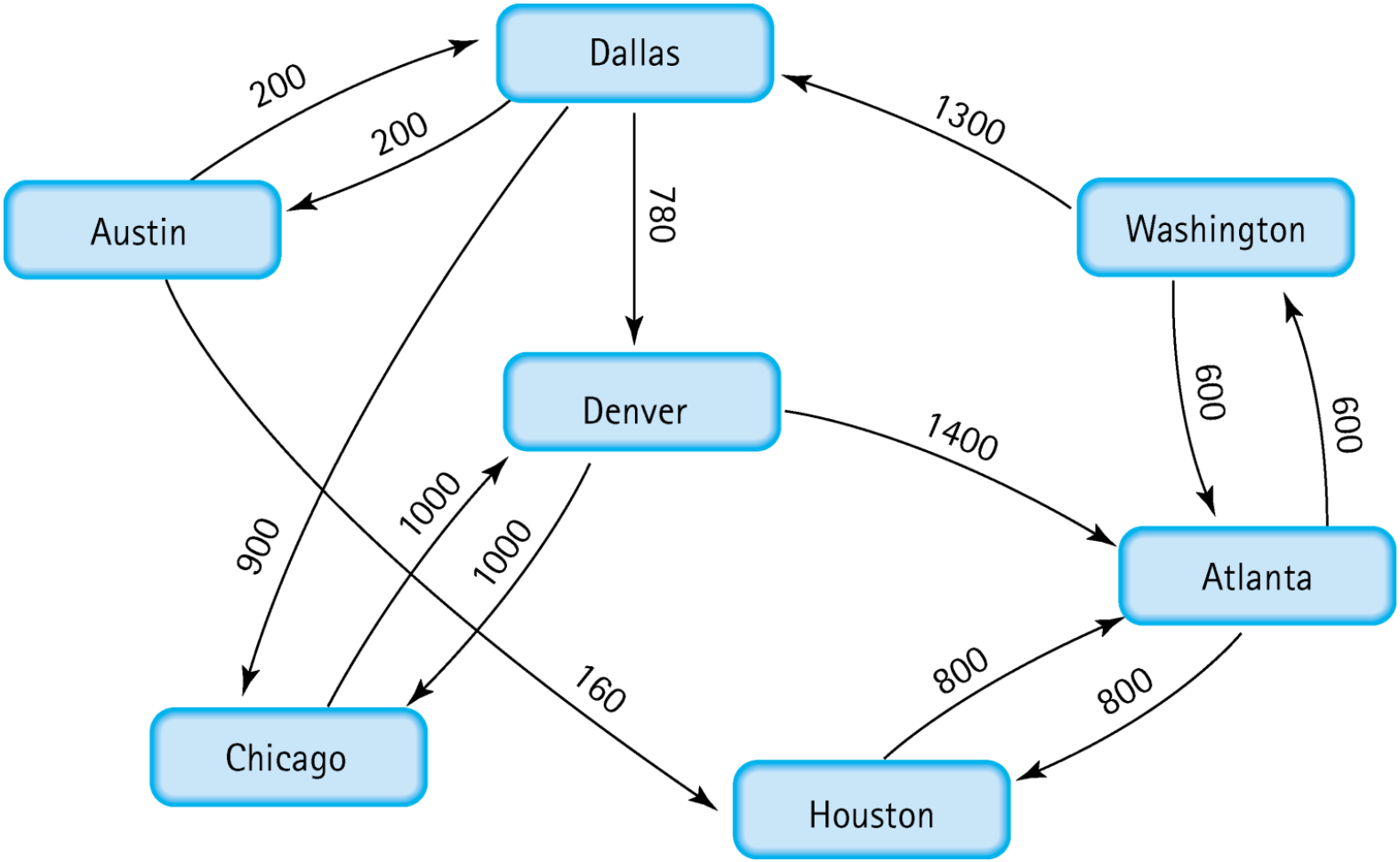
O(V^2)



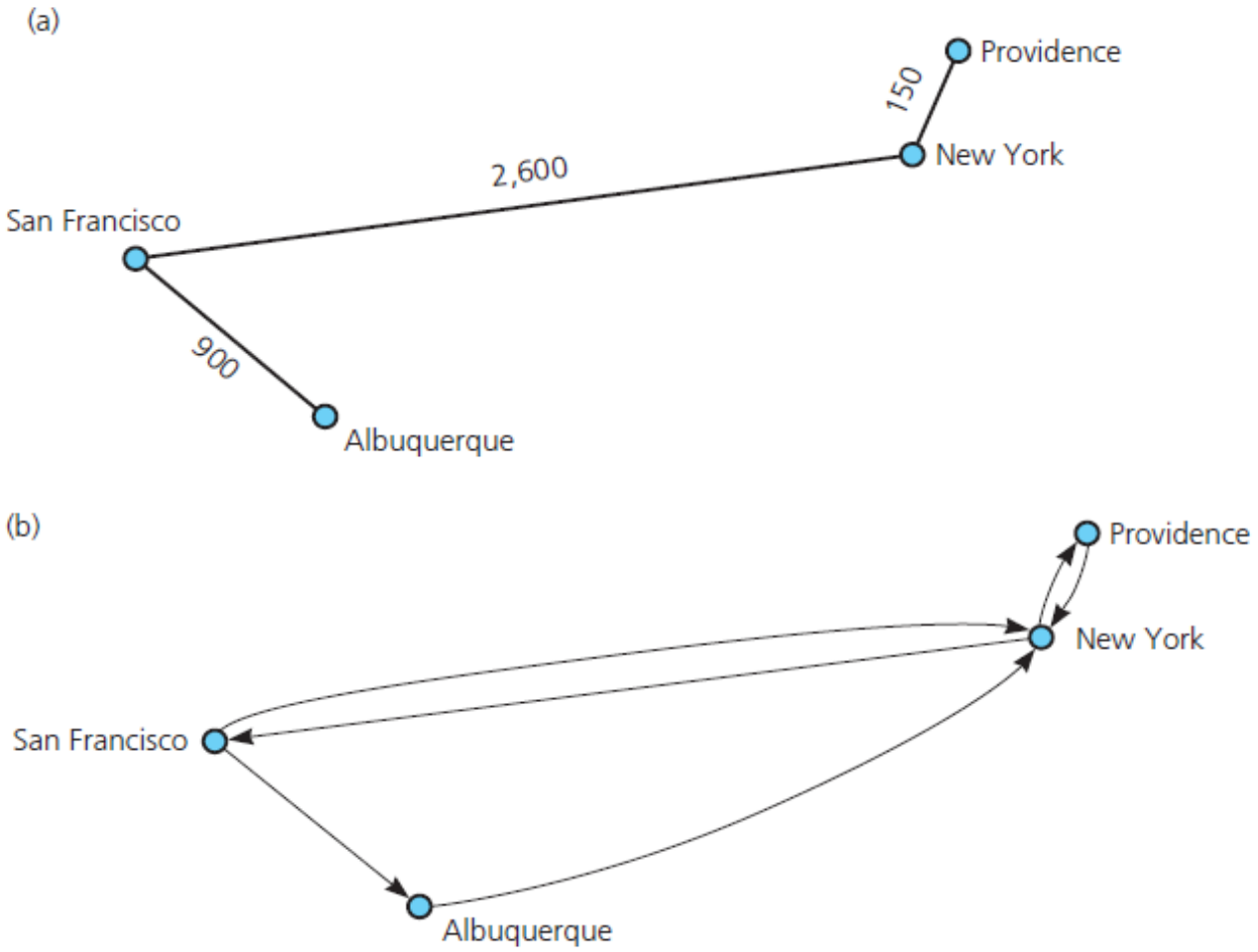
(a) Complete directed graph.

A Weighted Graph

Weighted graph: A graph in which each edge carries a value



Terminology



(a) A Weighted graph (b) A Directed graph

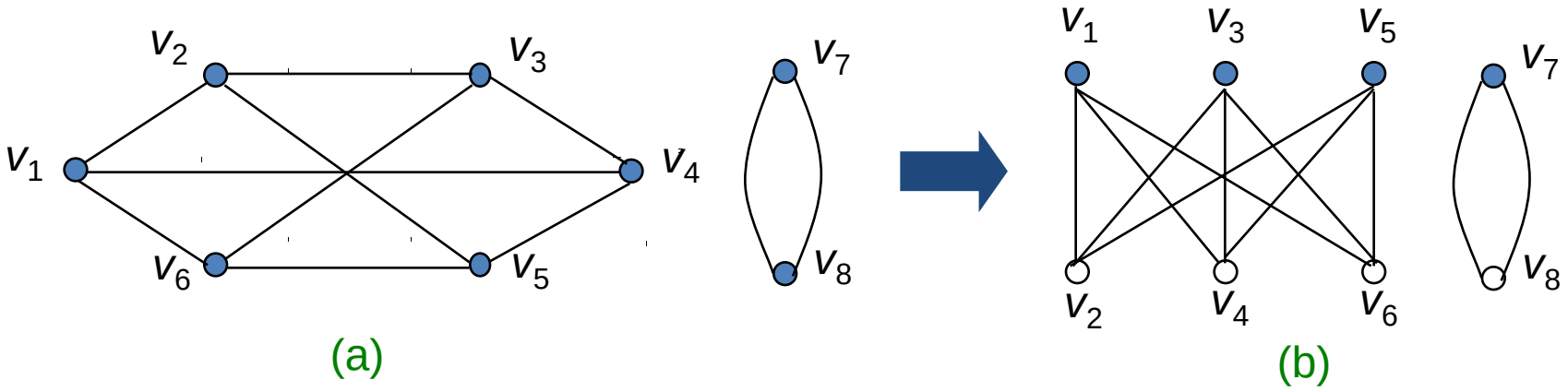
Bipartite Graph

1. A graph *G* is *bipartite* **if the node set *V* can be partitioned into two sets V_1 and V_2 in such a way that no nodes from the same set are adjacent.**
2. **The sets V_1 and V_2 are called the *color classes* of *G* and (V_1, V_2) is a bipartition of *G*.** In fact, a graph being bipartite means that the **nodes of *G* can be colored with at most two colors, so that no two adjacent nodes have the same color.**

Bipartite Graph

3. We will depict bipartite graphs with their nodes colored black and white to show one possible bipartition.

4. We will call a graph *m by n bipartite*, if $|V_1| = m$ and $|V_2| = n$, and a graph a *balanced bipartite graph* when $|V_1| = |V_2|$.



Properties of Bipartite Graph

Property 1 A connected bipartite graph has a **unique bipartition.**

Property 2 A bipartite graph with **no isolated nodes** and ***p* connected components has 2^{p-1} bipartitions**.

For example, the bipartite graph in the above figure has two bipartitions. One is shown in the figure and the other has $V_1 = \{v_1, v_3, v_5, v_8\}$ and $V_2 = \{v_2, v_4, v_6, v_7\}$.

Properties of Bipartite Graph

The following theorem belongs to König (1916).

Theorem 3 A graph G is bipartite if and only if **G has no cycle of odd length.**

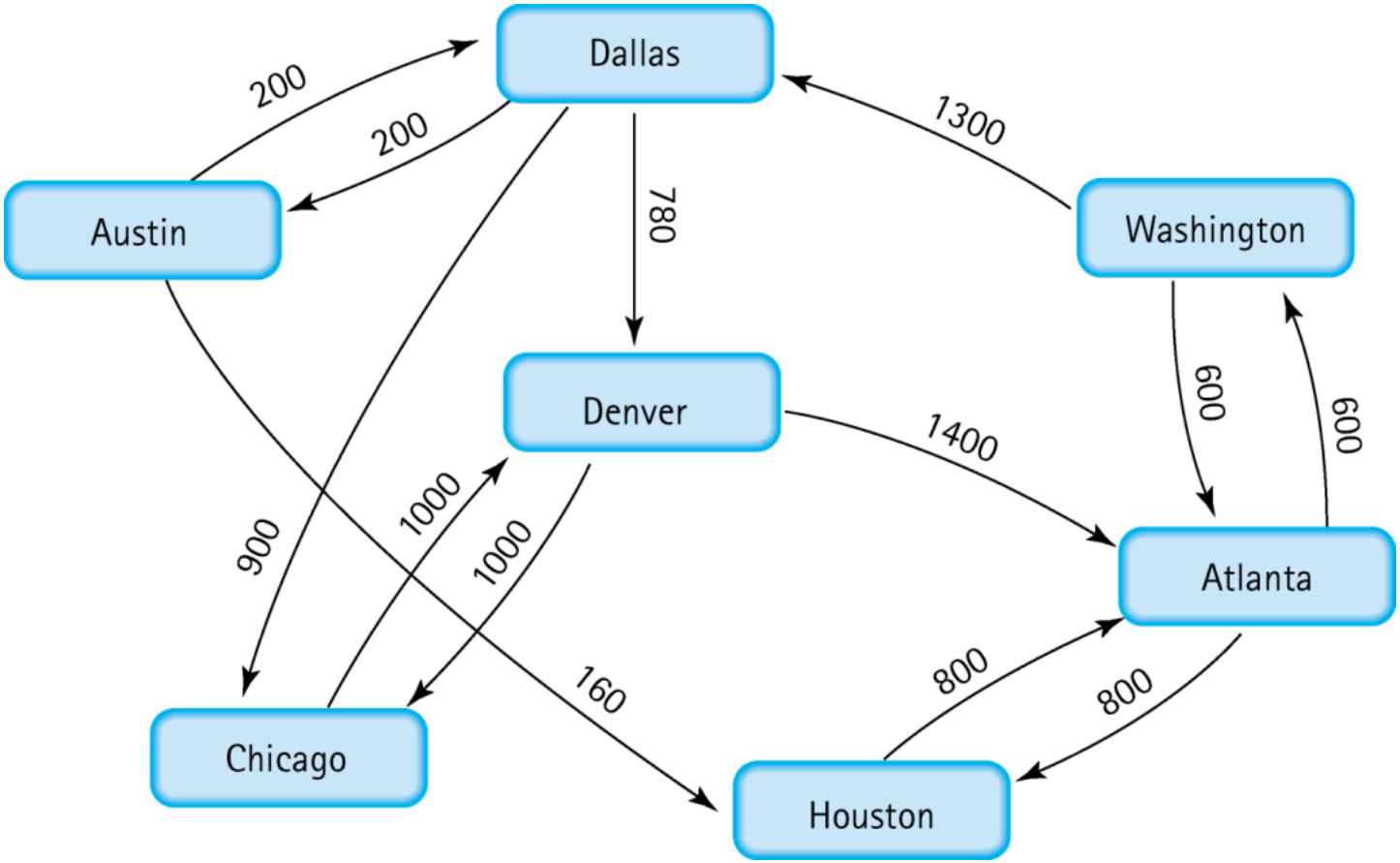
Corollary 4 A connected graph is bipartite if and only if for every node v there is no edge (x, y) such that distance(v, x) = distance(v, y).

Corollary 5 A graph G is bipartite if and only if it contains no closed walk of odd length.

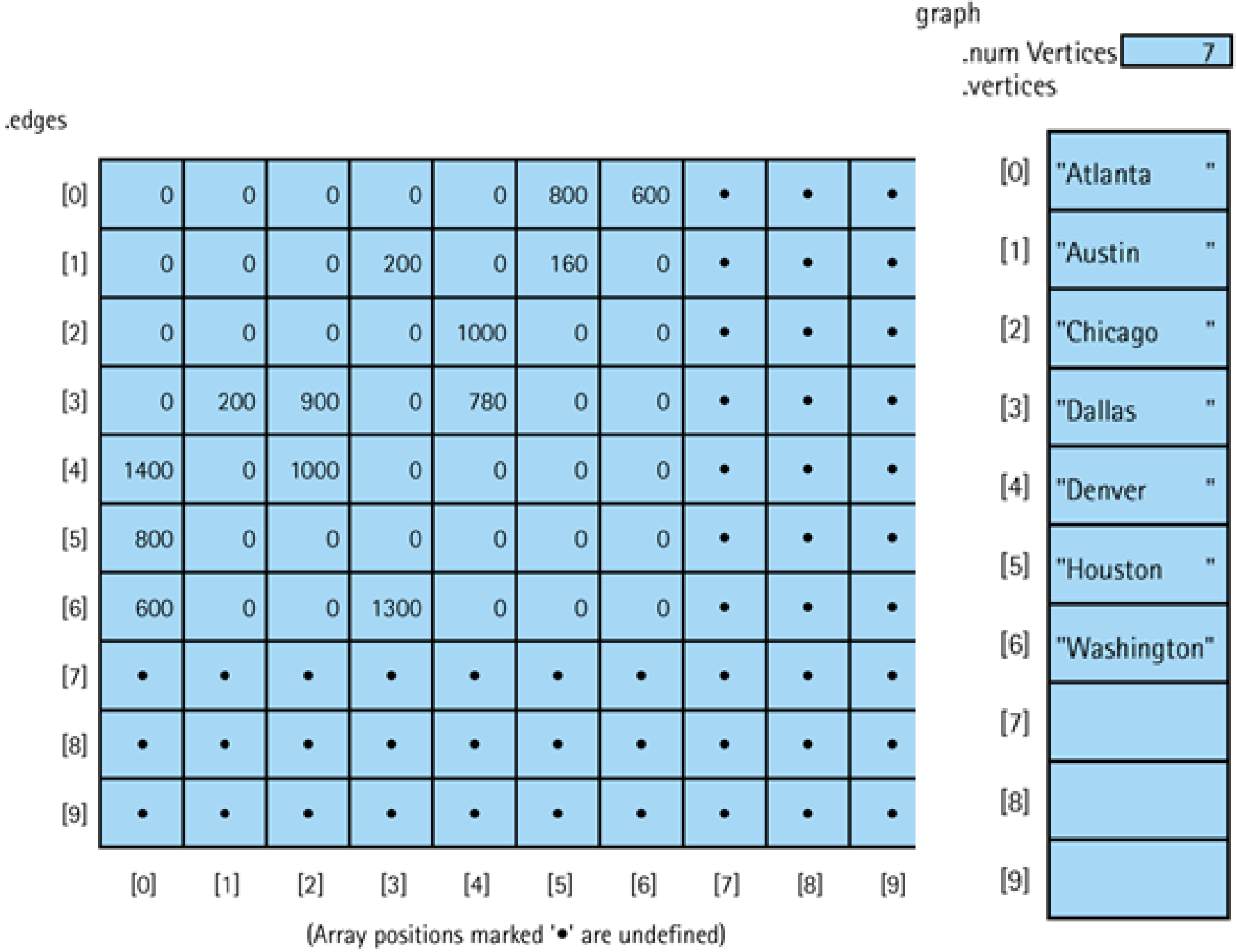
Array-Based Implementation

- **Use a 1D array to represent the vertices**
- **Use a 2D array (i.e., adjacency matrix) to represent the edges**
- **Adjacency Matrix:**
  - for a graph with N nodes,
- an N by N table that shows the existence (and weights) of all edges in the graph

Adjacency Matrix for Flight Connections

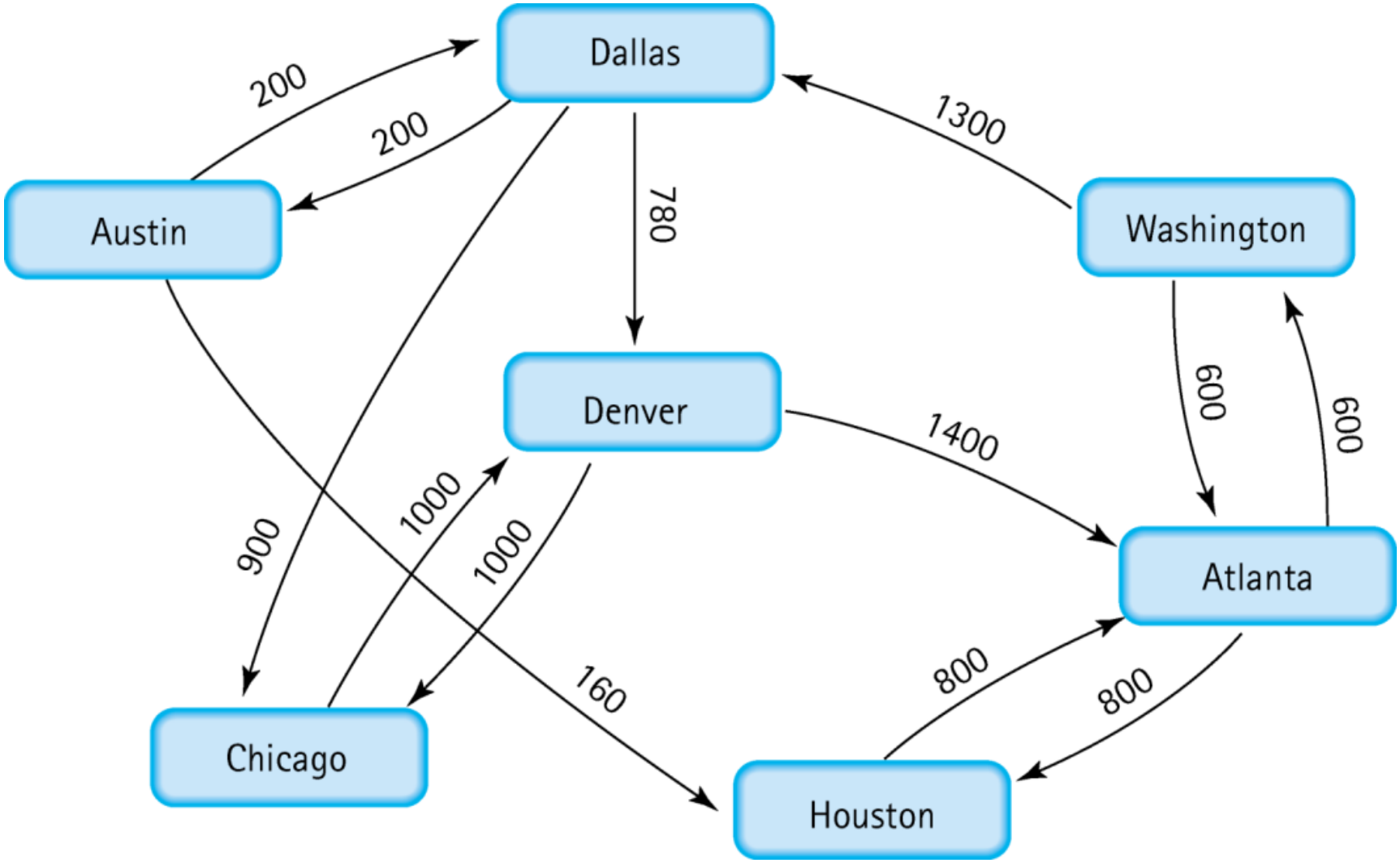


from node x ? to node x ?


graph
.num Vertices
7
.vertices
.edges
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[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] • • • • • • • • • •
(Array positions marked '•' are undefined)

Array-Based Implementation

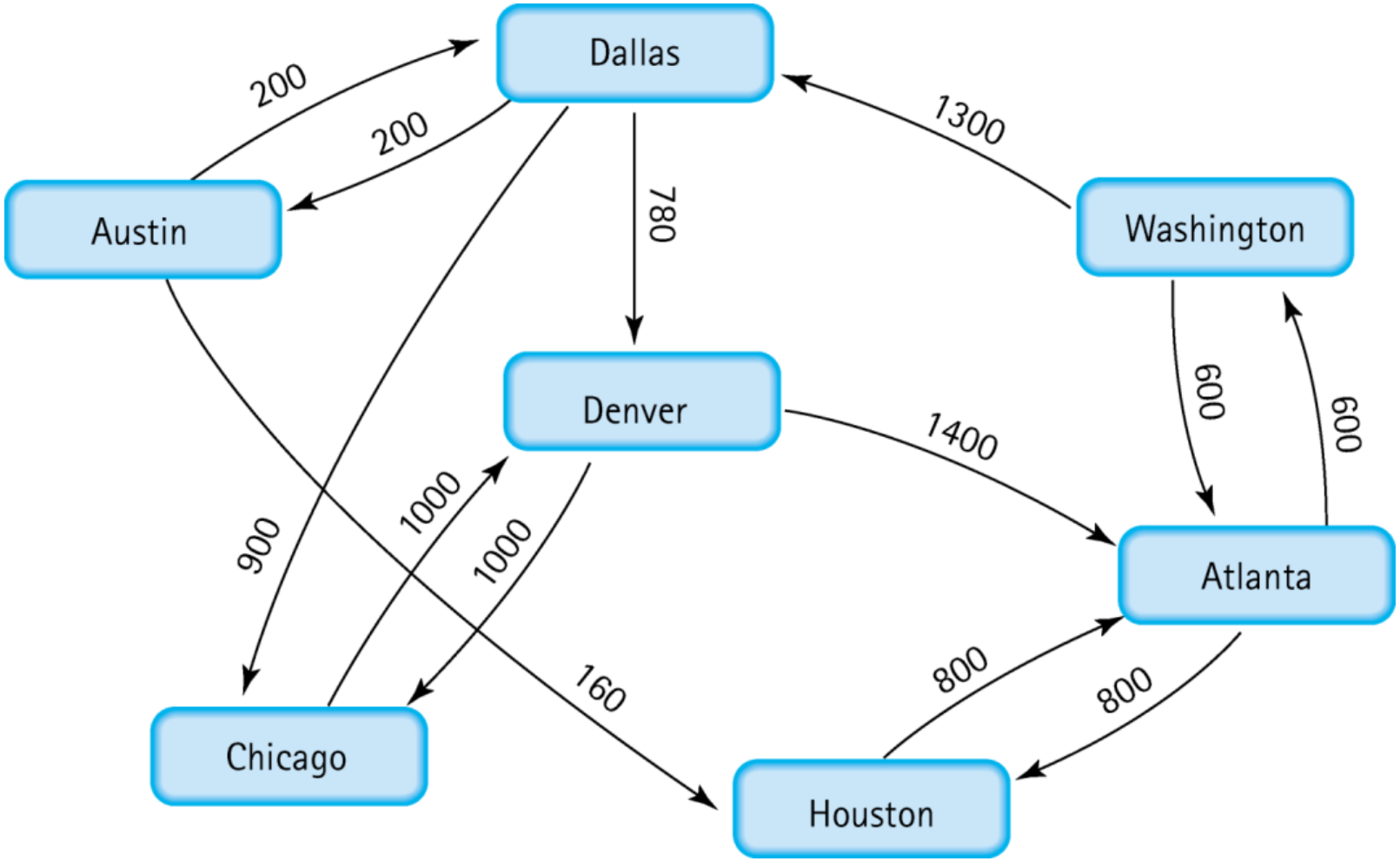
- Memory required
  - $O(V+V^2)=O(V^2)$
- Preferred when
  - The graph is **dense:** $E = O(V^2)$
- Advantage
  - Can quickly determine if there is an edge between two vertices
- Disadvantage
  - Consumes significant memory for sparse large graphs


Dallas
Austin
Washington
Denver
Atlanta
Chicago
Houston
200
200
780
1300
600
600
1400
1000
1000
900
160
800
800

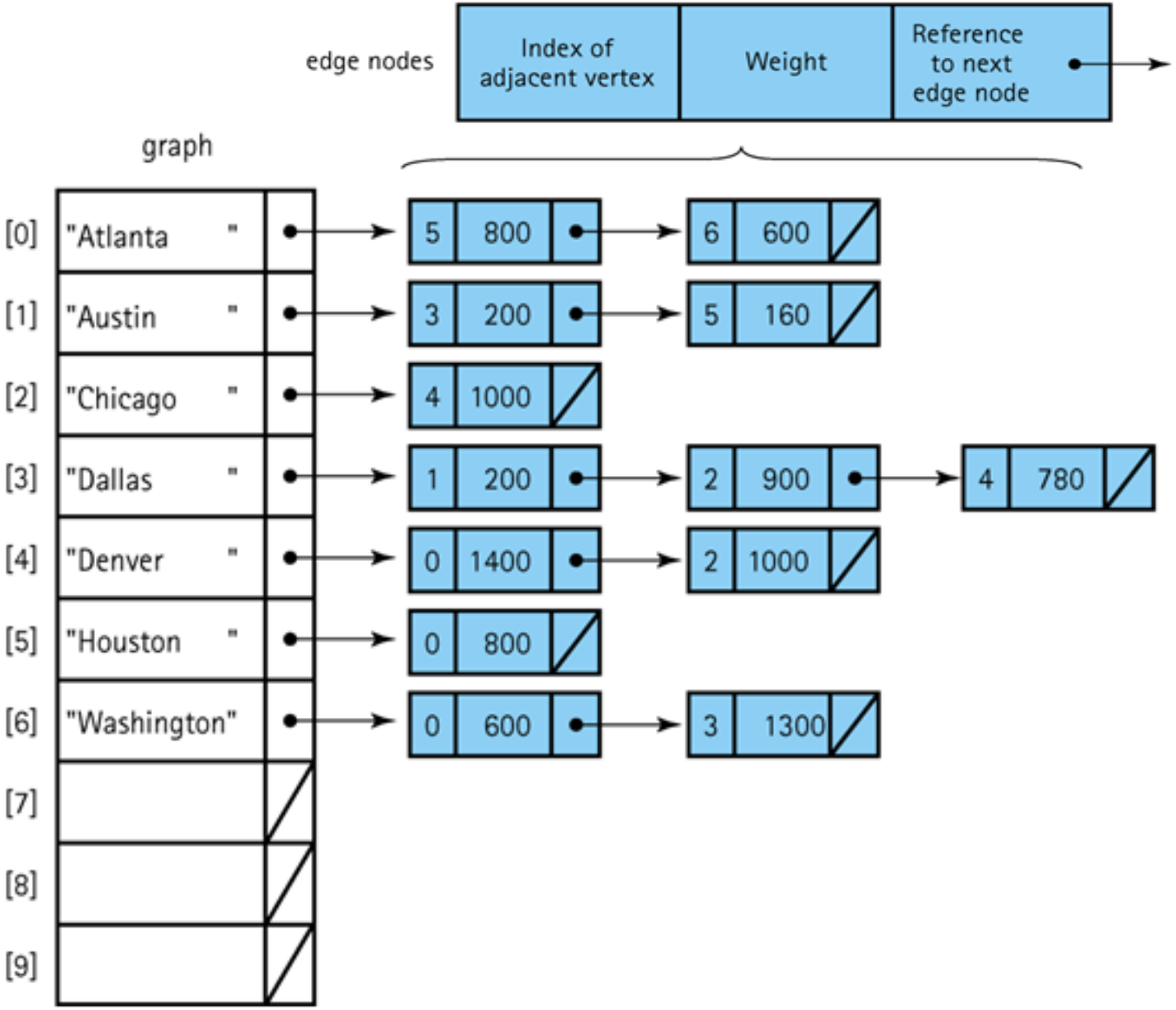
Linked Implementation

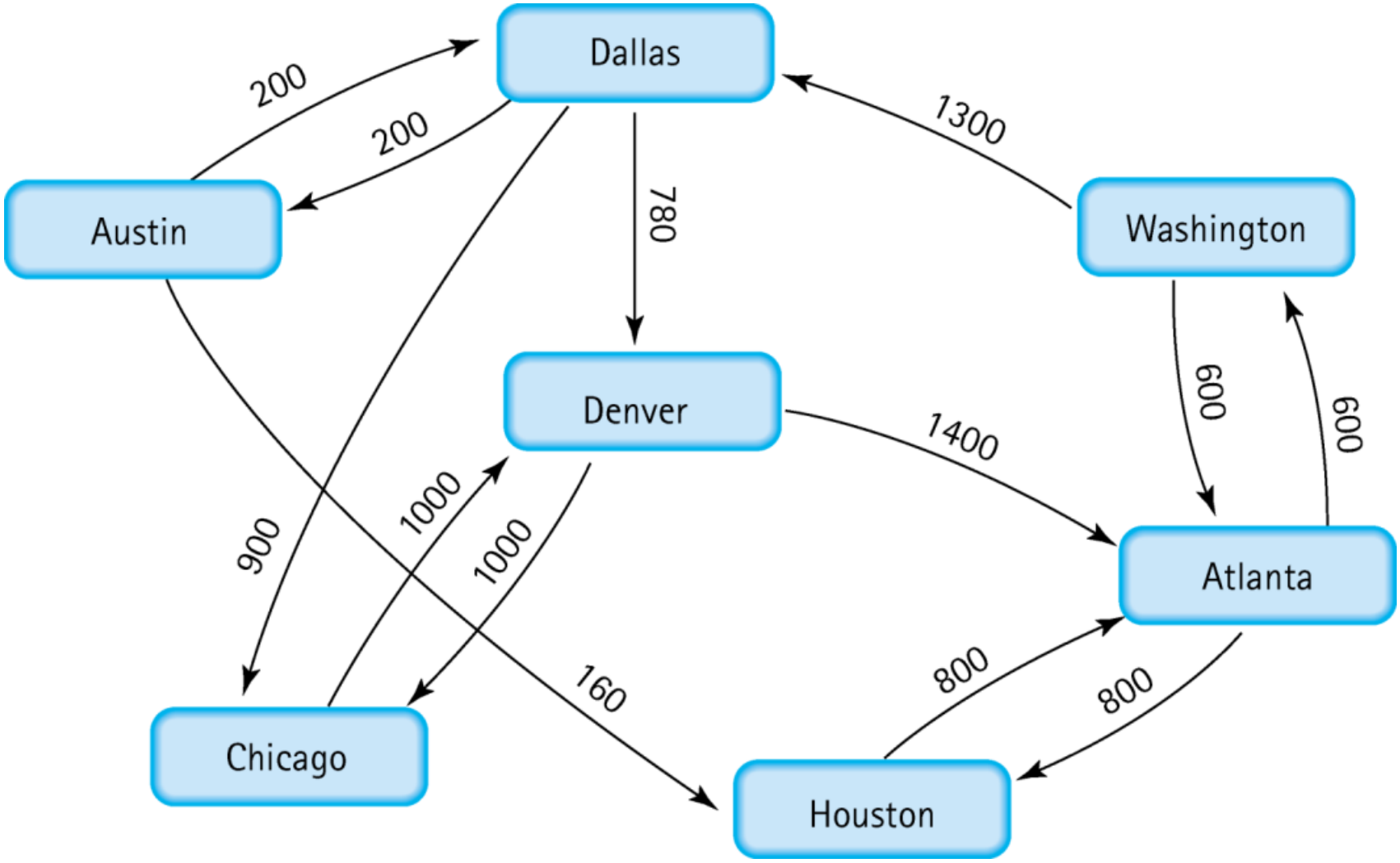
- Use a 1D array to represent the vertices
- Use a list for each vertex *v* which contains the vertices which are adjacent **from** *v* (i.e., adjacency list)
- **Adjacency List:**
  - A linked list that identifies all the vertices to which a particular vertex is connected;
    - each vertex has its own adjacency list

Adjacency List Representation of Graphs



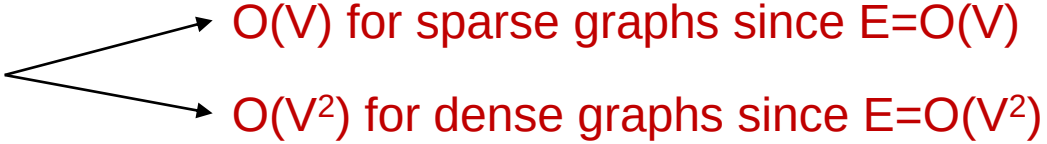
from node x ? to node x ?


edge nodes
Index of adjacent vertex
Weight
Reference to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800
6 600
3 200
5 160
4 1000
1 200
2 900
4 780
0 1400
2 1000
0 800
0 600
3 1300


Dallas
Austin
Washington
Denver
Atlanta
Chicago
Houston
200
200
780
1300
600
600
1400
900
1000
1000
160
800
800

Link-List-based Implementation

- Memory required
  - O(V + E)



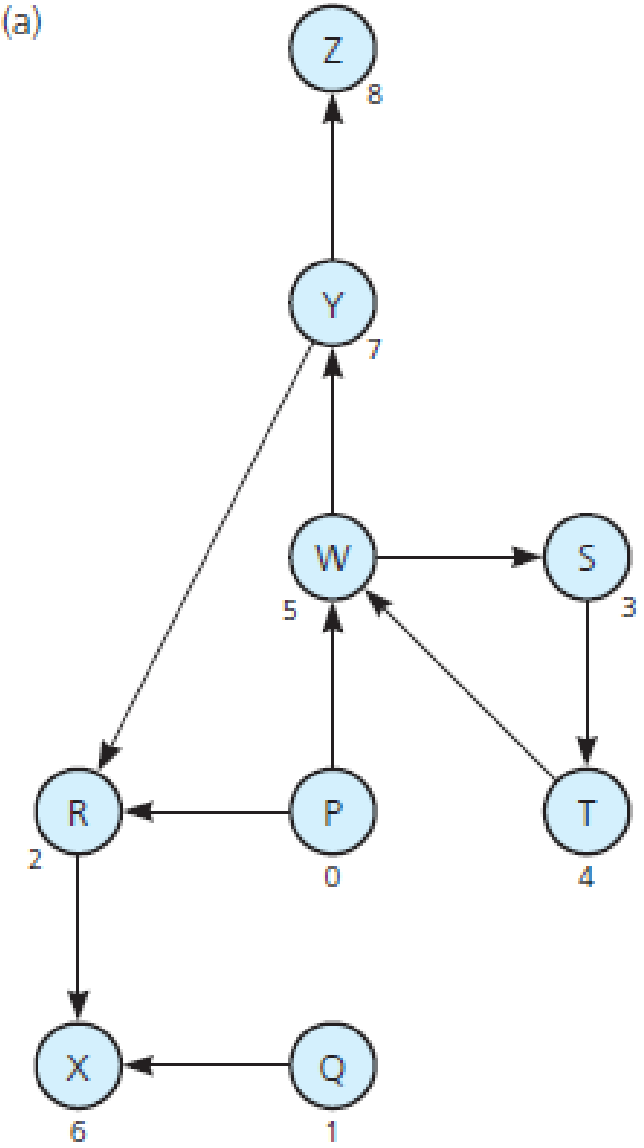
- Preferred when
  - for **sparse** graphs: E = O(V)
- Disadvantage
  - No quick way to determine the vertices adjacent to a given vertex
- Advantage
  - Can quickly determine the vertices adjacent **from** a given vertex

Implementing Graphs

(a)A directed graph and

(b)Its adjacency matrix

(a)

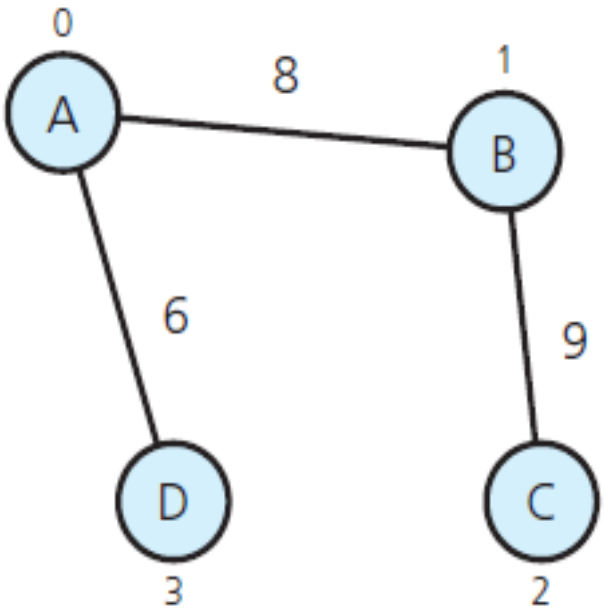


(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Implementing Graphs

(a)



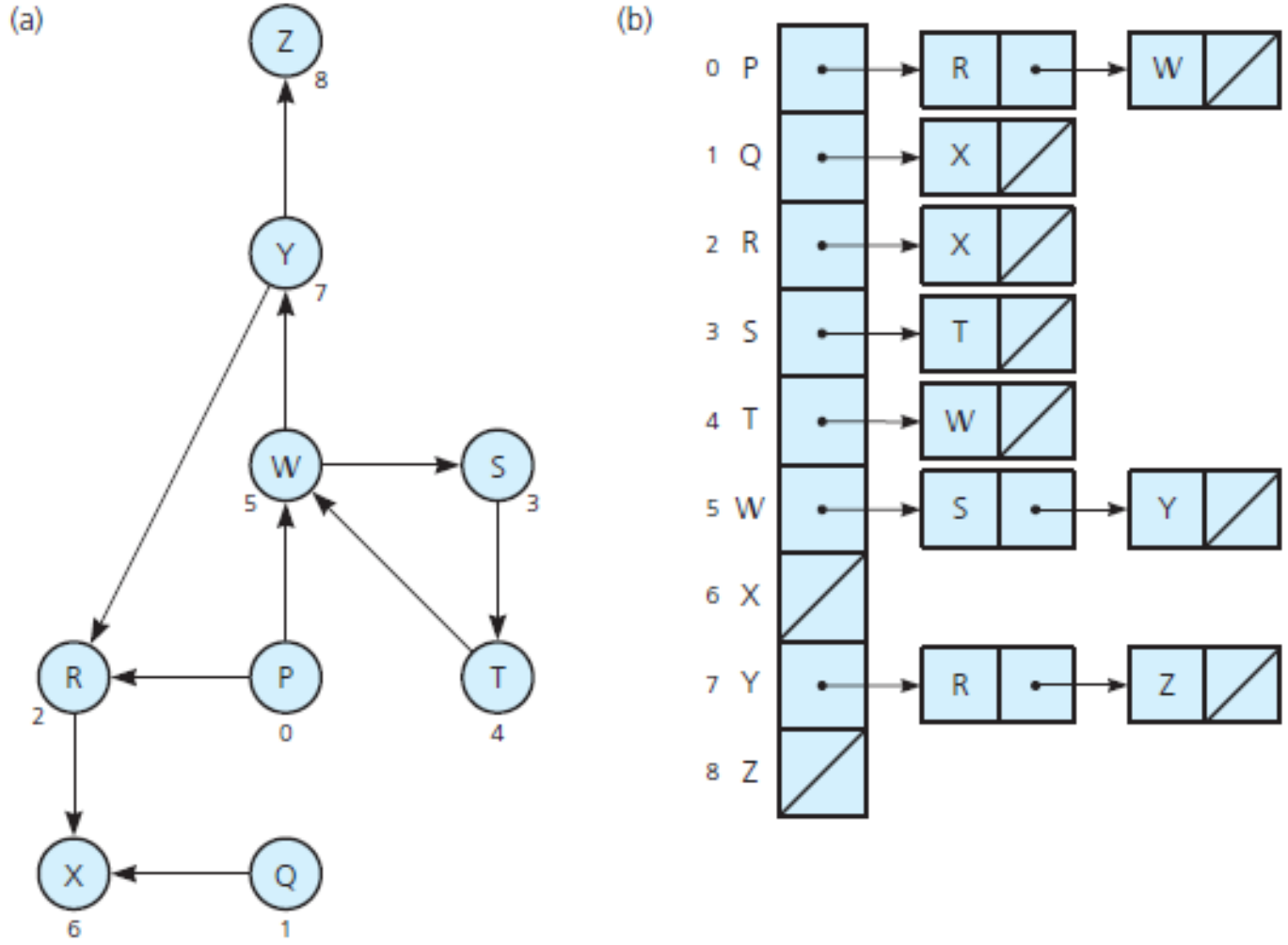
(b)

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

(a)A weighted undirected graph

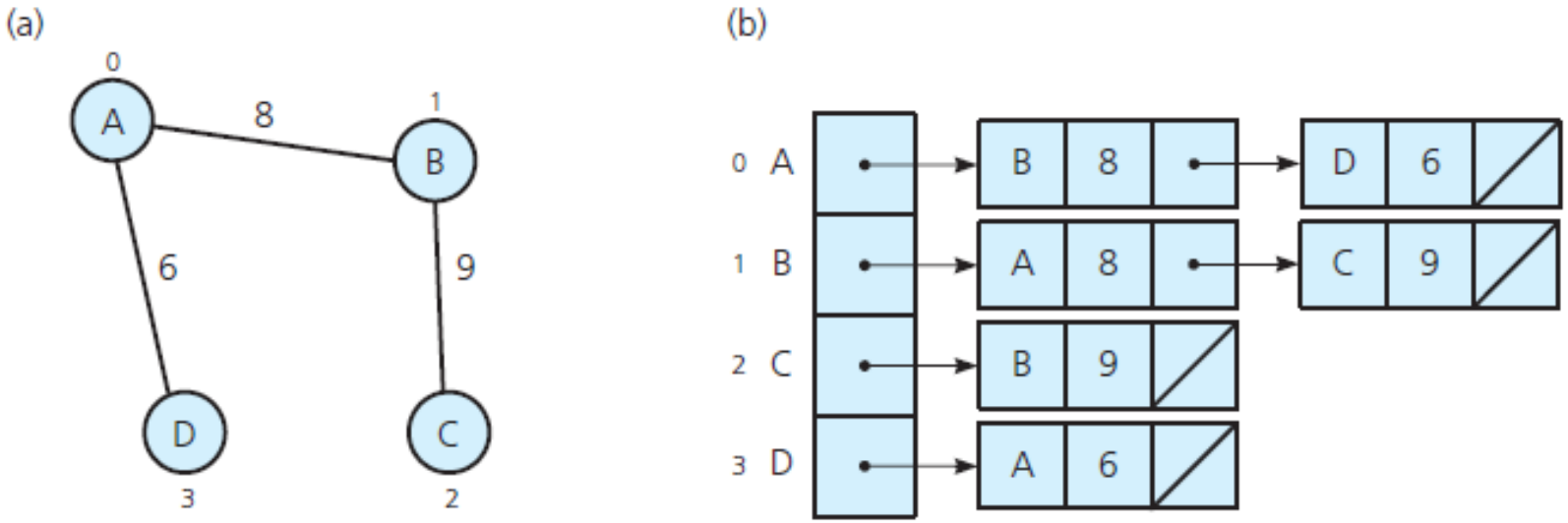
(b)Its adjacency matrix

Implementing Graphs



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

Implementing Graphs



(a) A weighted undirected graph
(b) Its adjacency list

Graph Traversals

- Visits all of the vertices that it can reach
  - Happens if graph is connected
- Connected component is subset of vertices visited during traversal that begins at given vertex

Graph Searching

- **Problem**: Find if there is a path between two vertices of the graph
  - e.g., Austin and Washington
- **Methods:** Depth-First-Search (DFS) or Breadth-First-Search (BFS)

Depth-First-Search (DFS)

Traversal means visiting all the nodes of a graph. Depth first traversal or Depth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

DFS algorithm

- A standard DFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Keep repeating steps 2 and 3 until the stack is empty.

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

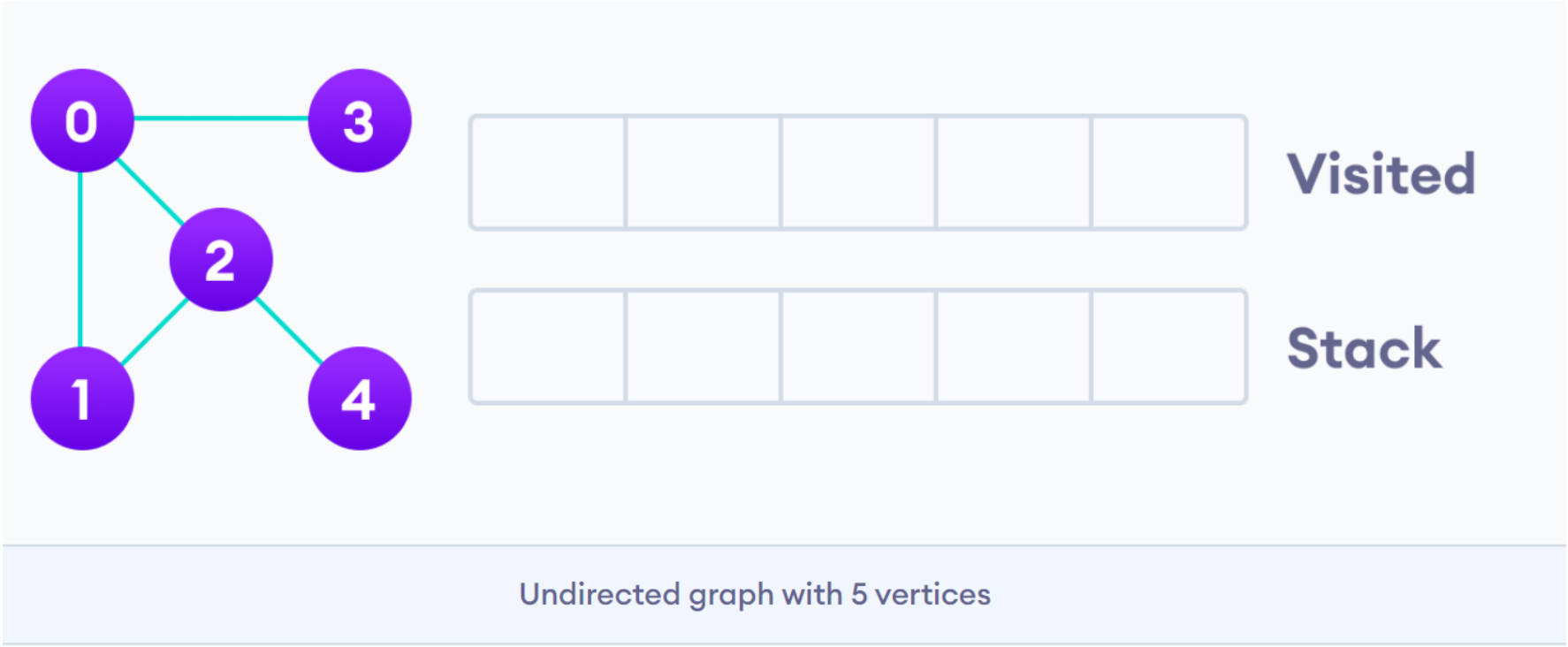
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

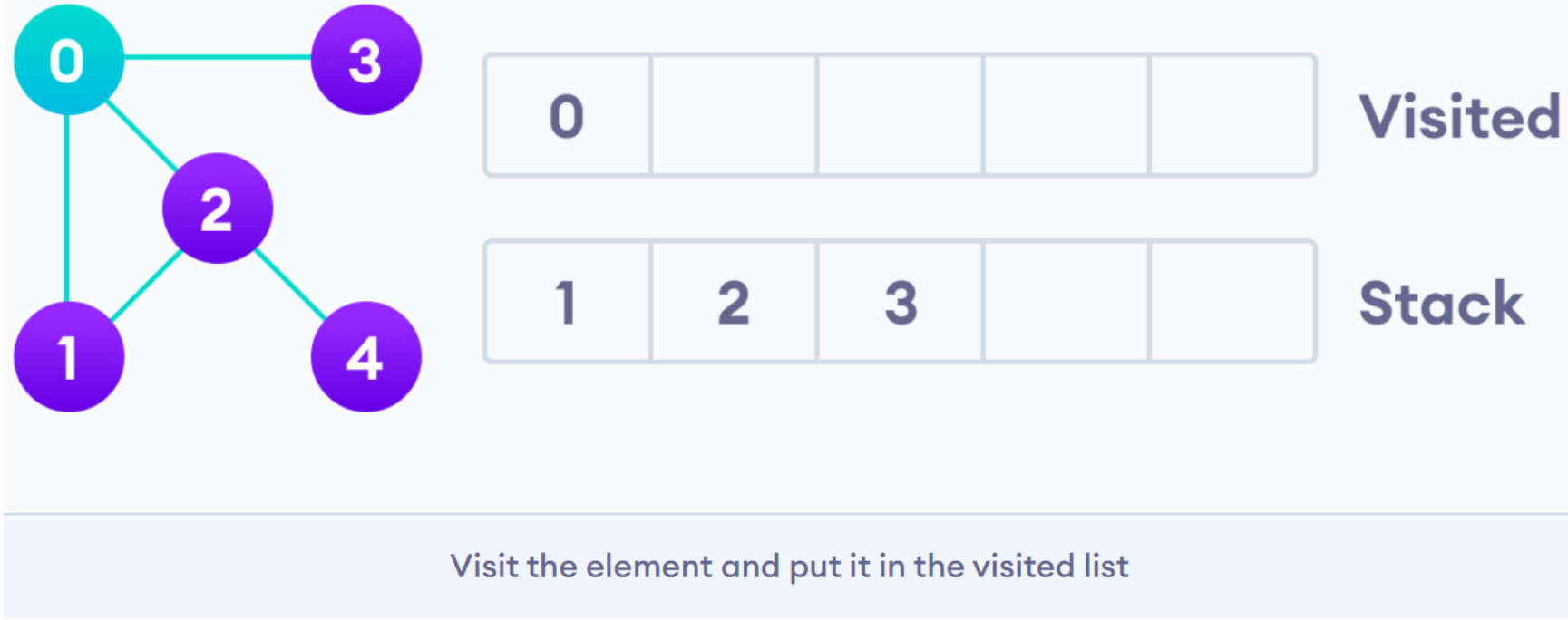
DFS example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.

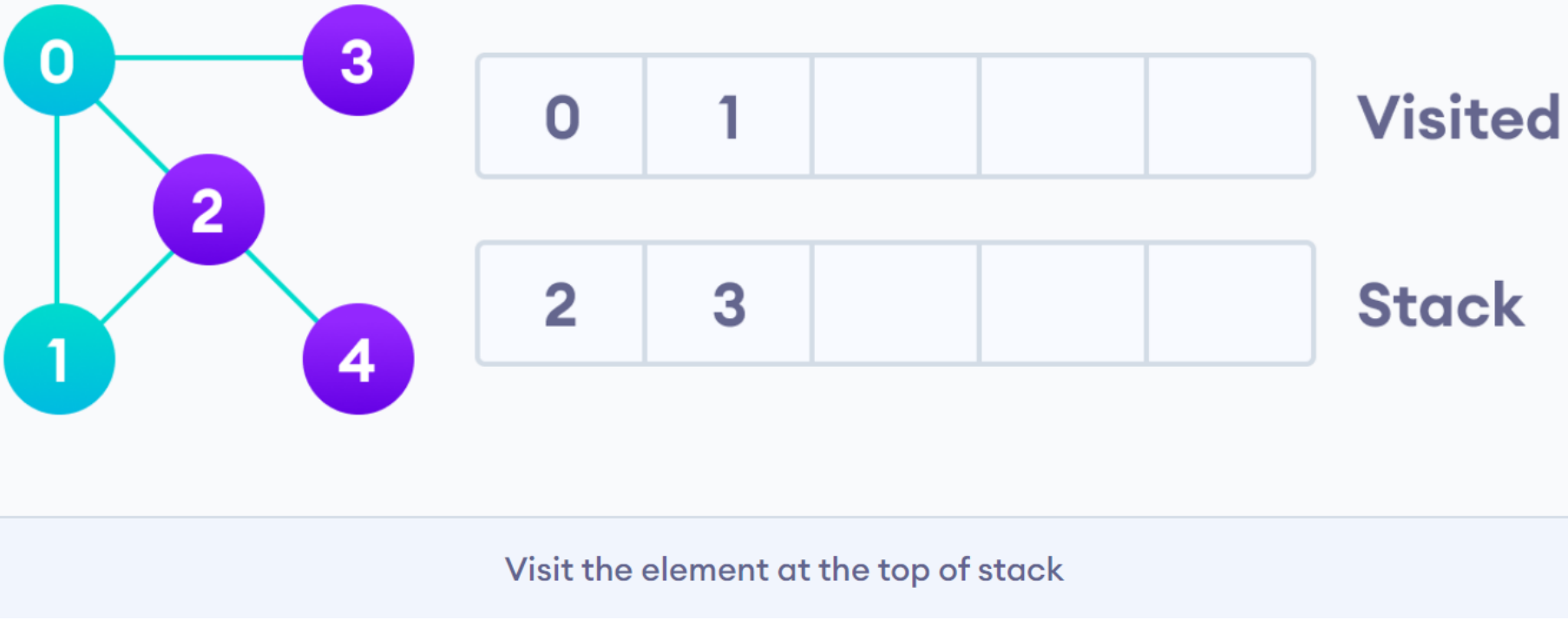


Undirected graph with 5 vertices

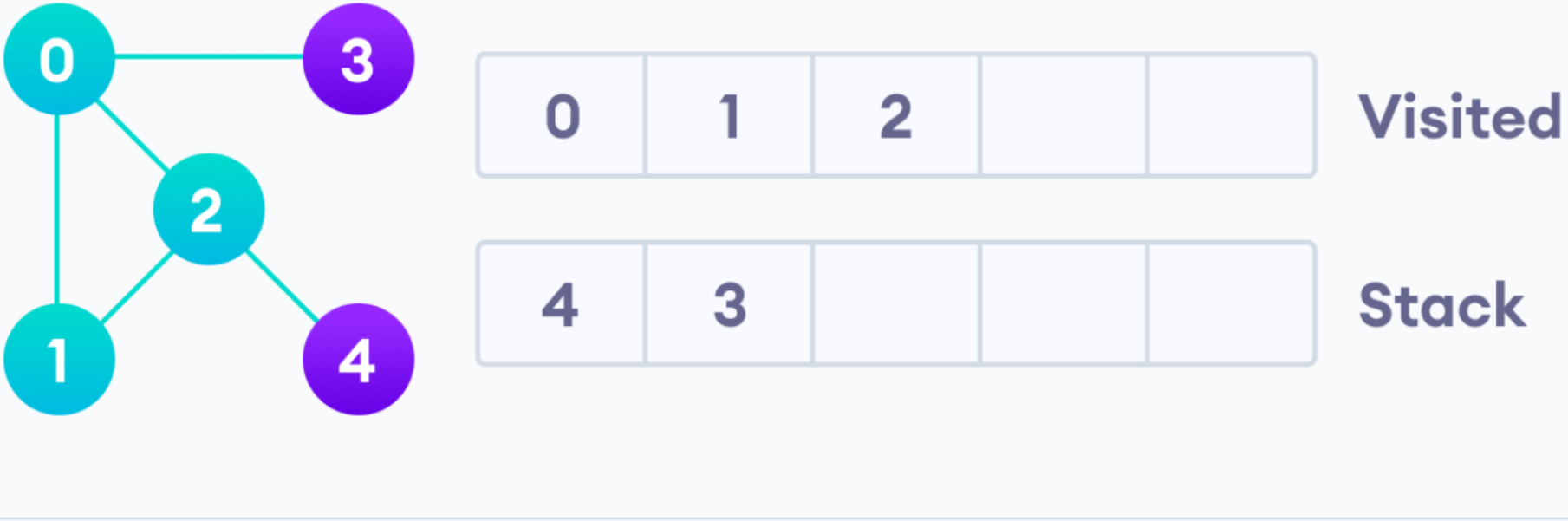
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



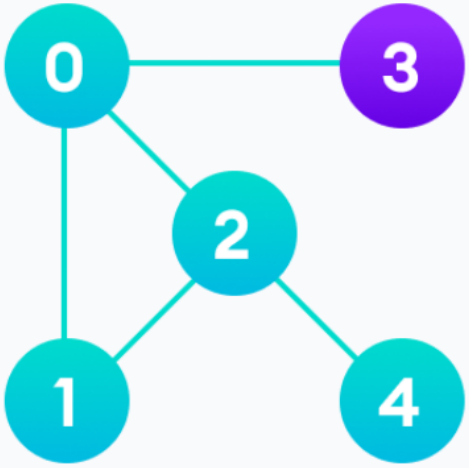
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



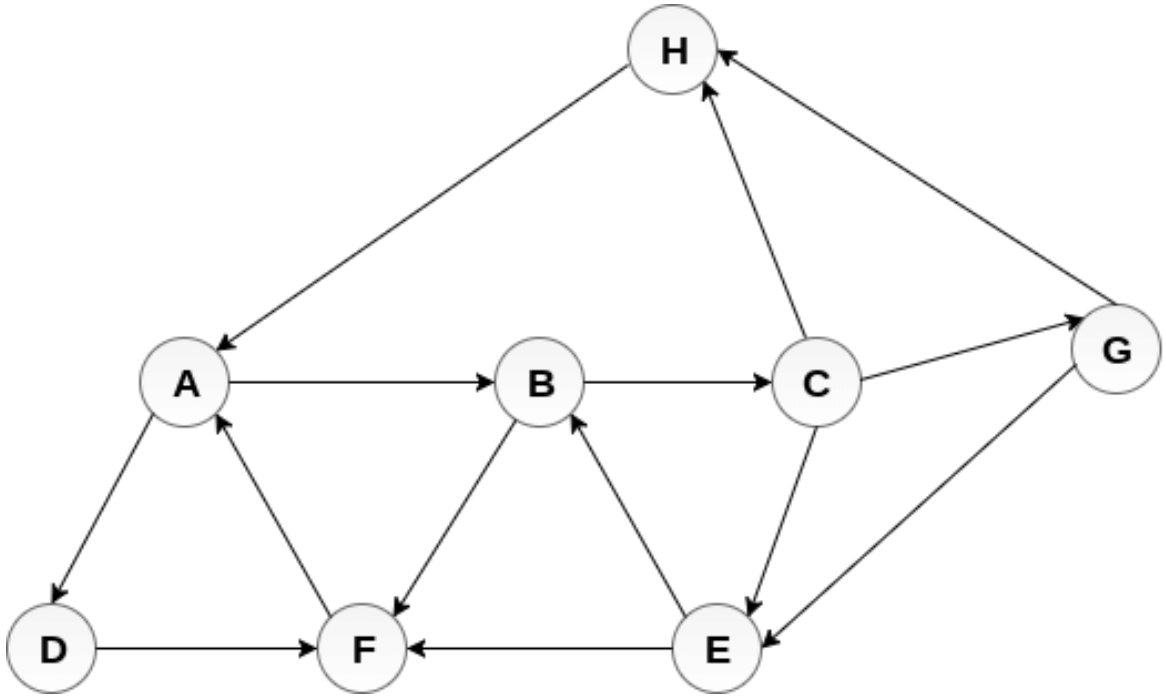
0	1	2	4		Visited
3					Stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F

F : A

D : F

H : A

1. PUSH ‘H’ ONTO THE STACK

STACK : H

2. POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

Print H

STACK : A

3. Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

Print A

Stack : B, D

4. Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

Print D

Stack : B, F

5. Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

Print F

Stack : B

6. Pop the top of the stack i.e. B and push all the neighbours

Print B

Stack : C

7. Pop the top of the stack i.e. C and push all the neighbours.

Print C

Stack : E, G

8. Pop the top of the stack i.e. G and push all its neighbours.

Print G

Stack : E

9. Pop the top of the stack i.e. E and push all its neighbours.

Print E

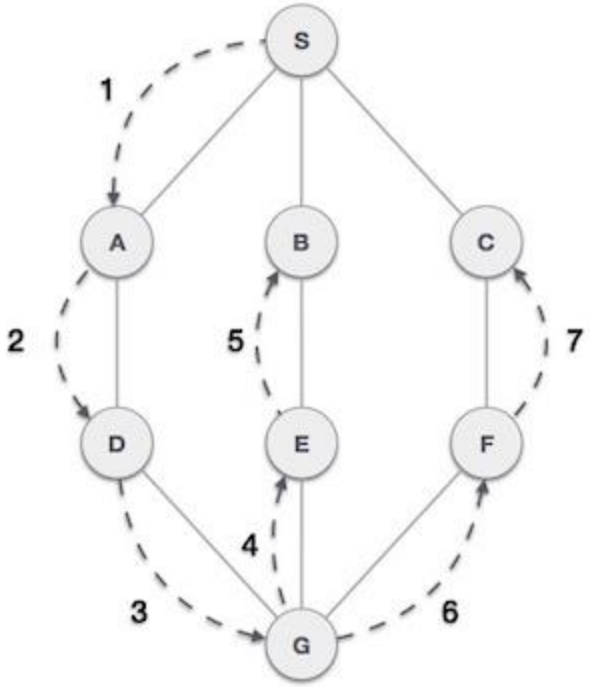
Stack :

10. Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

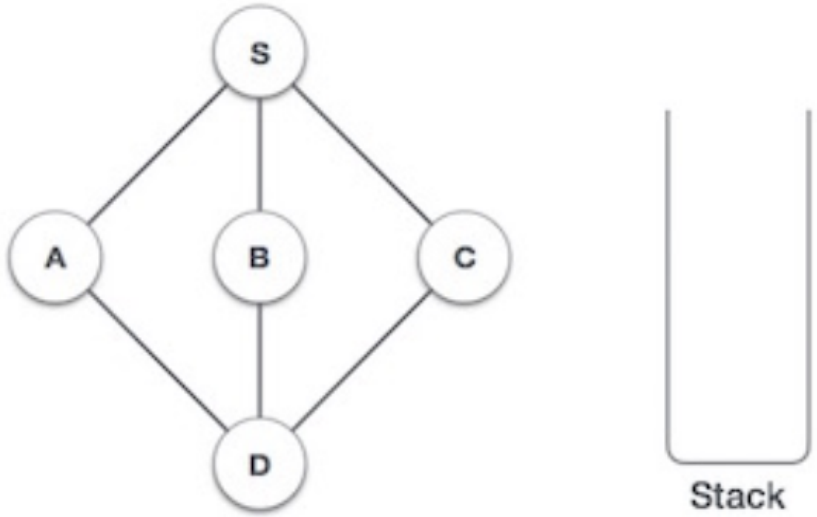
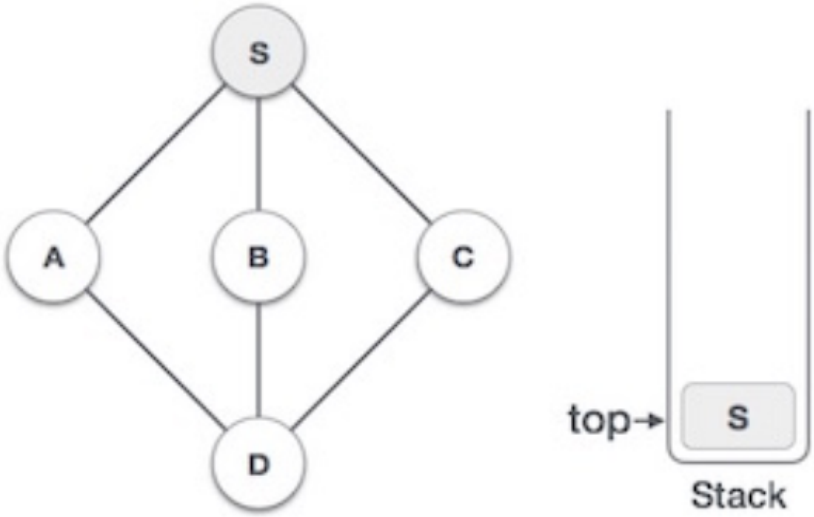


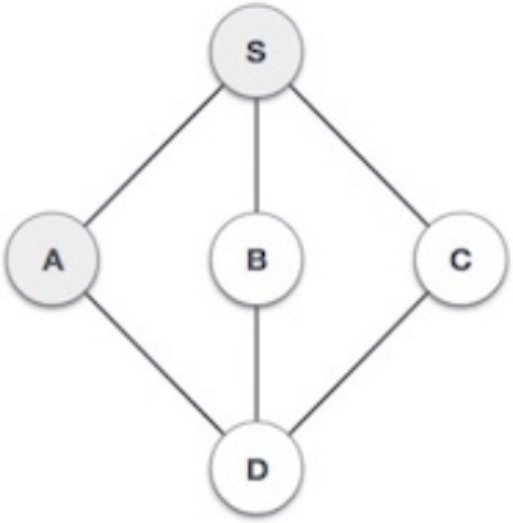
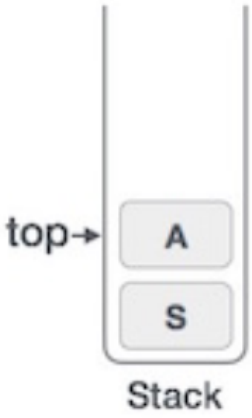
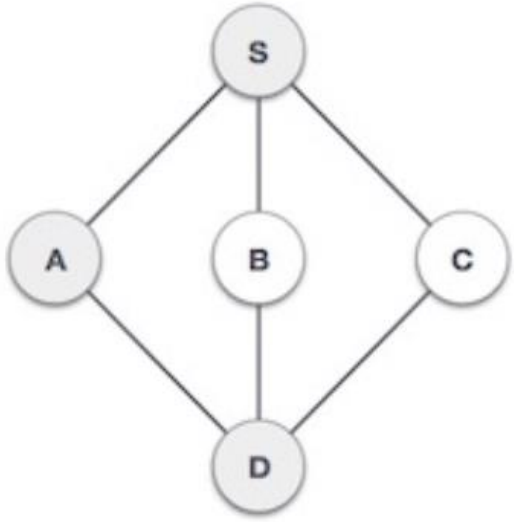
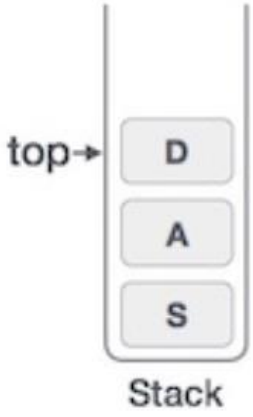
In the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

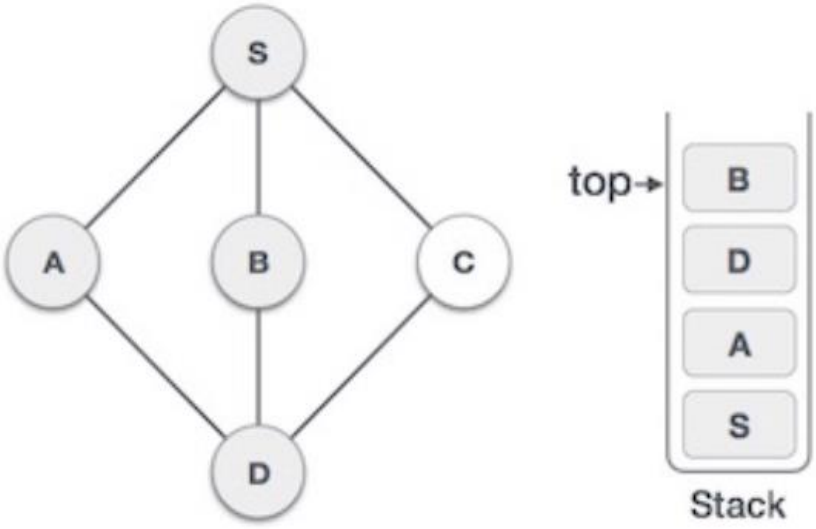
Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

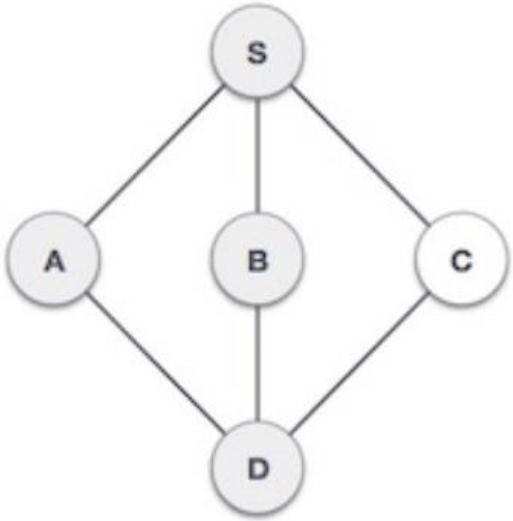
3			Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4			Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

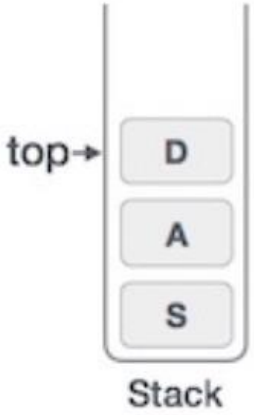
5



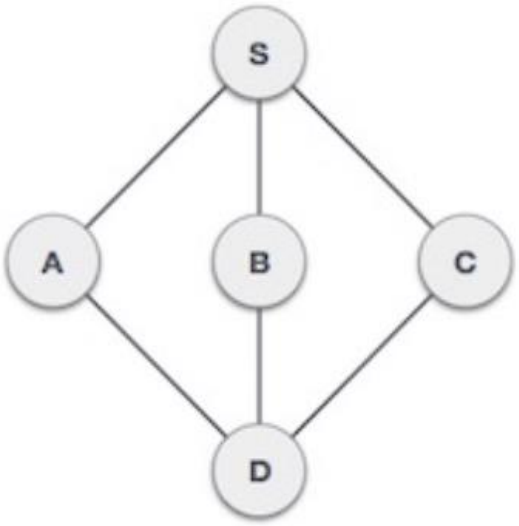
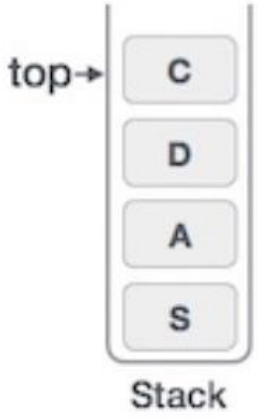
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6





We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7			Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

DFS Algorithm Applications

- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

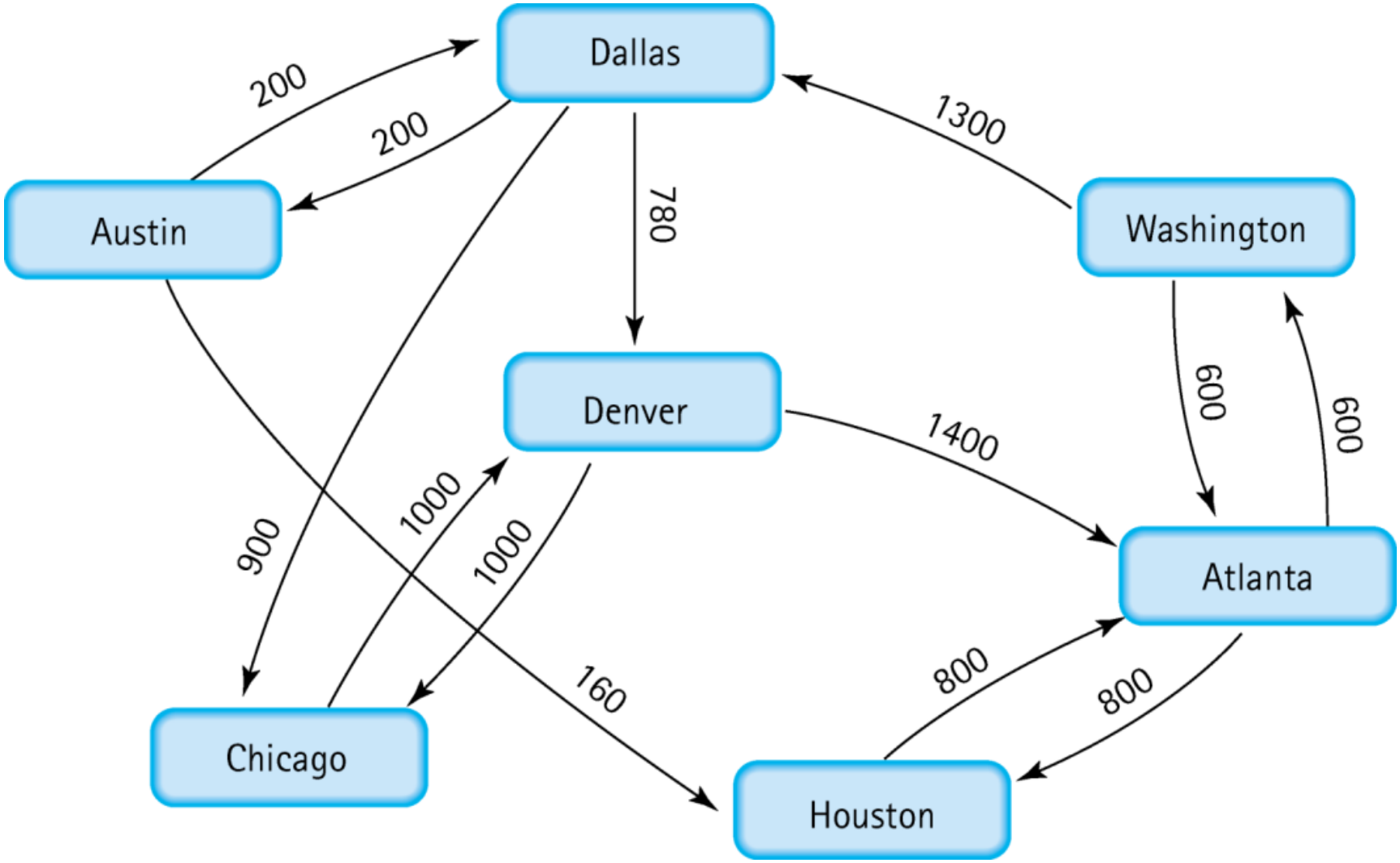
Depth-First-Search (DFS)

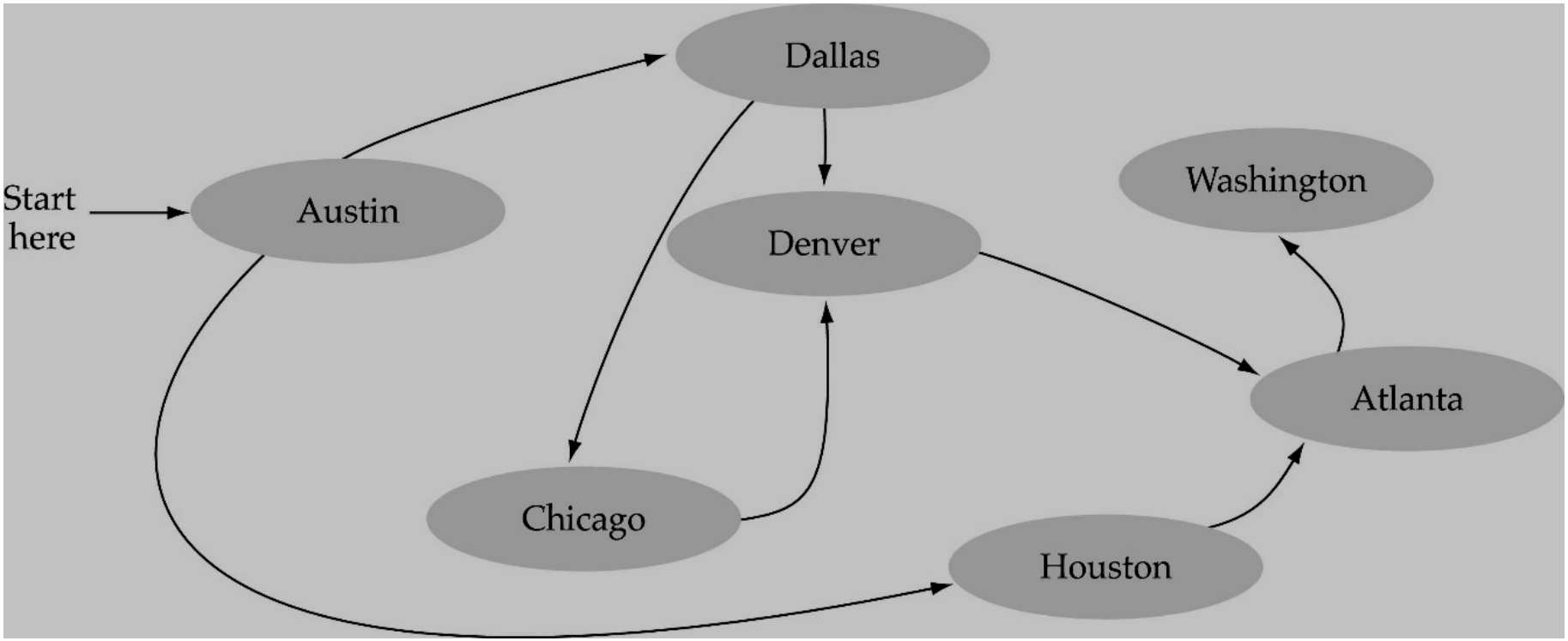
Main idea:

Travel as far as you can down a path

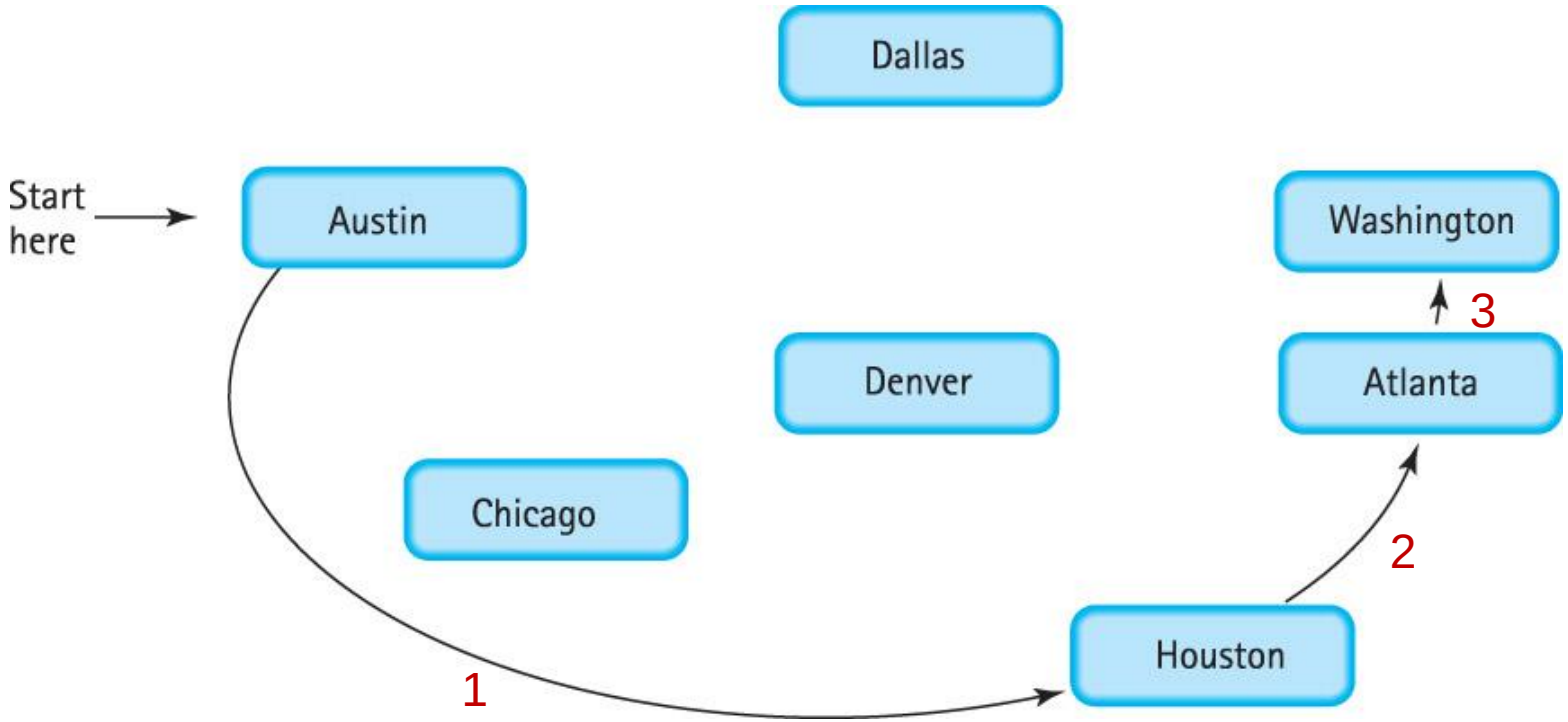
Back up **as little as possible** when you reach a "dead end“

i.e., next vertex has been "marked" or there is no next vertex


Dallas
Austin
Washington
Denver
Atlanta
Chicago
Houston
200
200
780
1300
600
600
1400
1000
1000
900
160
800
800


Dallas
Start here
Austin
Denver
Washington
Atlanta
Chicago
Houston

Depth First Search: Follow Down



DFS uses Stack !

graph

.numVertices 7

.vertices

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

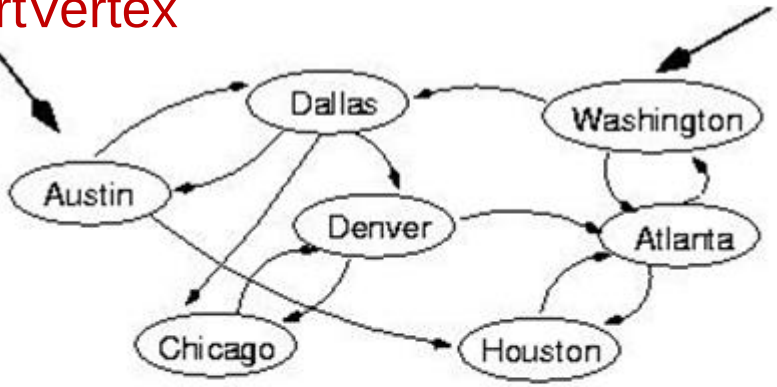
.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[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]	•	•	•	•	•	•	•	•	•	•

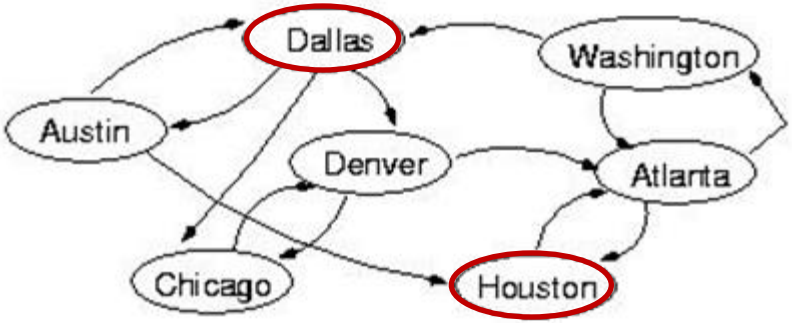
(Array positions marked '•' are undefined)

startVertex

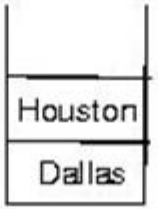
endVertex

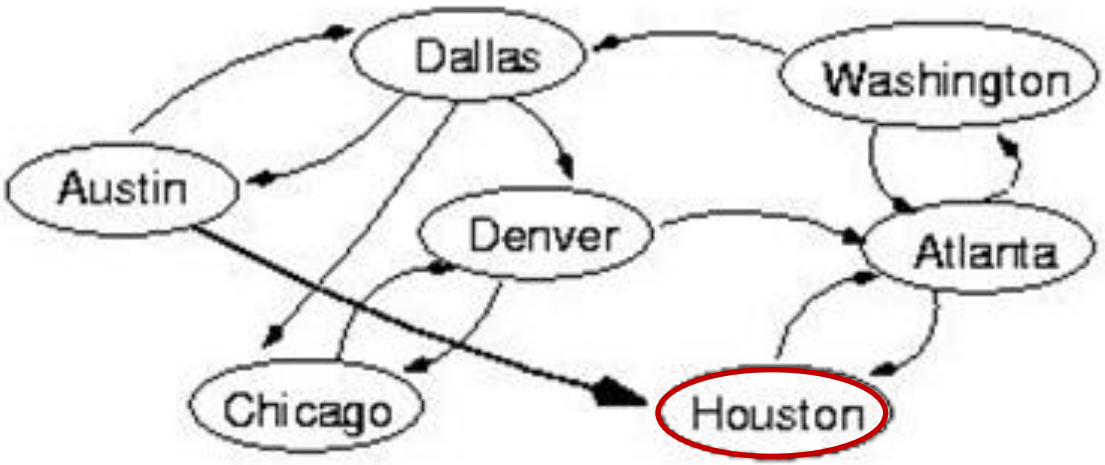


(initialization)

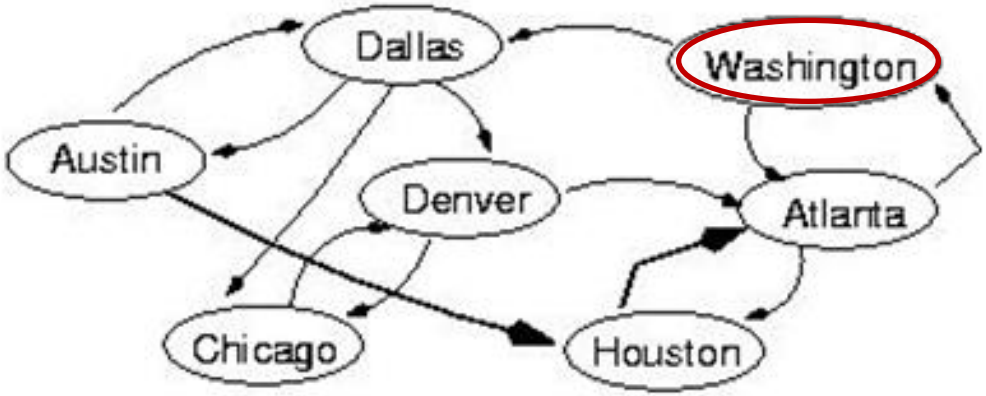


pop Austin



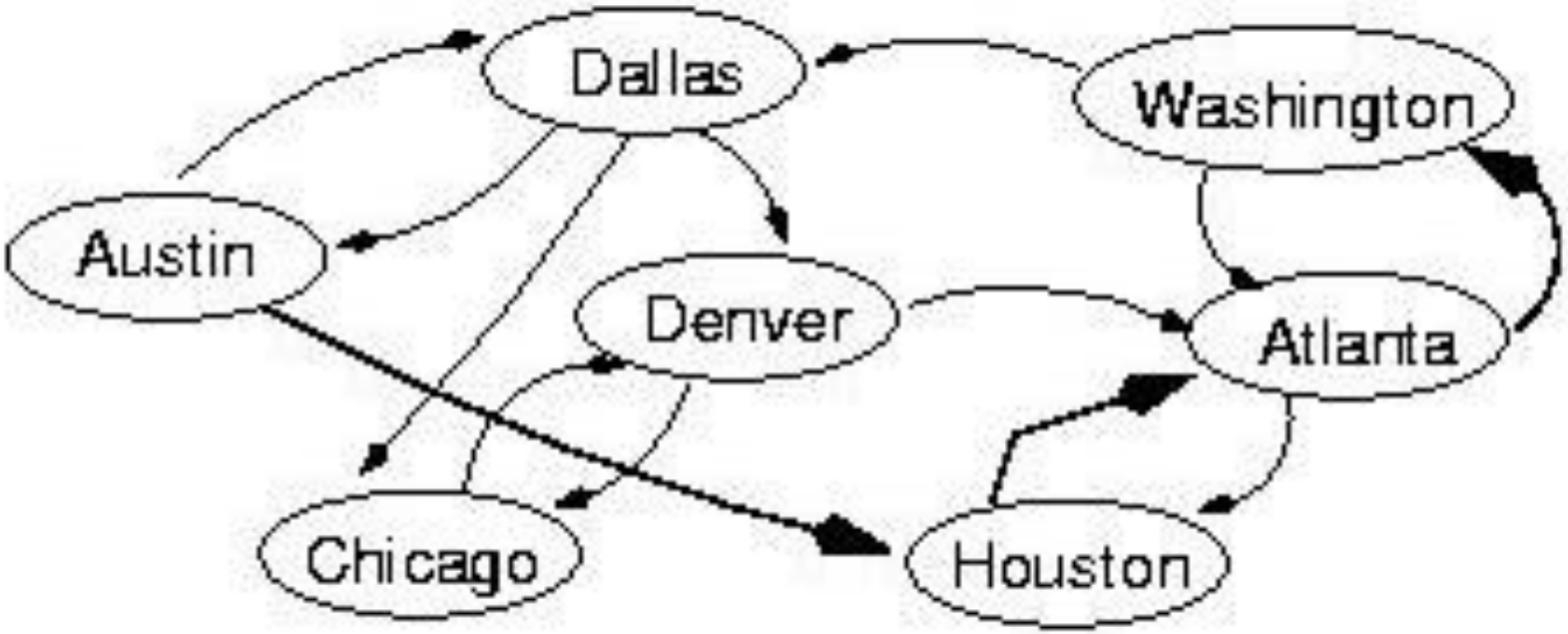


pop Houston

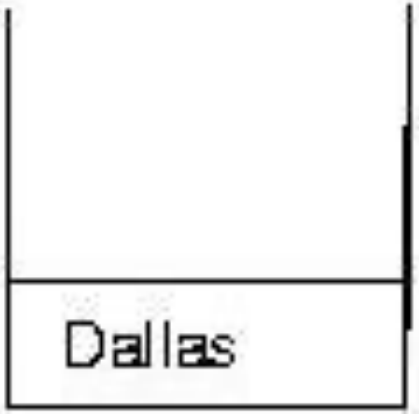


pop Atlanta

endVertex



pop Washington

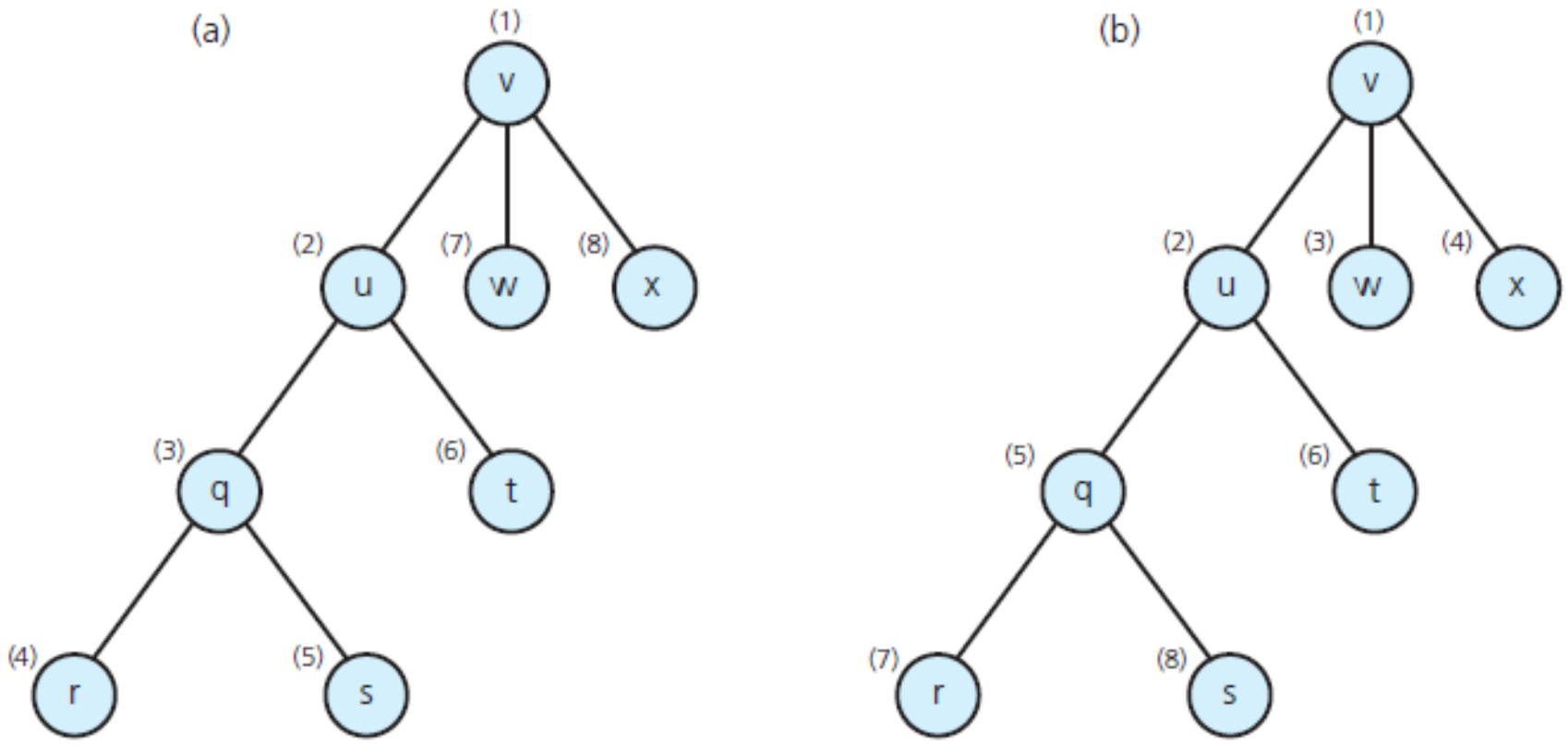


Depth-First Search

Visitation order for

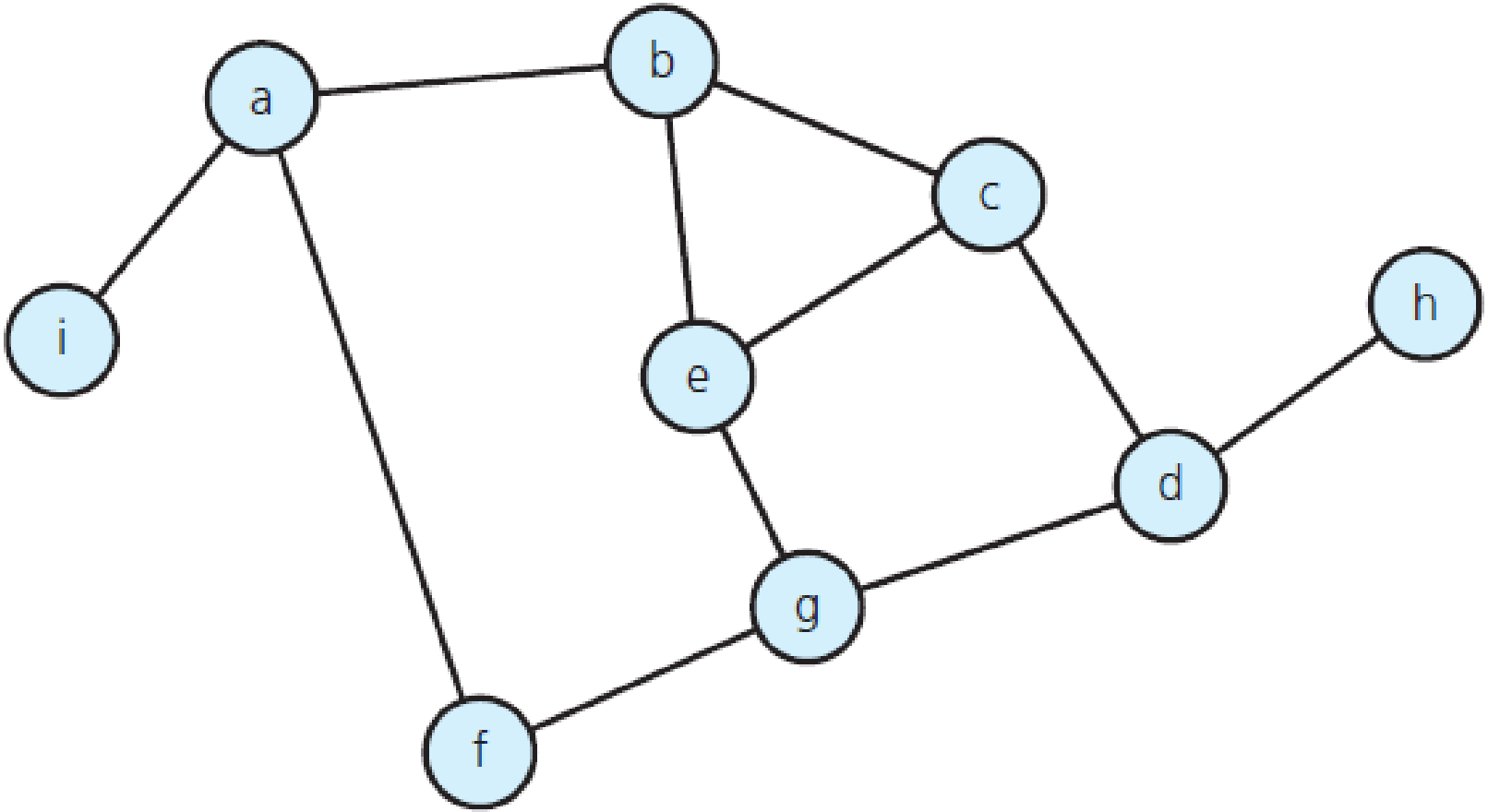
(a) a depth-first search;

(b) a breadth-first search


(a)
(1)
v
(2)
u
(7)
w
(8)
x
(3)
q
(6)
t
(4)
r
(5)
s
(b)
(1)
v
(2)
u
(3)
w
(4)
x
(5)
q
(6)
t
(7)
r
(8)
s

Depth-First Search

- A connected graph with cycles



Depth-First Search

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

The results of a depth-first traversal, beginning at vertex a, of the graph

Breadth-First-Searching (BFS)

Breadth first traversal or Breadth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

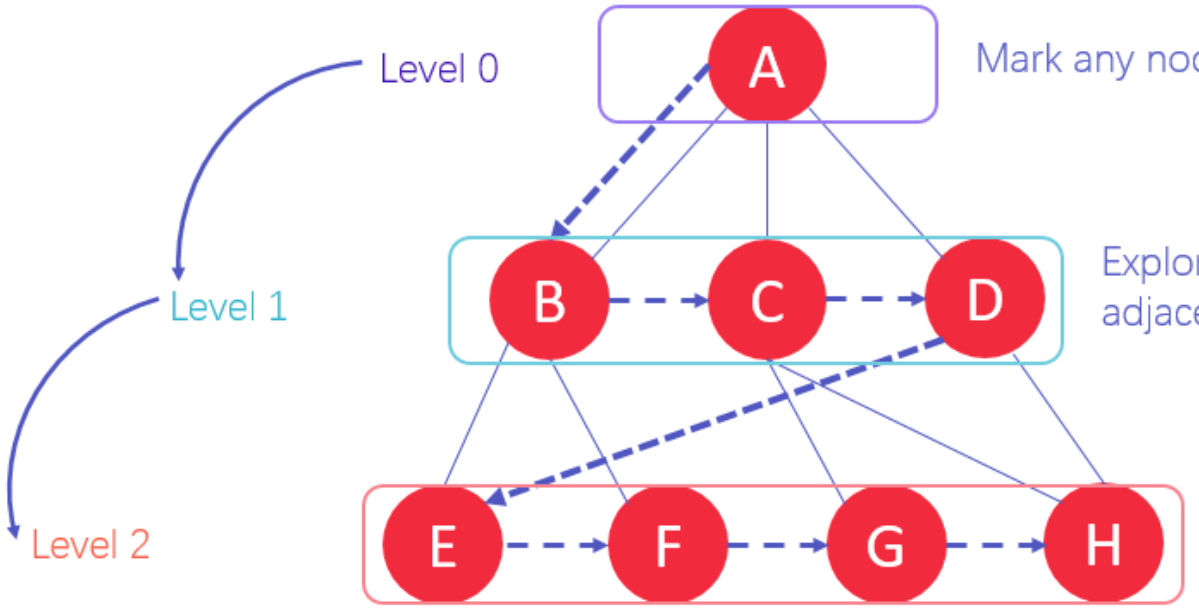
- Visited
- Not Visited

Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the **Breadth-first search**.

The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Remember, BFS accesses these nodes one by one.

Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them. Once visited, all nodes are marked. These iterations continue until all the nodes of the graph have been successfully visited and marked.

CONCEPT DIAGRAM



Mark any node as Starter or Initial 1

Explore and traverse un-visited nodes adjacent to starting node 2

Mark Node as Completed and move to next adjacent and un-visited nodes 3

Why do we need BFS Algorithm?

There are numerous reasons to utilize the BFS Algorithm to use as searching for your dataset. Some of the most vital aspects that make this algorithm your first choice are:

- BFS is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.
- BFS can traverse through a graph in the smallest number of iterations.
- The architecture of the BFS algorithm is simple and robust.
- The result of the BFS algorithm holds a high level of accuracy in comparison to other algorithms.
- BFS iterations are seamless, and there is no possibility of this algorithm getting caught up in an infinite loop problem.

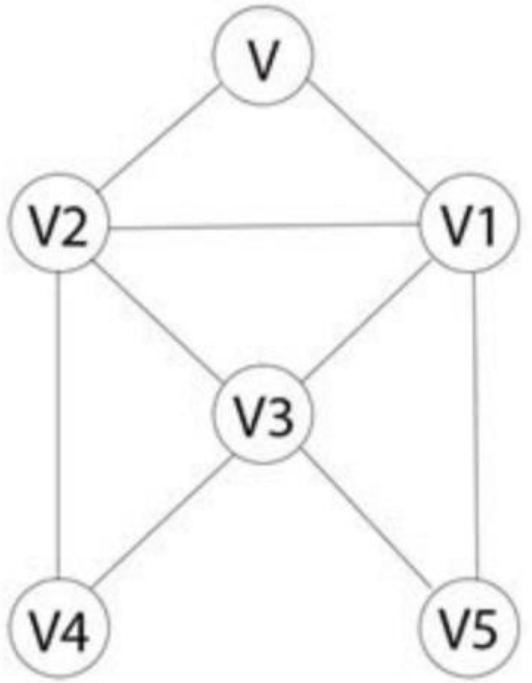
How does BFS Algorithm Work?

Graph traversal requires the algorithm to visit, check, and/or update every single un-visited node in a tree-like structure. Graph traversals are categorized by the order in which they visit the nodes on the graph.

BFS algorithm starts the operation from the first or starting node in a graph and traverses it thoroughly. Once it successfully traverses the initial node, then the next non-traversed vertex in the graph is visited and marked.

Hence, you can say that all the nodes adjacent to the current vertex are visited and traversed in the first iteration. A simple queue methodology is utilized to implement the working of a BFS algorithm, and it consists of the following steps:

Step 2)



In case V is not visited
add it to BFS queue

Queue = | V | |

In case the vertex V is not accessed then add the vertex V into the BFS Queue

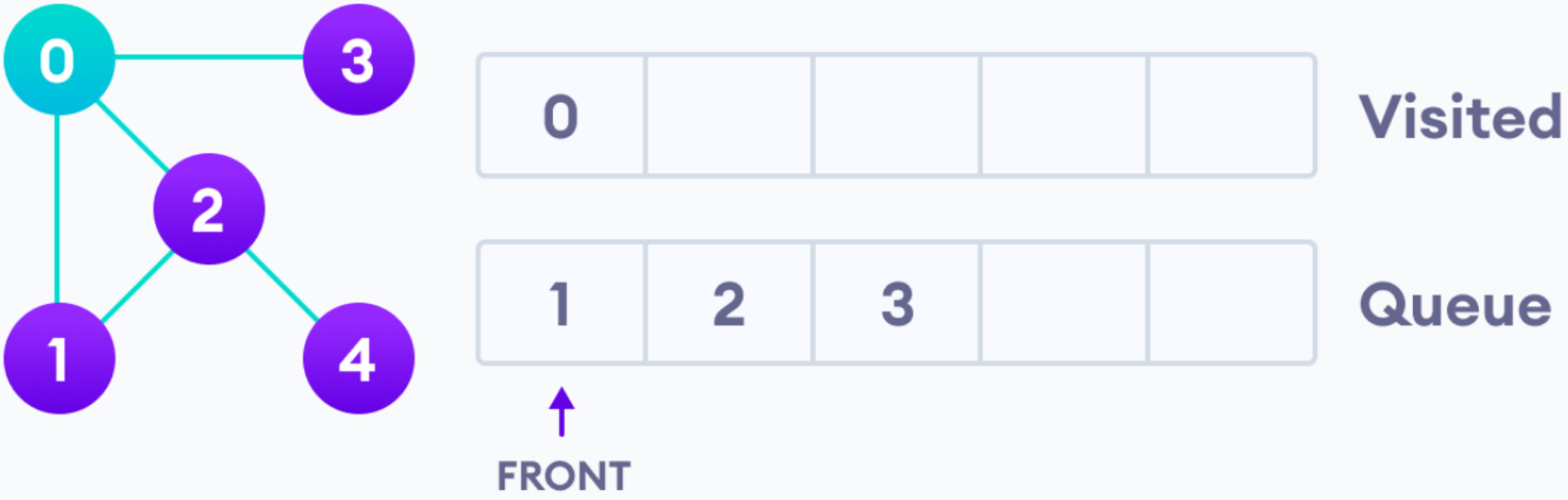
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

- Start by putting any one of the graph's vertices at the back of a queue.
- Take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
- Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

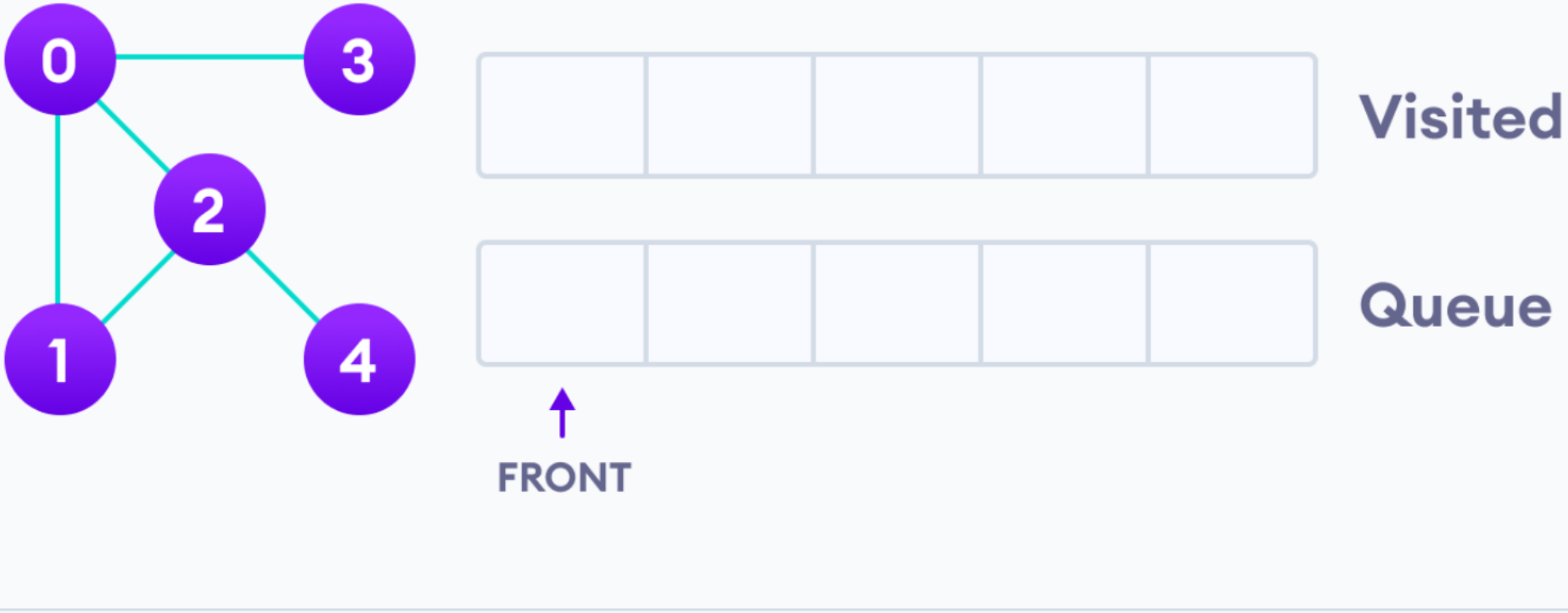
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Visit start vertex and add its adjacent vertices to queue

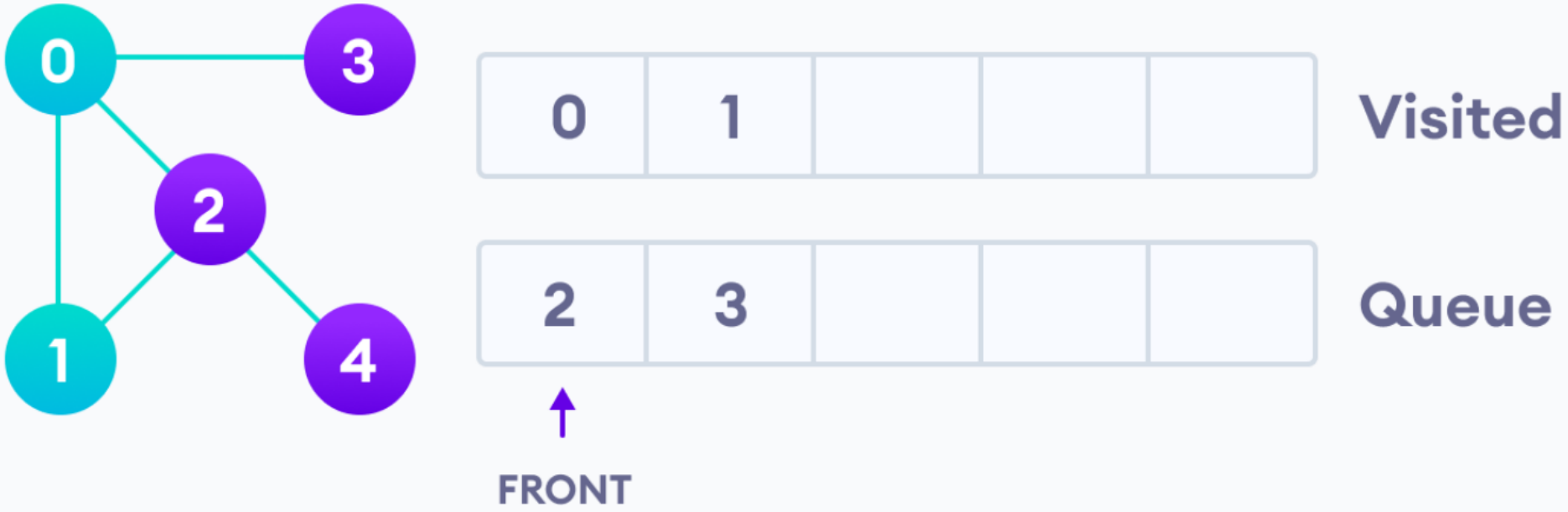
BFS example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



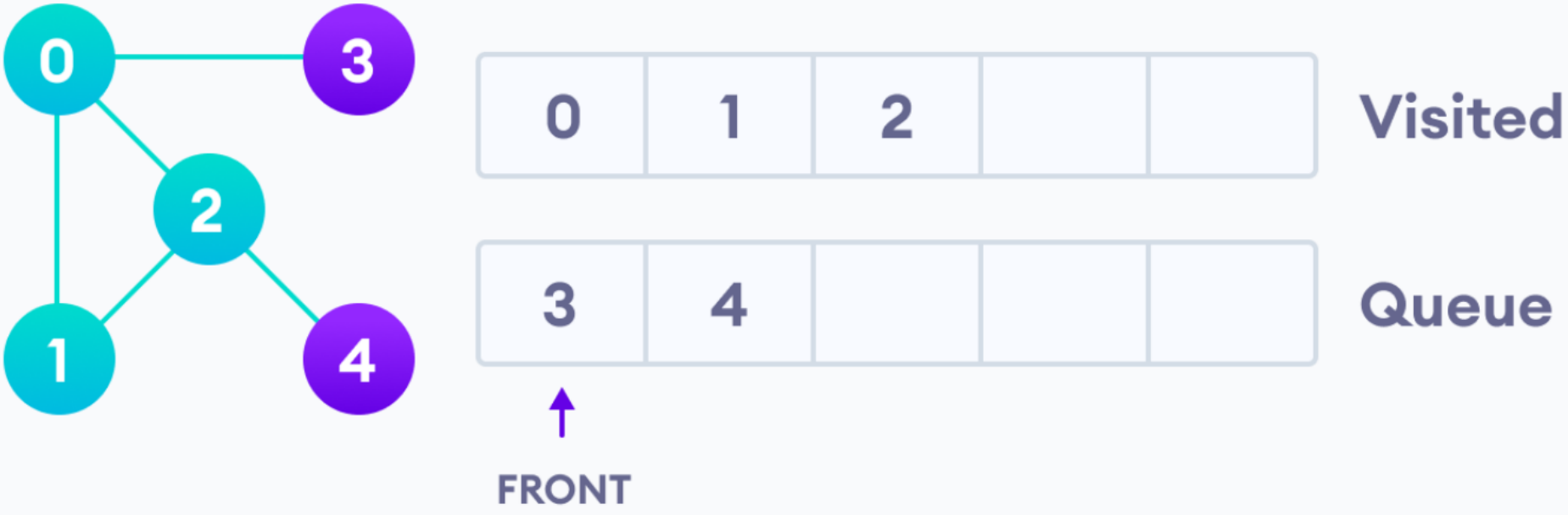
Undirected graph with 5 vertices

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

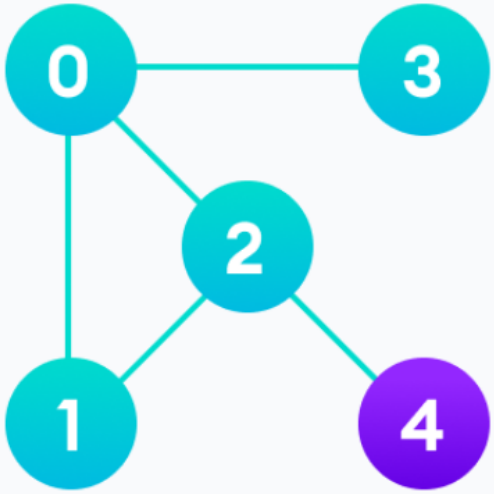


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Visit 2 which was added to queue earlier to add its neighbours



0	1	2	3	

Visited

4				

Queue

FRONT

4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.

Visit last remaining item in stack to check if it has unvisited neighbours

Since the queue is empty, we have completed the Depth First Traversal of the graph.

In the various levels of the data, you can mark any node as the starting or initial node to begin traversing. The BFS will visit the node and mark it as visited and places it in the queue.

Now the BFS will visit the nearest and un-visited nodes and marks them. These values are also added to the queue. The queue works on the FIFO model.

In a similar manner, the remaining nearest and un-visited nodes on the graph are analyzed marked and added to the queue. These items are deleted from the queue as receive and printed as the result.

In the various levels of the data, you can mark any node as the starting or initial node to begin traversing. The BFS will visit the node and mark it as visited and places it in the queue.

Now the BFS will visit the nearest and un-visited nodes and marks them. These values are also added to the queue. The queue works on the FIFO model.

In a similar manner, the remaining nearest and un-visited nodes on the graph are analyzed marked and added to the queue. These items are deleted from the queue as receive and printed as the result.

Rules of BFS Algorithm

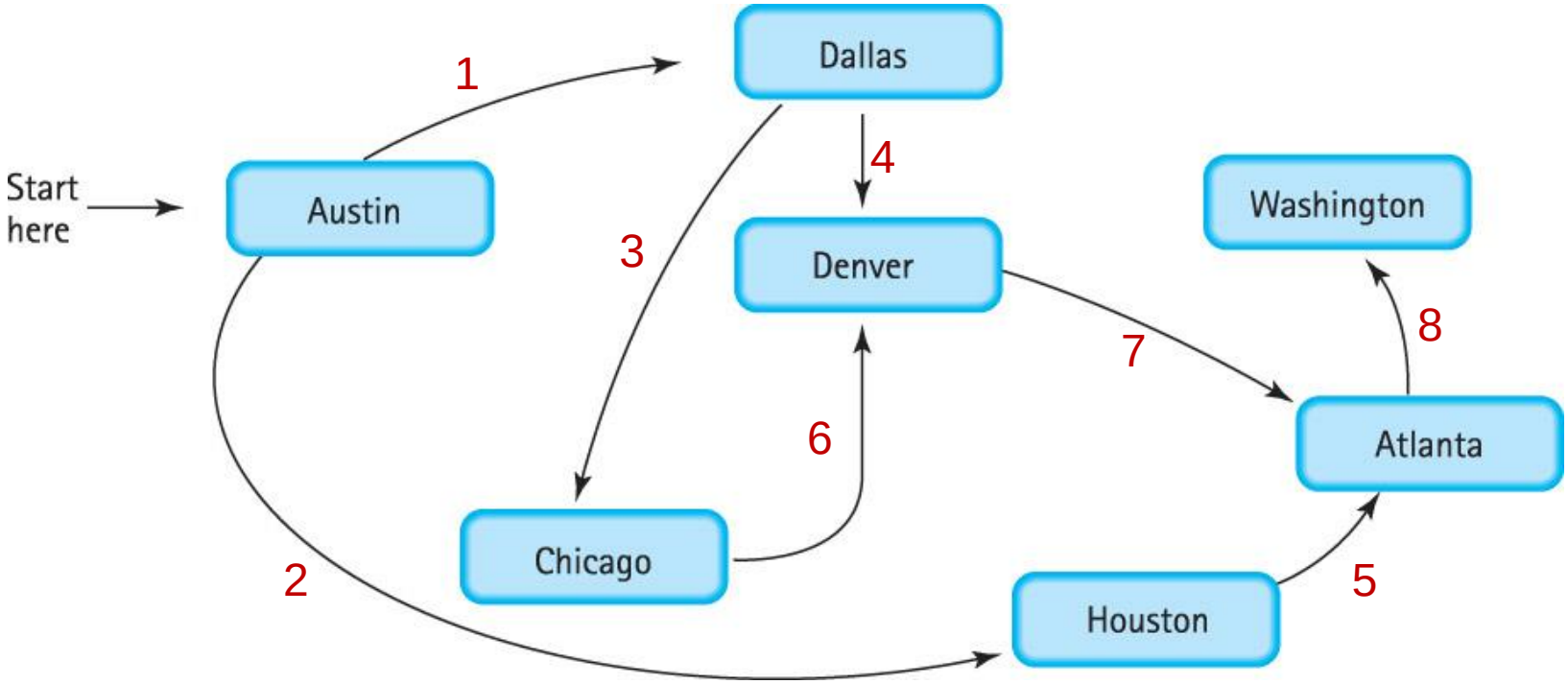
Here, are important rules for using BFS algorithm:

1. A queue (FIFO-First in First Out) data structure is used by BFS.
2. You mark any node in the graph as root and start traversing the data from it.
3. BFS traverses all the nodes in the graph and keeps dropping them as completed.
4. BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.
5. Removes the previous vertex from the queue in case no adjacent vertex is found.
6. BFS algorithm iterates until all the vertices in the graph are successfully traversed and marked as completed.
7. There are no loops caused by BFS during the traversing of data from any node.

Breadth-First-Searching (BFS)

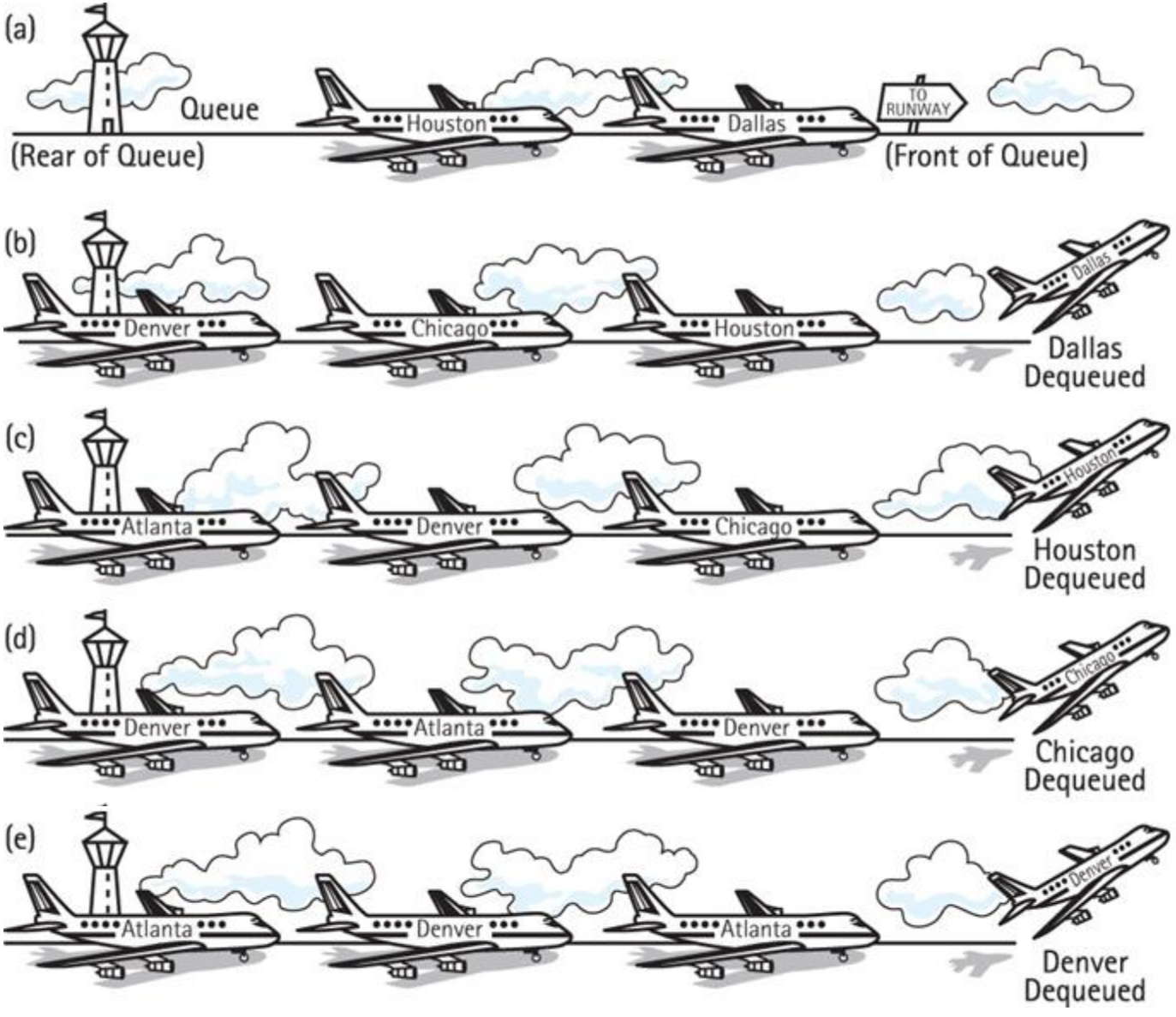
- Main idea:
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up **as far as possible** when you reach a "dead end“
    - i.e., next vertex has been "marked" or there is no next vertex

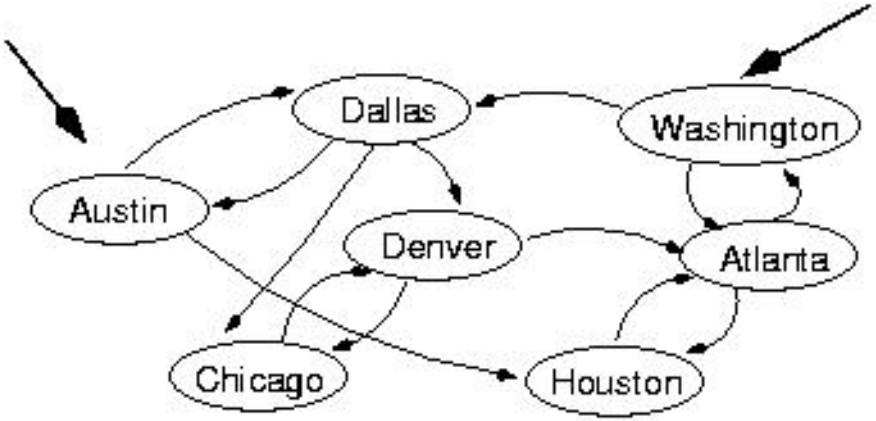
Breadth First: Follow Across



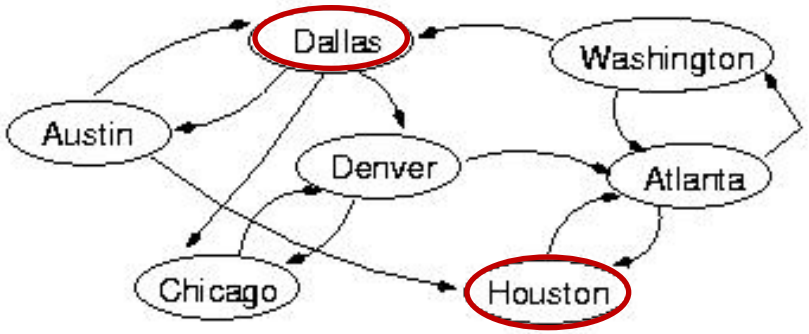
BFS uses Queue !

Breadth First Uses Queue

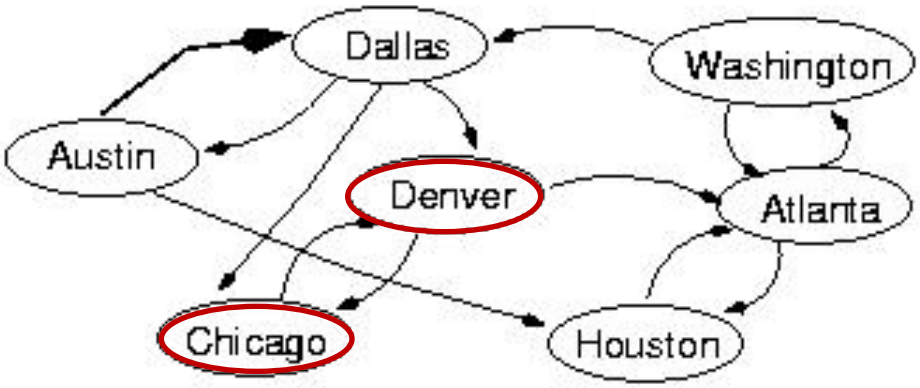




(initialization)

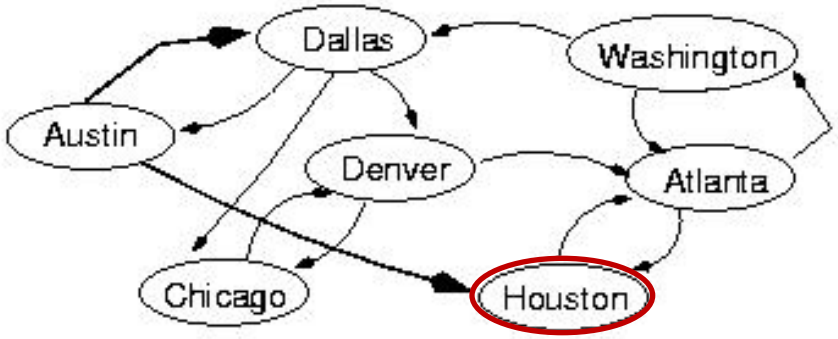


dequeue Austin



dequeue Dallas

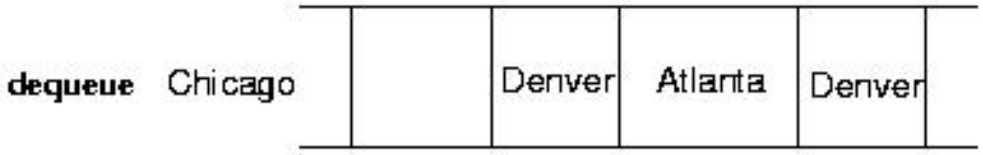
	Houston	Chicago	Denver

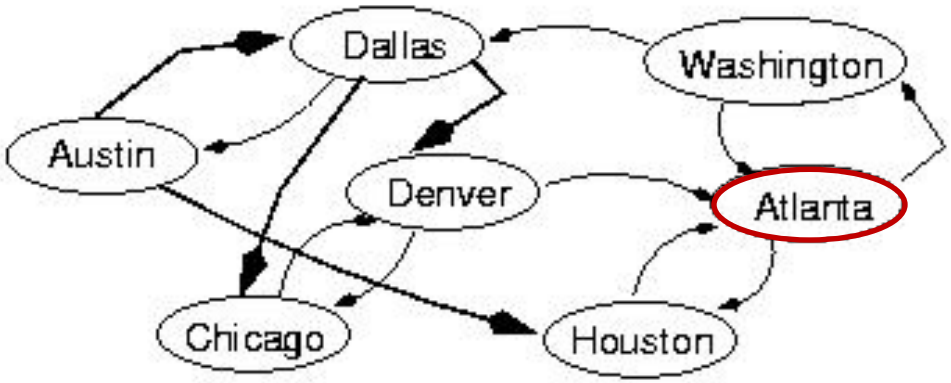


dequeue Houston

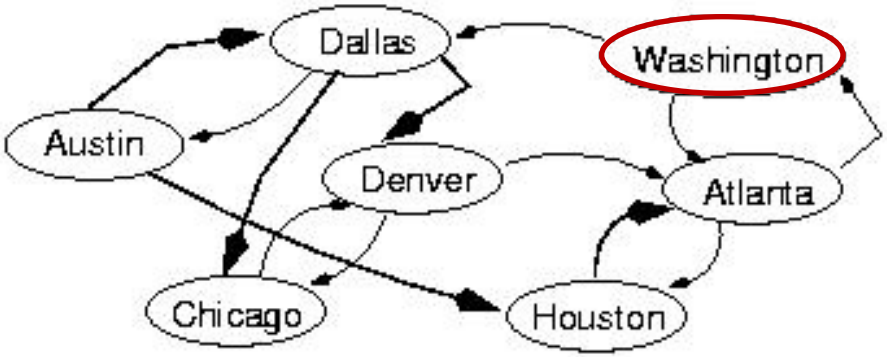
	Chicago	Denver	Atlanta


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston

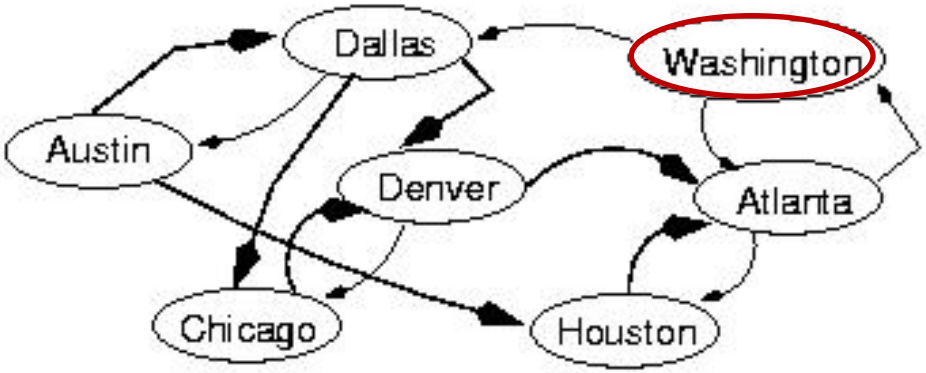

dequeue
Chicago
Denver
Atlanta
Denver


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston

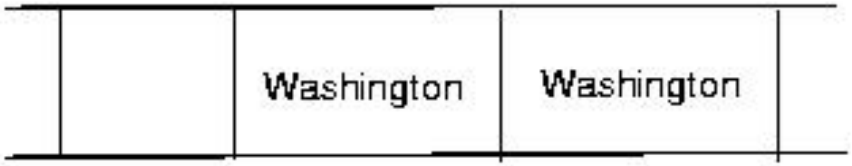

dequeue
Denver
Atlanta
Denver
Atlanta

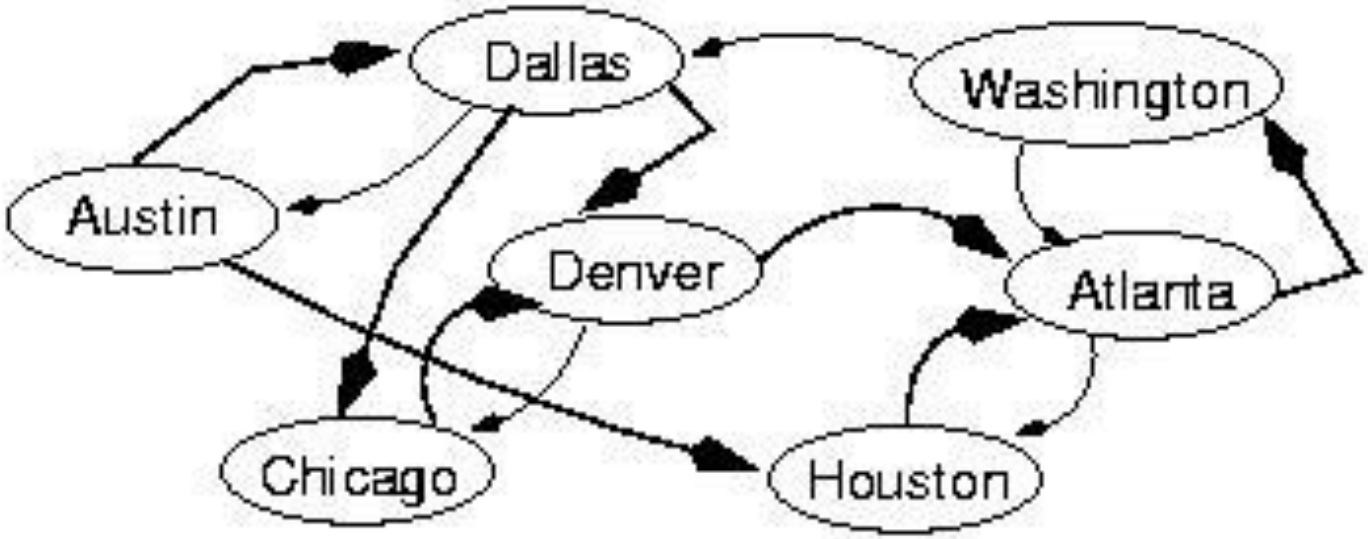


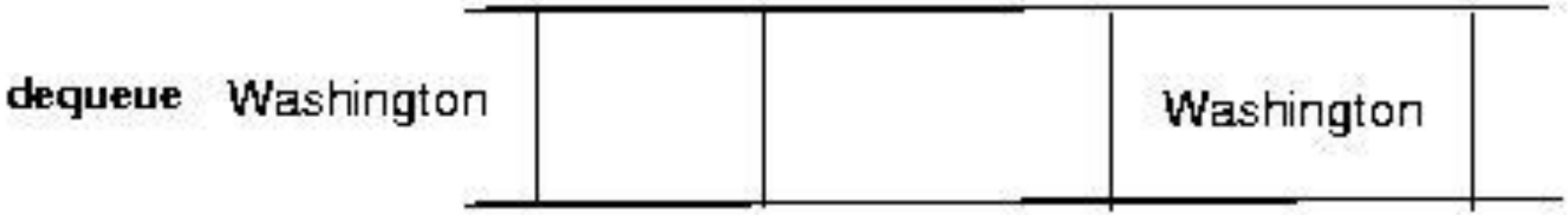
dequeue Atlanta



dequeue Denver, Atlanta




Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston


dequeue
Washington
Washington

BFS Algorithm Applications

- To build index by search index
- For GPS navigation
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph
- In minimum spanning tree

Applications of BFS Algorithm

Let's take a look at some of the real-life applications where a BFS algorithm implementation can be highly effective.

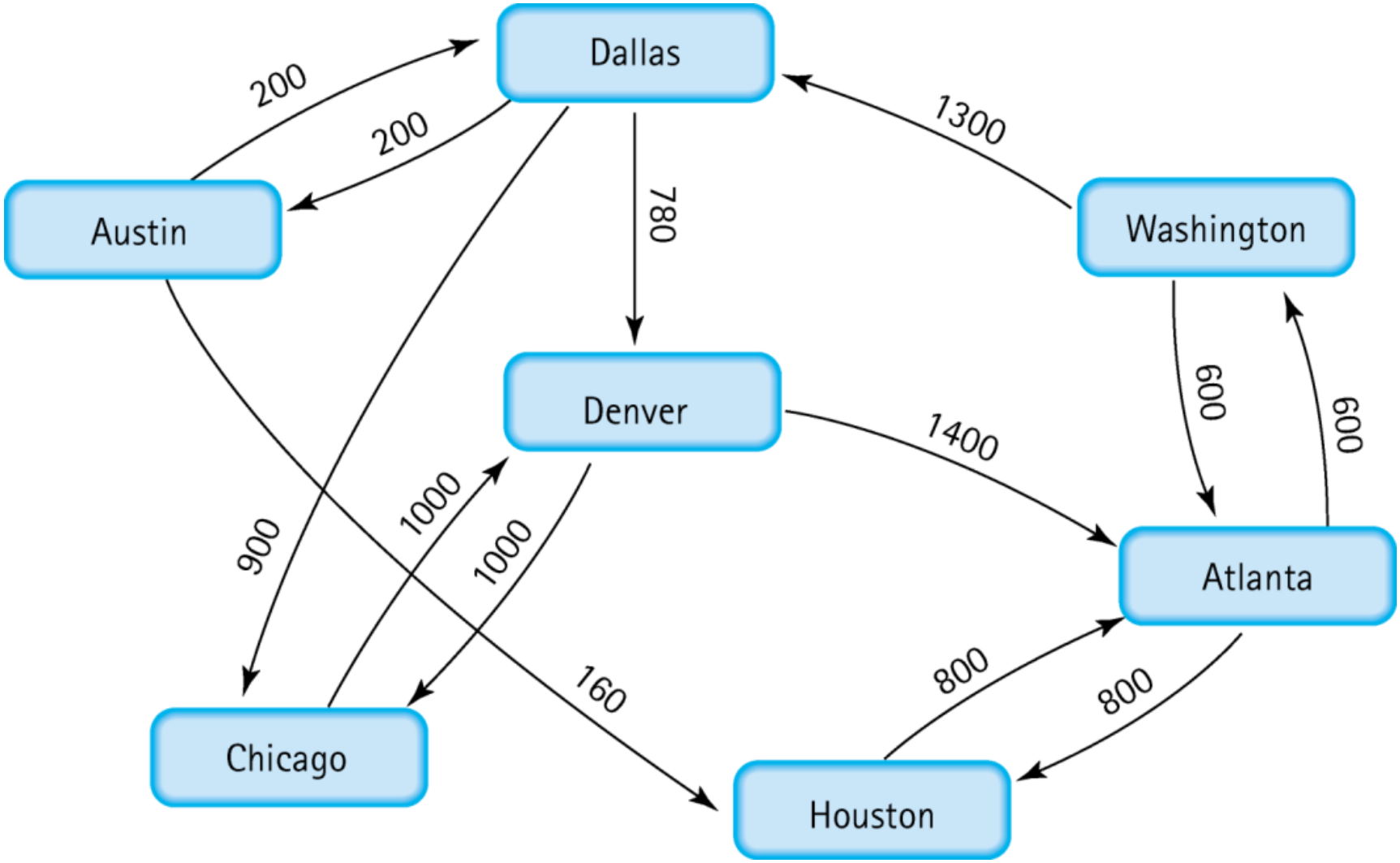
1. **Un-weighted Graphs:** BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.
2. **P2P Networks:** BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.
3. **Web Crawlers:** Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.
4. **Navigation Systems:** BFS can help find all the neighboring locations from the main or source location.
5. **Network Broadcasting:** A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

Variants of Shortest Path

- **Single-pair** shortest path
  - Find a shortest path from u to v
    - for given vertices u and v
- **Single-source** shortest paths
  - $G = (V, E) \Rightarrow$ find a shortest path from a given source vertex **s** to each vertex $v \in V$

Variants of Shortest Paths

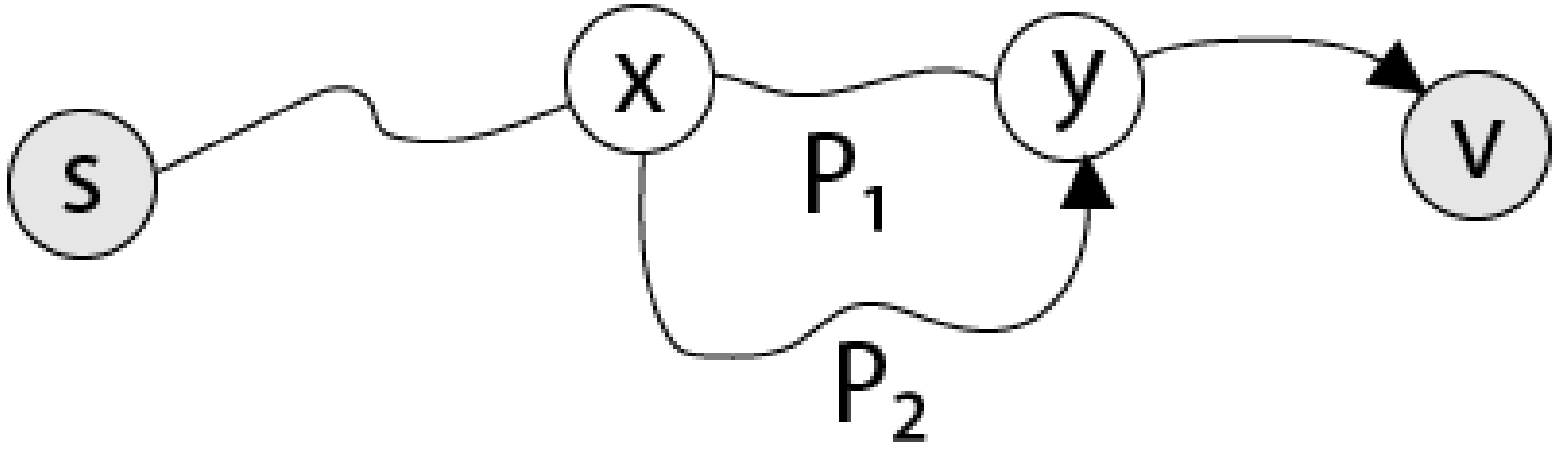
- **Single-destination** shortest paths
  - Find a shortest path to a given destination vertex **t** from each vertex **v**
  - Reversing the direction of each edge → single-source
- **All-pairs** shortest paths
  - Find a shortest path from **u** to **v** for every pair of vertices **u** and **v**


Dallas
Austin
Washington
Denver
Atlanta
Chicago
Houston
200
200
780
1300
600
600
1400
1000
1000
900
160
800
800

Single Source Shortest Path

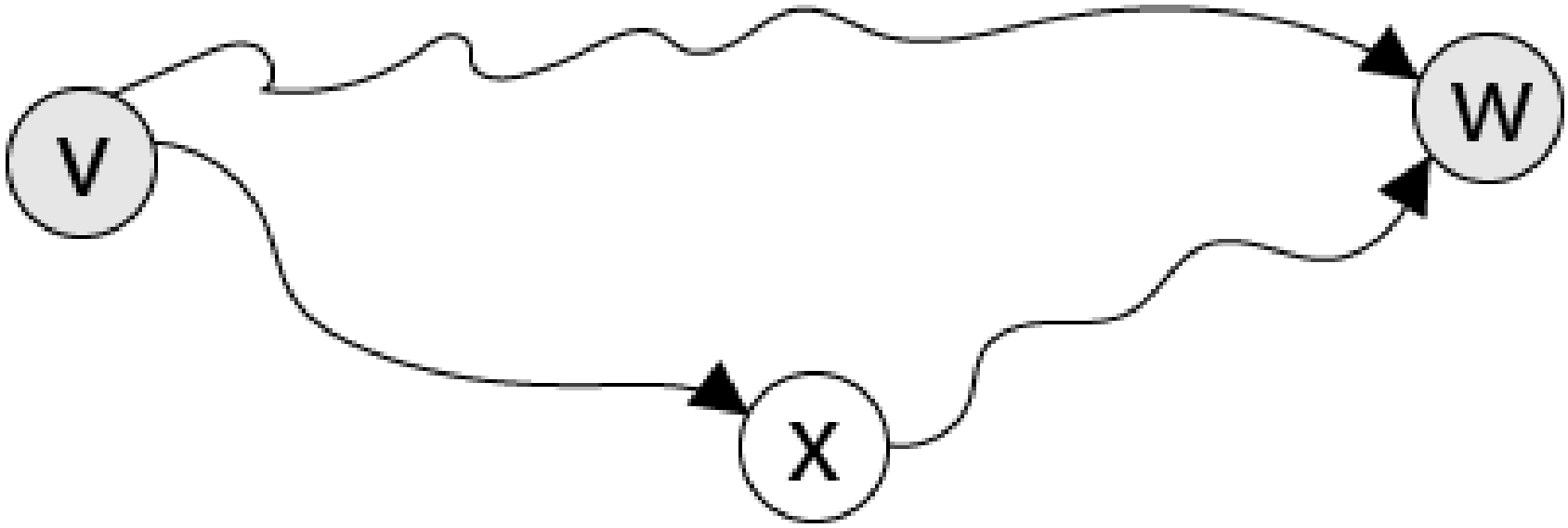
Properties of Shortest Path:

1. Optimal substructure property: All subpaths of shortest paths are shortest paths.



Let P_1 be x - y sub path of shortest s - v path. Let P_2 be any x - y path. Then cost of $P_1 \leq$ cost of P_2,otherwise P not shortest s - v path.

2. Triangle inequality: Let d (v, w) be the length of shortest path from v to w. Then, d (v, w) ≤ d (v, x) + d (x, w)



3. Upper-bound property: We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ conclude the value $\delta(s, v)$, it never changes.

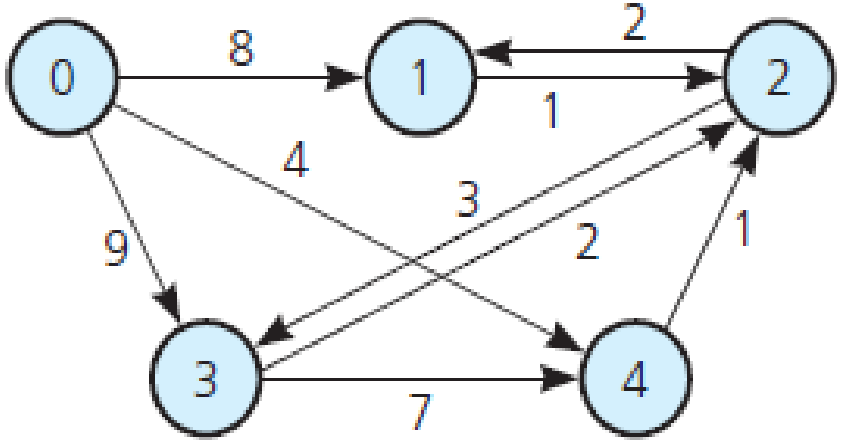
4. No-path property: If there is no path from s to v, then we regularly have $d[v] = \delta(s, v) = \infty$.

5. Convergence property: If s->u->v is a shortest path in G for some $u, v \in V$, and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times thereafter.

Shortest Paths

- Shortest path between two vertices in a weighted graph has smallest edge-weight sum

(a)



(b)

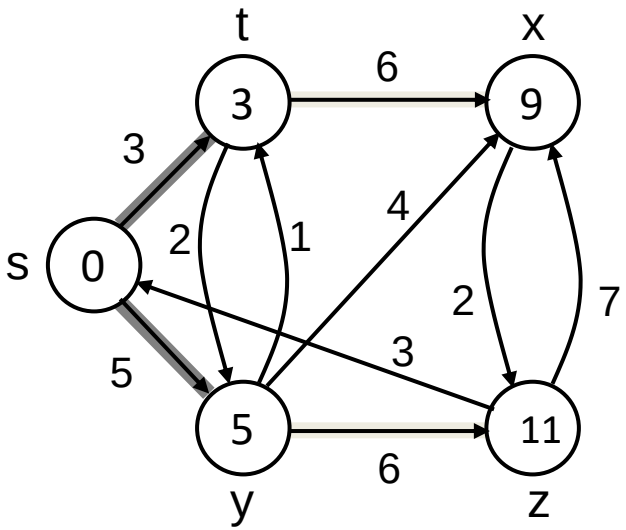
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

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

Notation

- **Weight of path** $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$



- **Shortest-path weight** from s to **v**:

$$\delta(v) = \begin{cases} \min\ w(p) : s \overset{p}{\rightsquigarrow} v & \text{if there exists a path from s to v} \\ \infty & \text{otherwise} \end{cases}$$

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.

Cost of C < 0

Dijkstra's Algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source. The graph has the following:

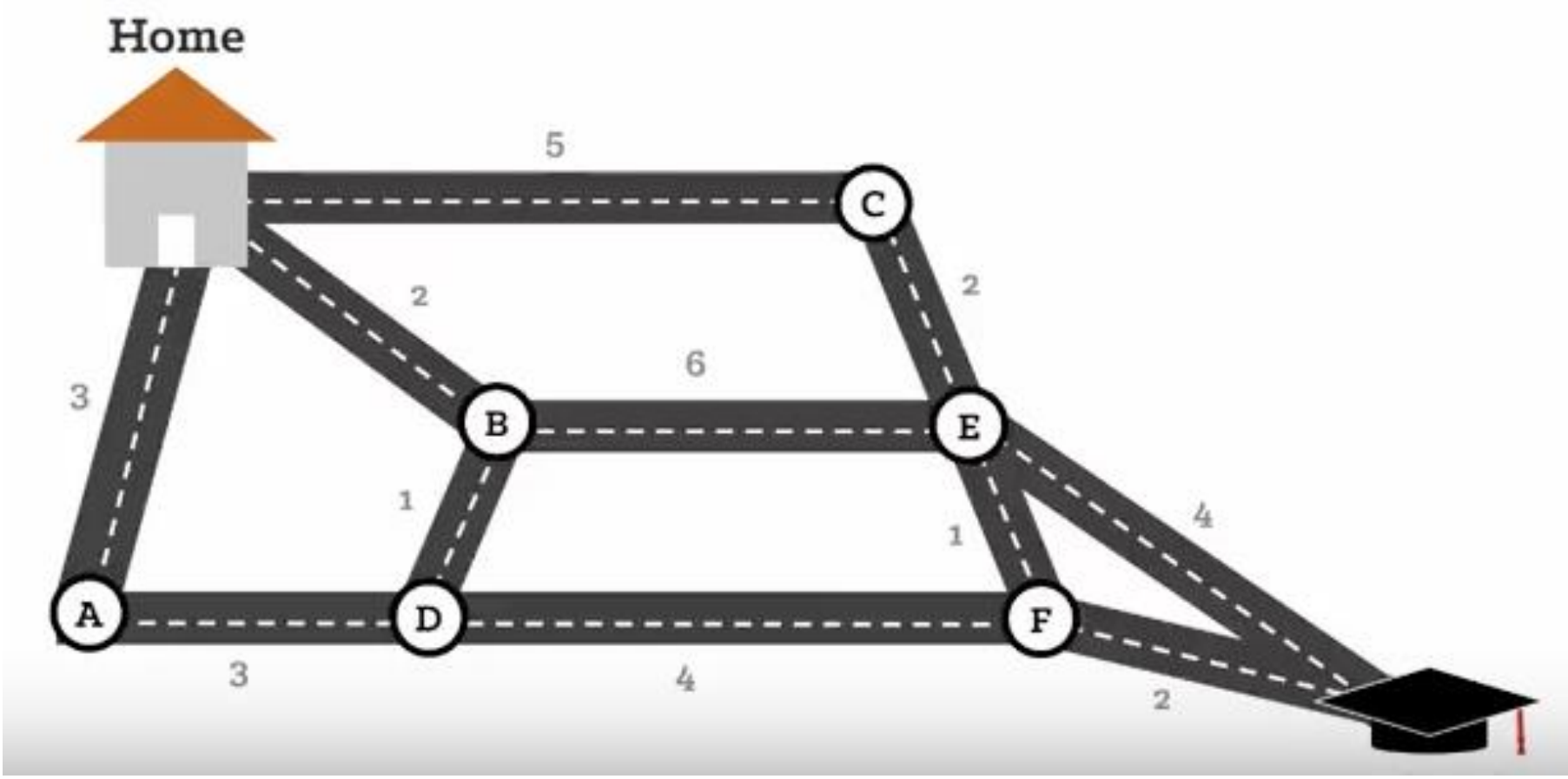
vertices, or nodes, denoted in the algorithm by v or u; weighted edges that connect two nodes: (u,v) denotes an edge, and $w(u,v)$ denotes its weight.

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra's algorithm. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

Dijkstra's algorithm, published in 1959 and named after its creator Dutch computer scientist Edsger Dijkstra, can be applied on a weighted graph. The graph can either be directed or undirected. One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge.

Suppose a student wants to go from home to school in the shortest possible way. She knows some roads are heavily congested and difficult to use. In Dijkstra's algorithm, this means the edge has a large weight--the shortest path tree found by the algorithm will try to avoid edges with larger weights. If the student looks up directions using a map service, it is likely they may use Dijkstra's algorithm, as well as others.

Find the shortest path from home to school in the following graph:



The shortest path, which could be found using Dijkstra's algorithm, is

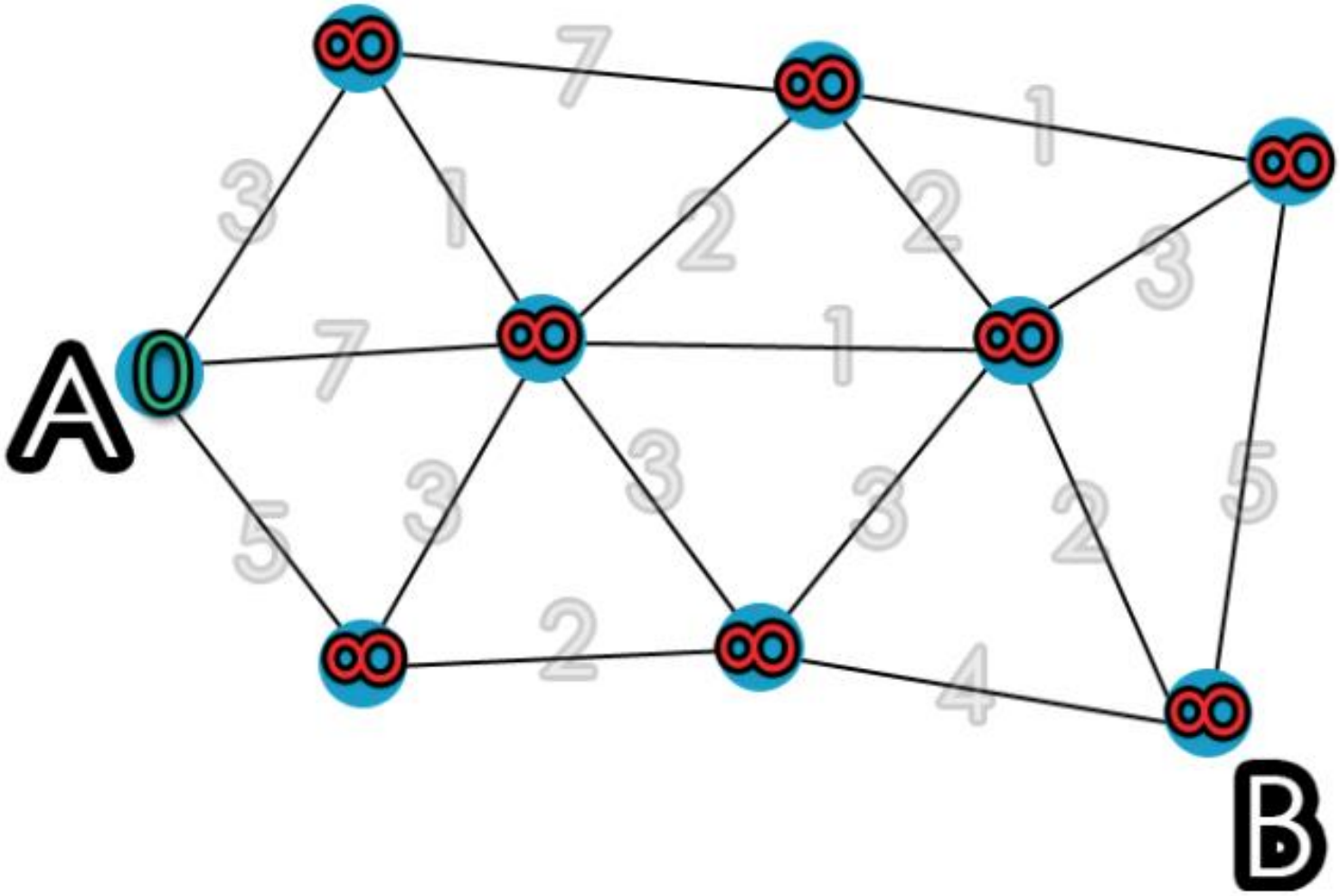
Home→*B*→*D*→*F*→School.

Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

The graph has the following:

vertices, or nodes, denoted in the algorithm by v or u;

weighted edges that connect two nodes: (u,v) denotes an edge, and $w(u,v)$ denotes its weight. In the diagram on the right, the weight for each edge is written in gray.


A
0
B

This is done by initializing three values:

1. *dist*, an array of distances from the source node *s* to each node in the graph, initialized the following way: *dist*(*s*) = 0; and for all other nodes *v*, *dist*(*v*) = ∞. This is done at the beginning because as the algorithm proceeds, the *dist* from the source to each node *v* in the graph will be recalculated and finalized when the shortest distance to *v* is found
2. *Q*, a queue of all nodes in the graph. At the end of the algorithm's progress, *Q* will be empty.
3. *S*, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S will contain all the nodes of the graph.

The algorithm proceeds as follows:

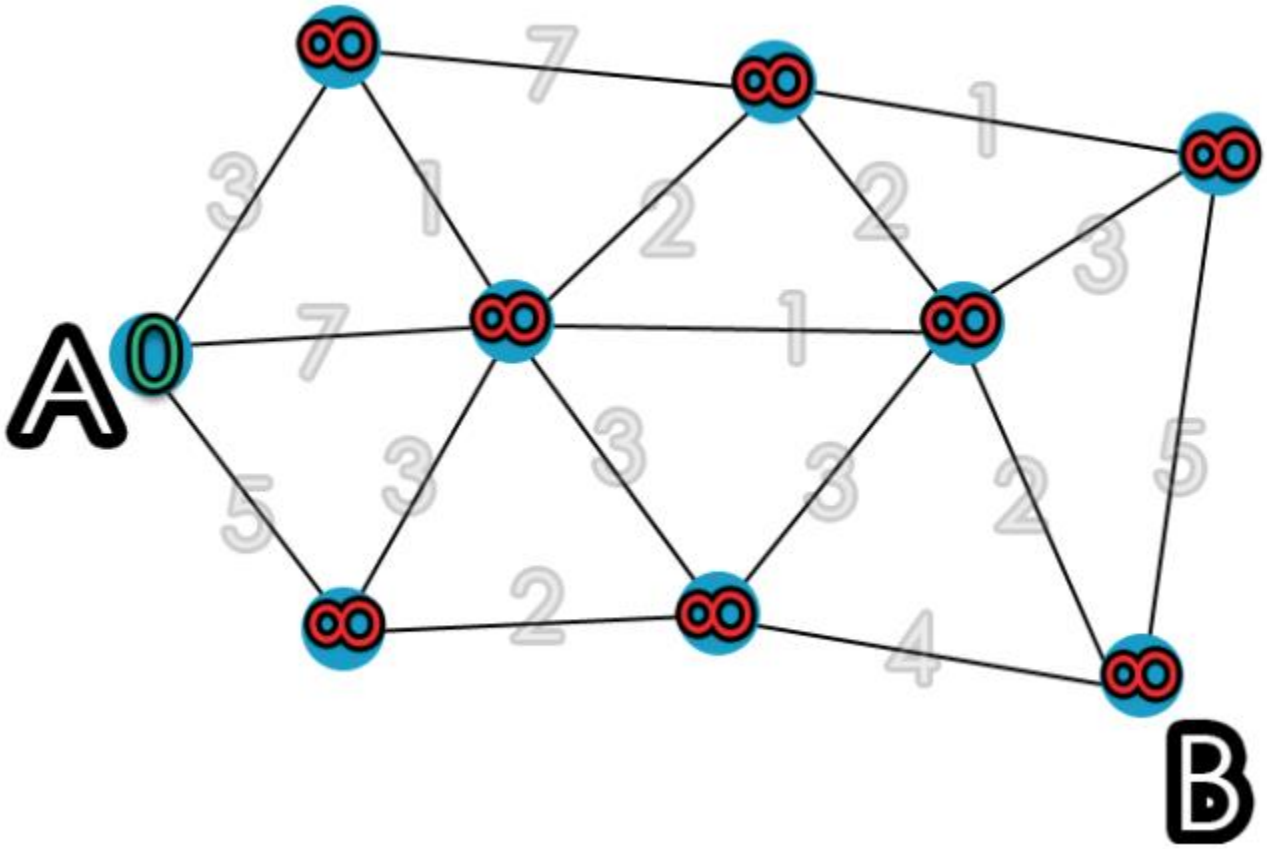
1. While *Q* is not empty, pop the node *v*, that is not already in *S*, from *Q* with the smallest *dist* (*v*). In the first run, source node *s* will be chosen because *dist*(*s*) was initialized to 0. In the next run, the next node with the smallest *dist* value is chosen.
2. Add node *v* to *S*, to indicate that *v* has been visited
3. Update *dist* values of adjacent nodes of the current node *v* as follows: for each new adjacent node *u*,

- if *dist* (*v*) + *weight*(*u*,*v*) < *dist* (*u*), there is a new minimal distance found for *u*, so update *dist* (*u*) to the new minimal distance value;
- otherwise, no updates are made to *dist* (*u*).

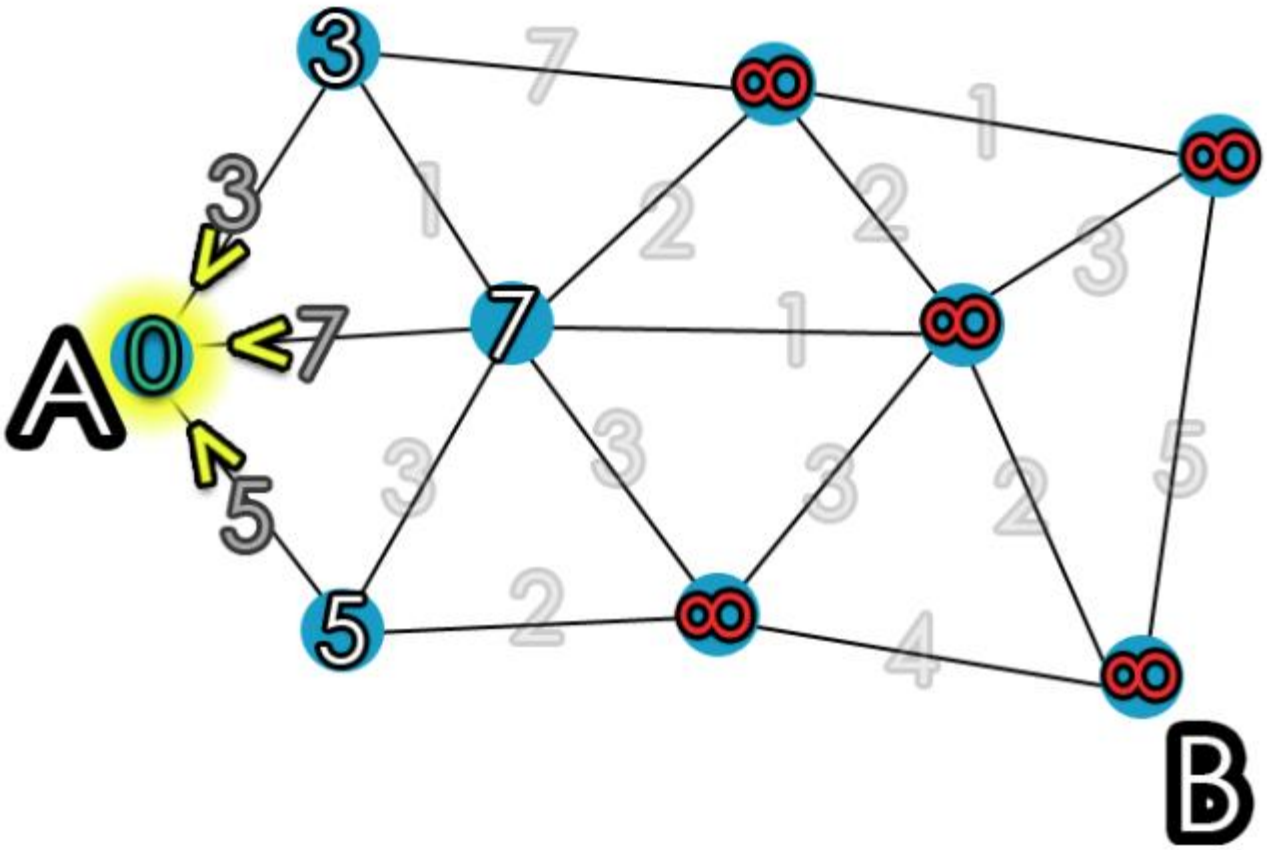
The algorithm has visited all nodes in the graph and found the smallest distance to each node. *dist* now contains the shortest path tree from source s*s*.

Note: The weight of an edge (*u*,*v*) is taken from the value associated with (*u*,*v*) on the graph.

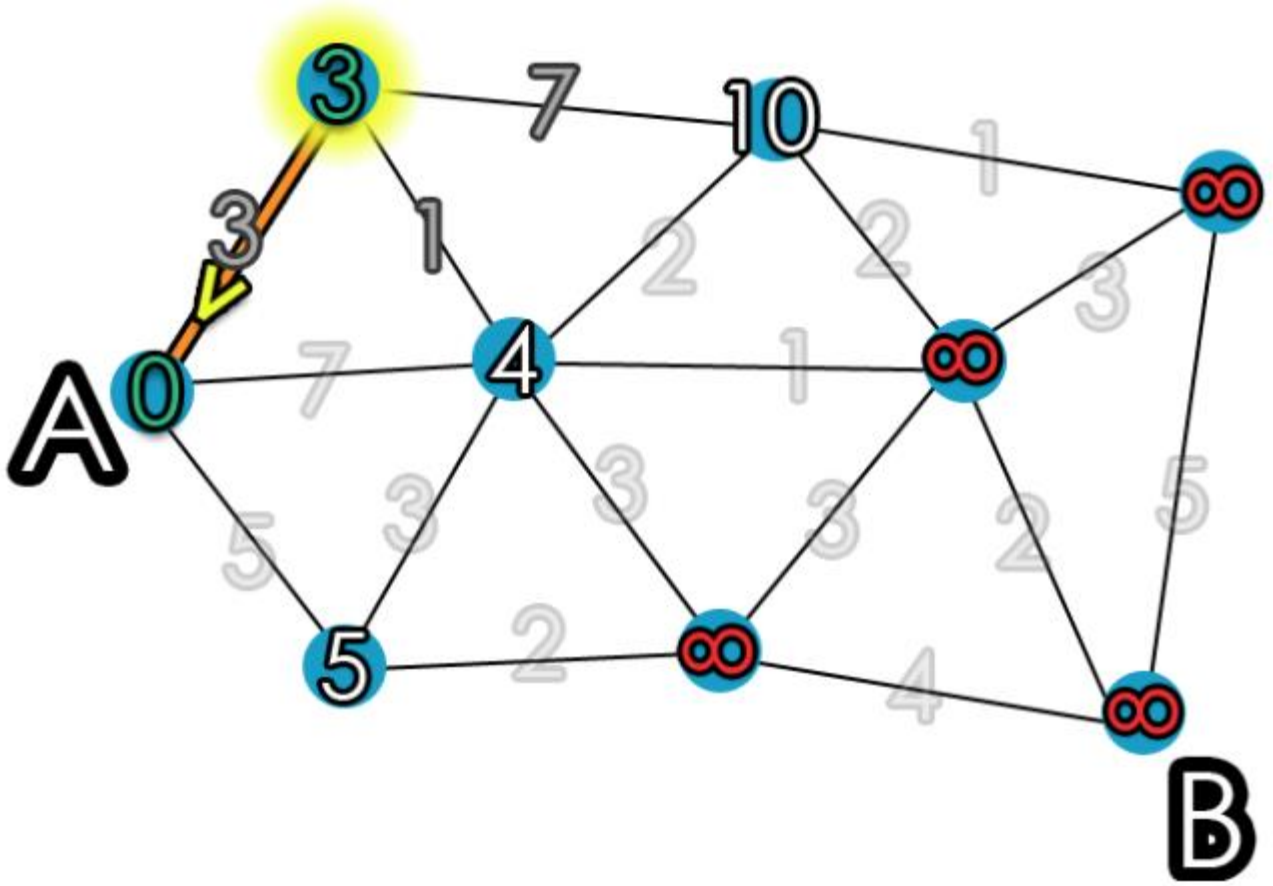
1. Initialize distances according to the algorithm.



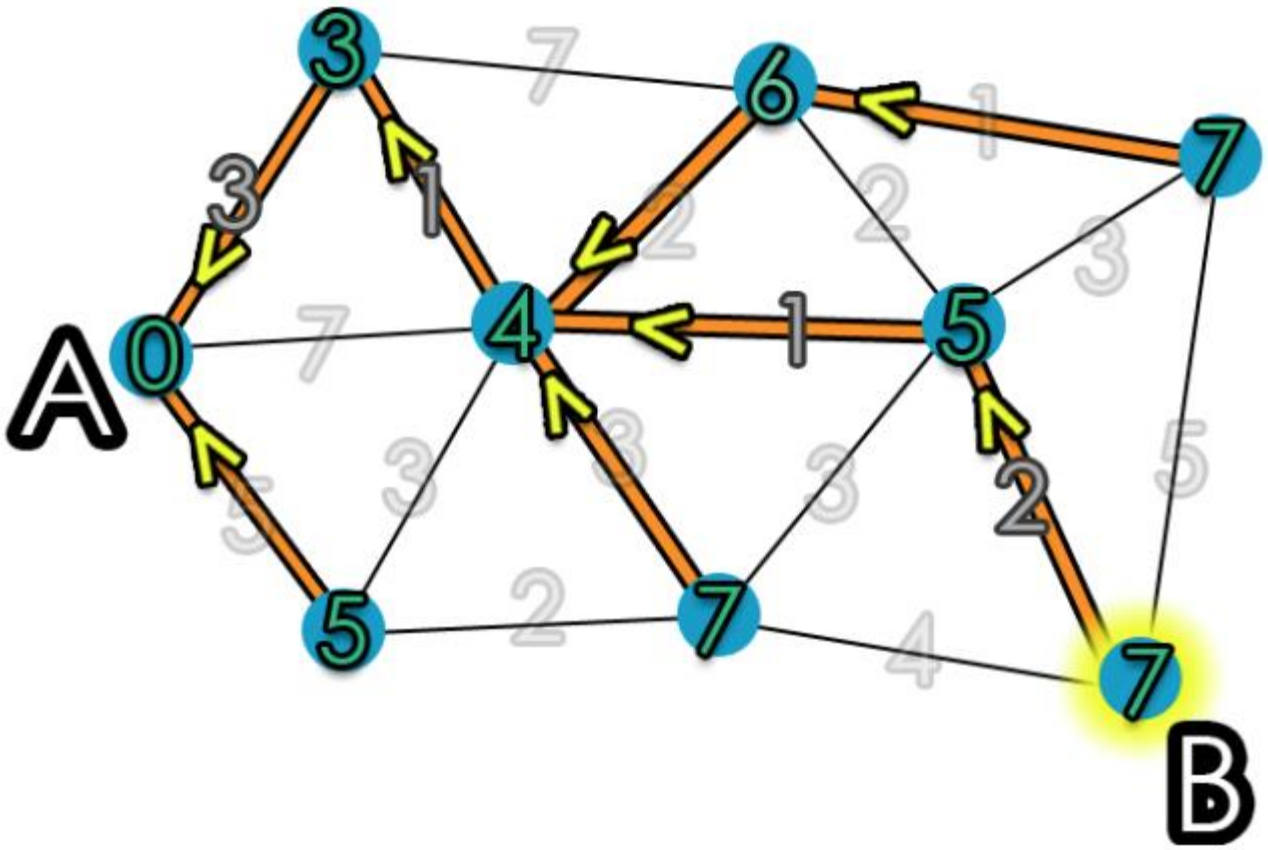
2. Pick first node and calculate distances to adjacent nodes.



3. Pick next node with minimal distance; repeat adjacent node distance calculations.



4. Final result of shortest-path tree



How Dijkstra's Algorithm works

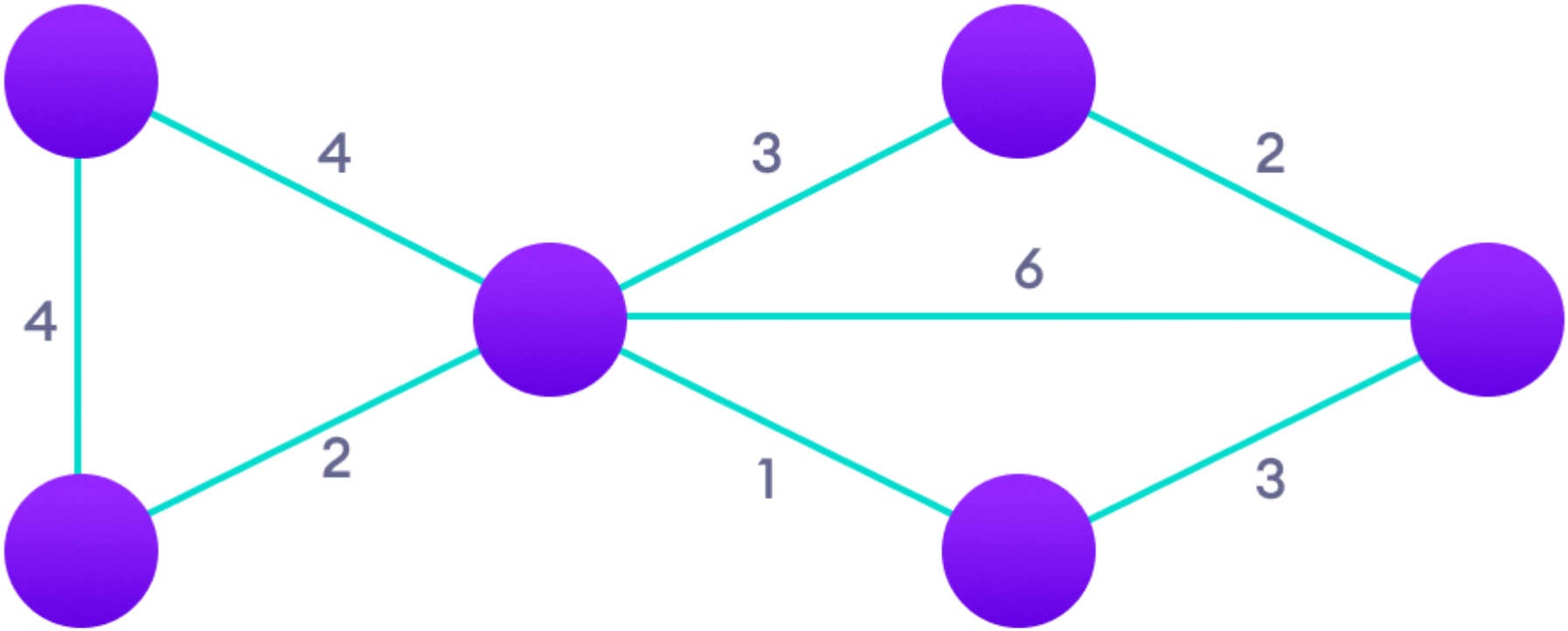
Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D

Each subpath is the shortest path

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

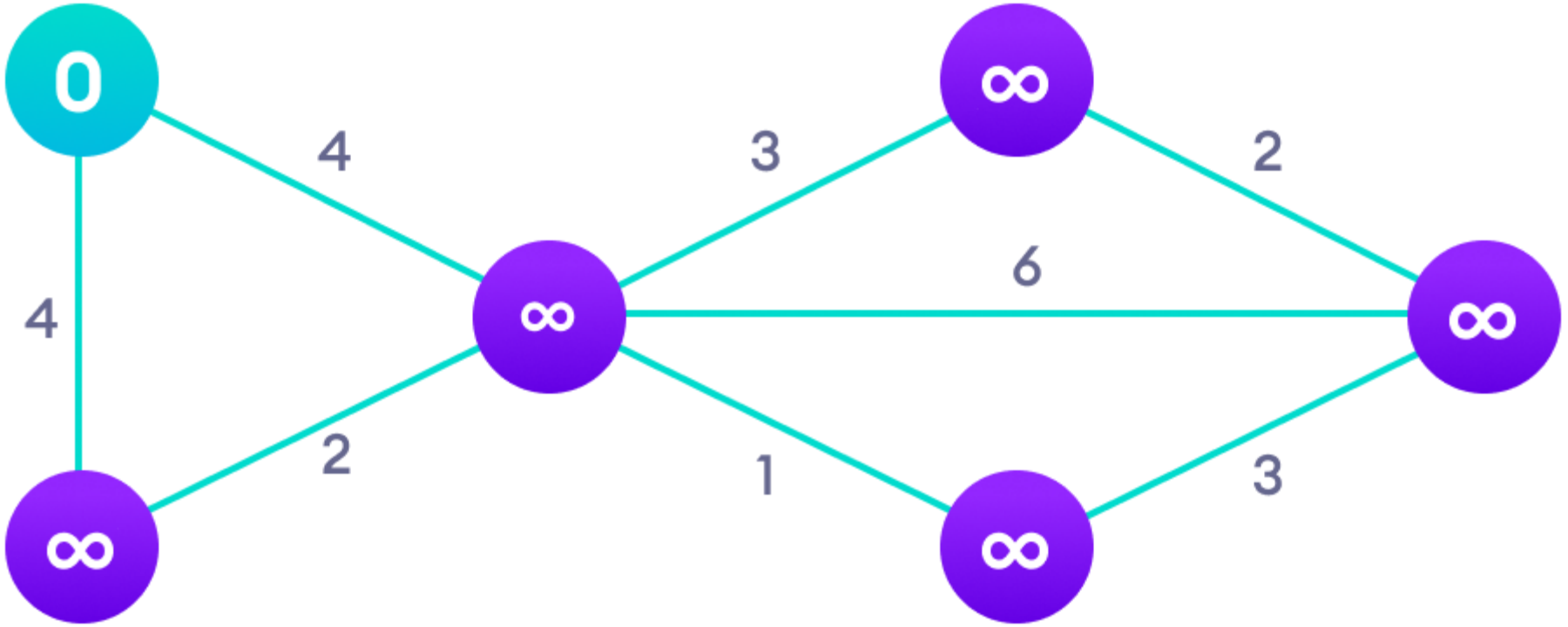
Example of Dijkstra's Algorithm



Step: 1

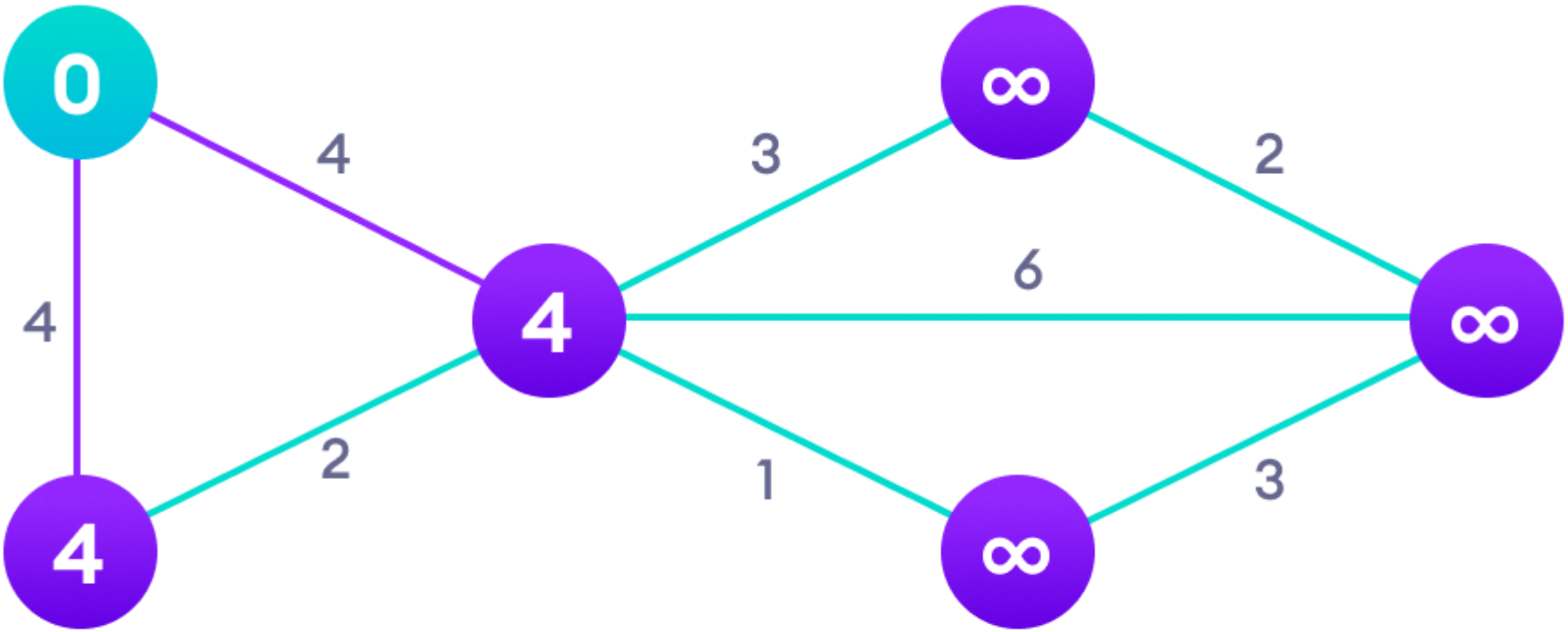
Start with a weighted graph

Choose a starting vertex and assign infinity path values to all other devices



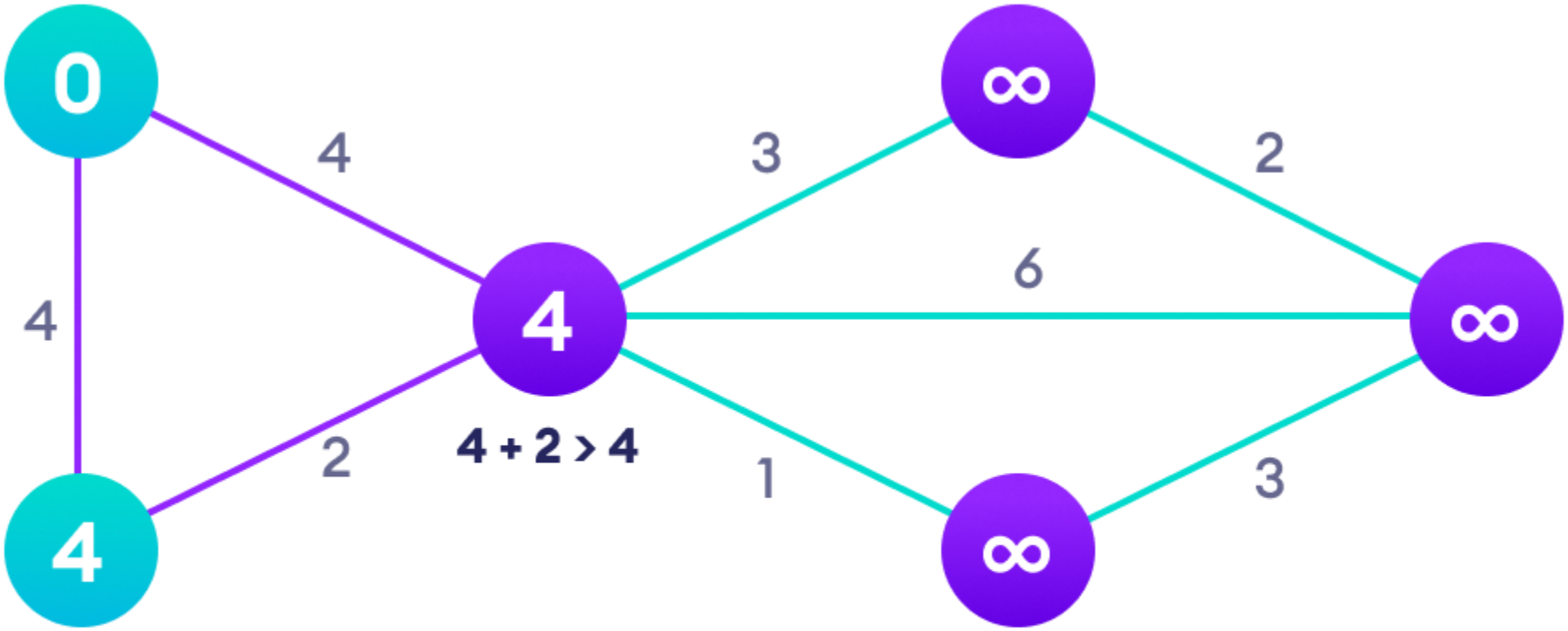
Step: 2

Go to each vertex and update its path length



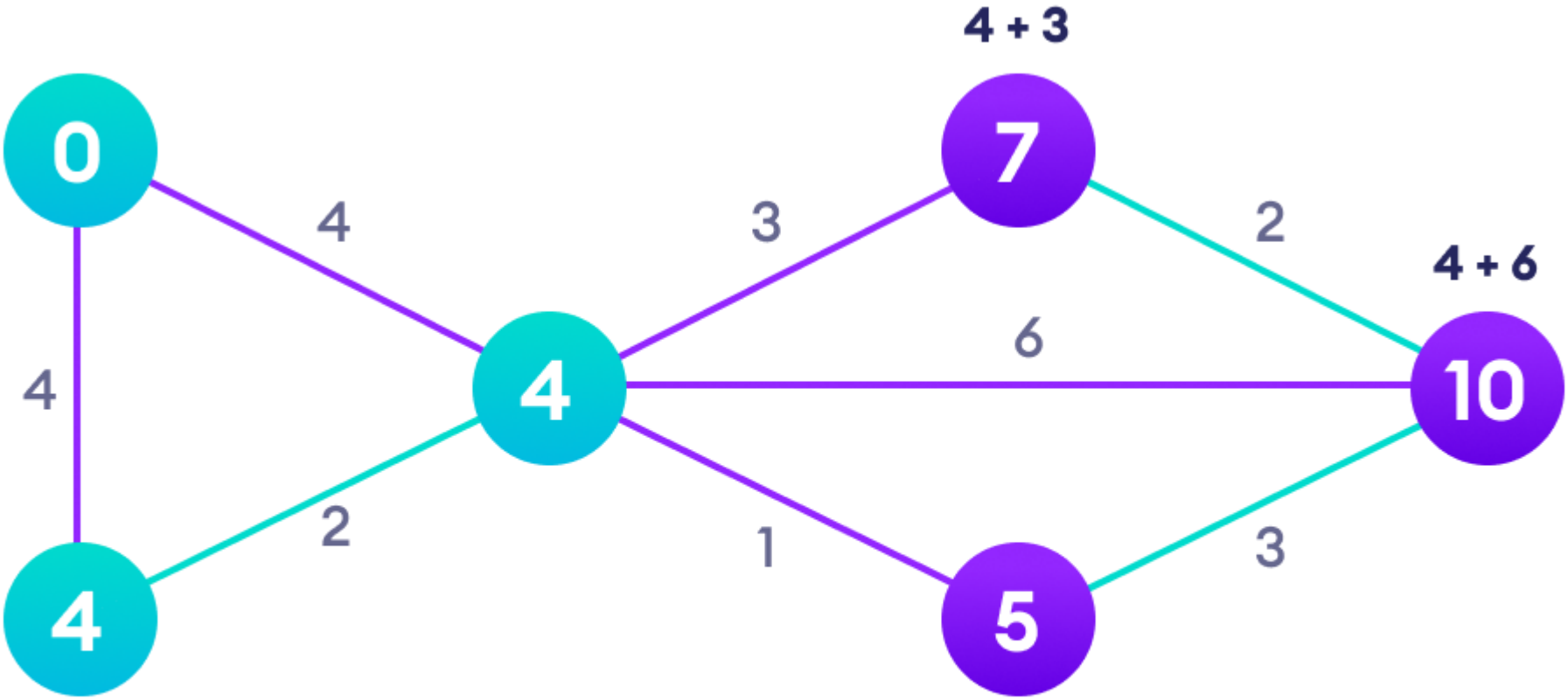
Step: 3

If the path length of the adjacent vertex is lesser than new path length, don't update it



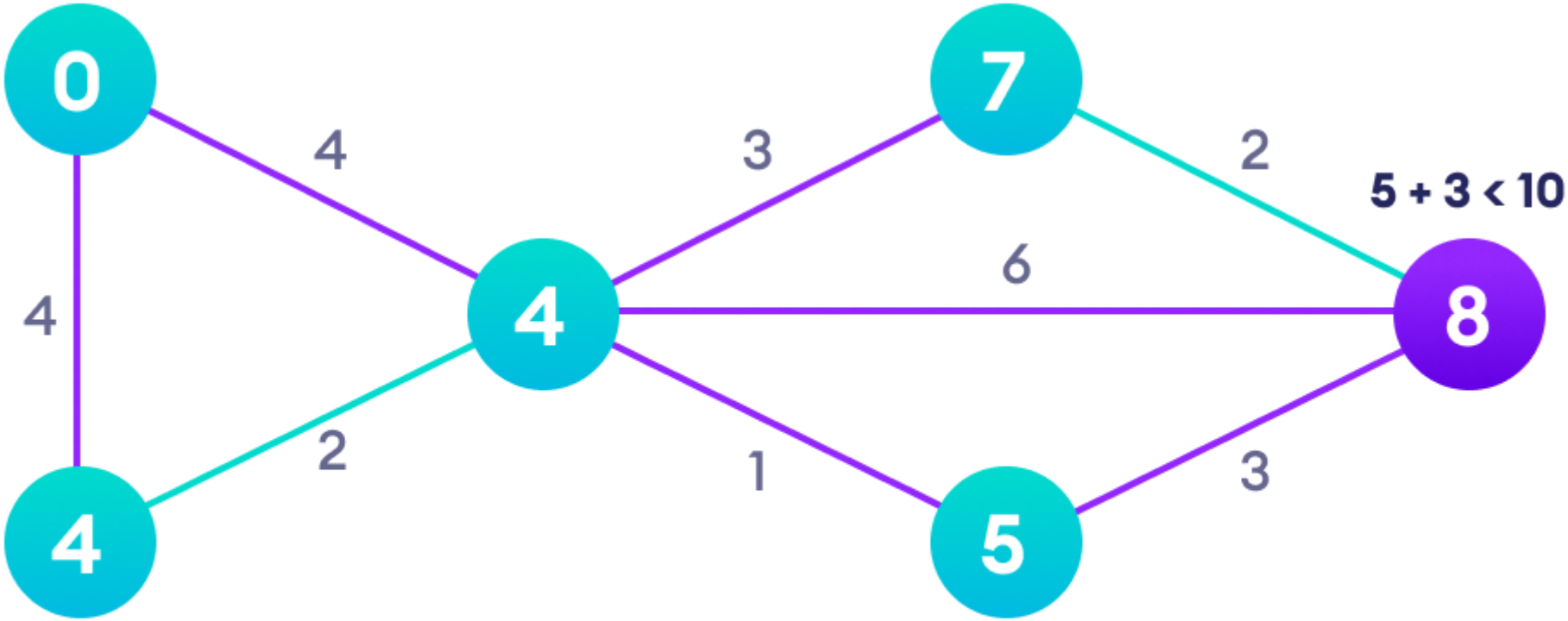
Step: 4

Avoid updating path lengths of already visited vertices



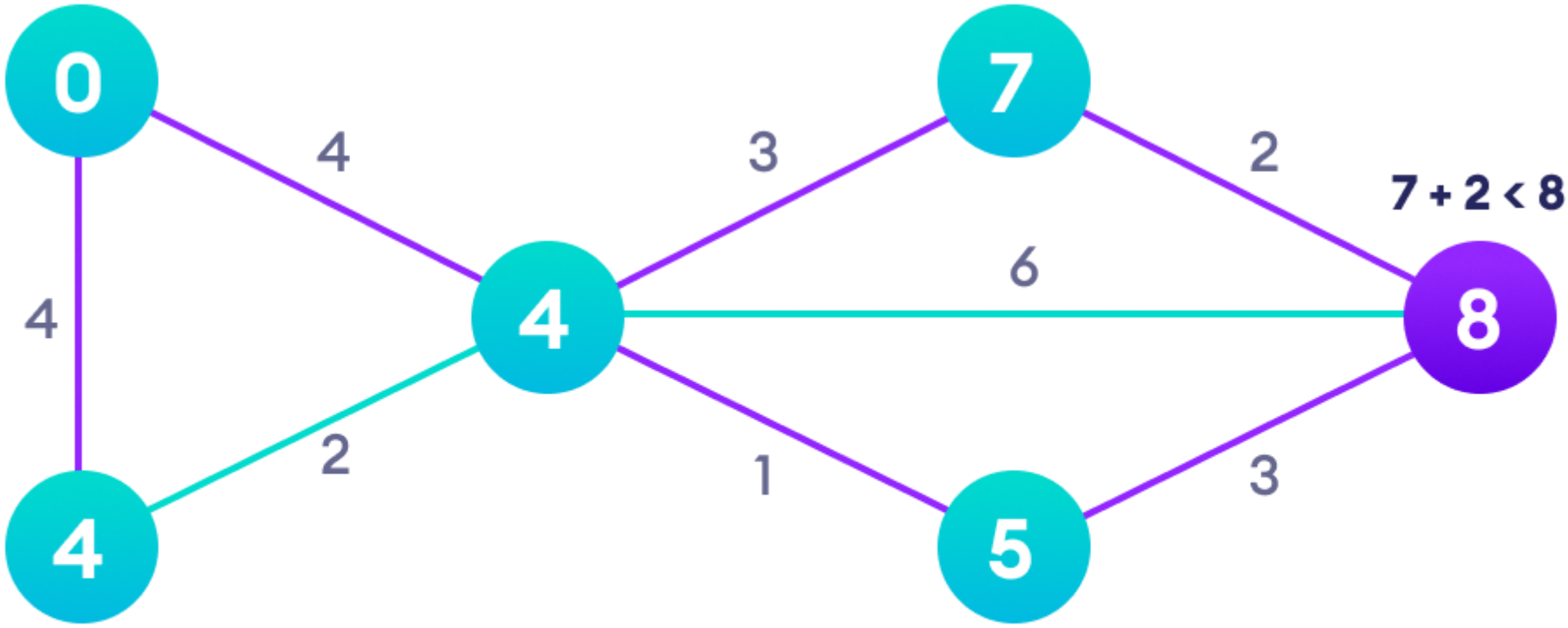
Step: 5

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7

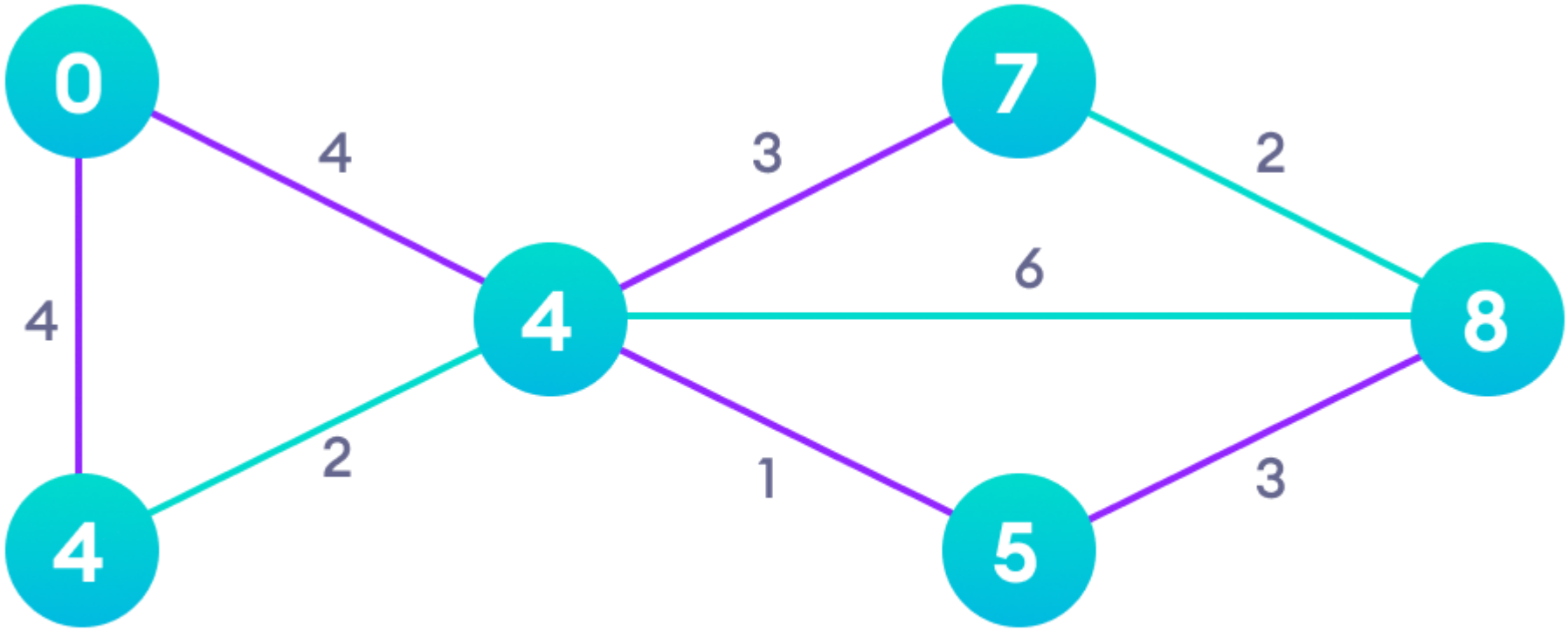


Step: 6

Notice how the rightmost vertex has its path length updated twice

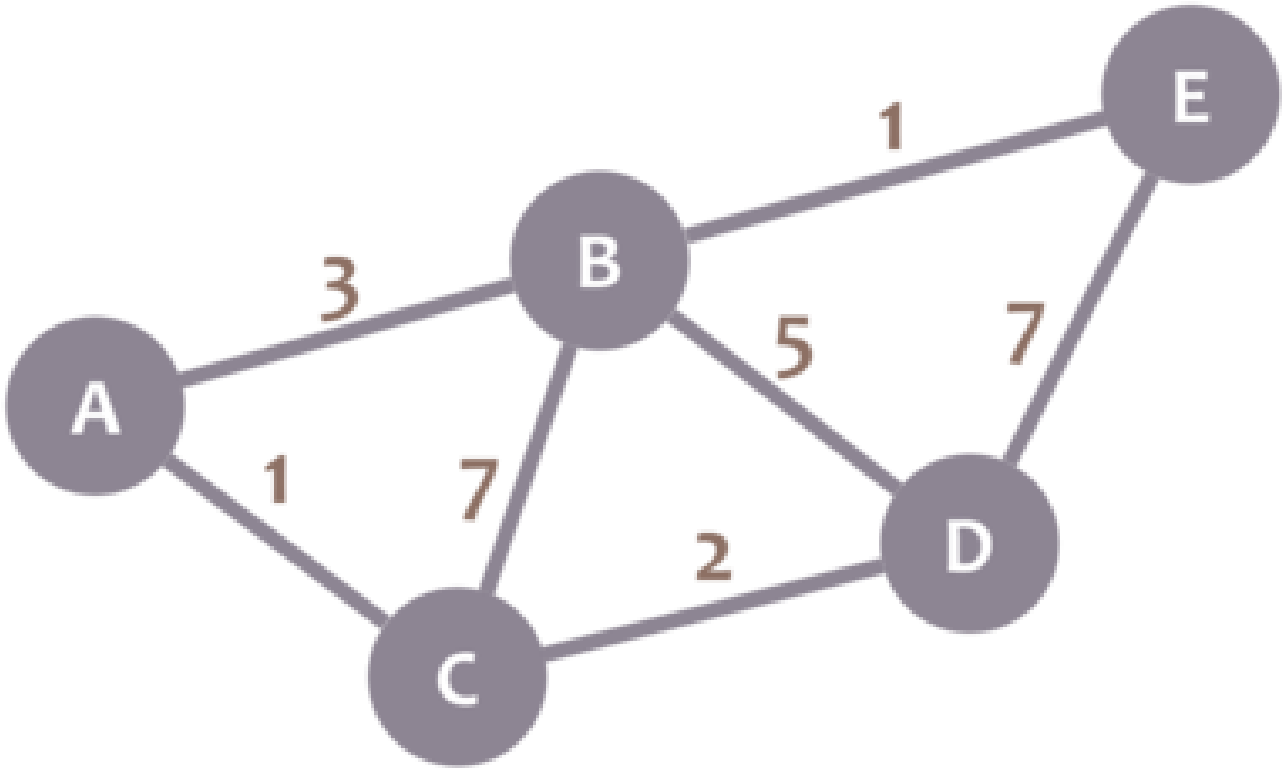


Repeat until all the vertices have been visited

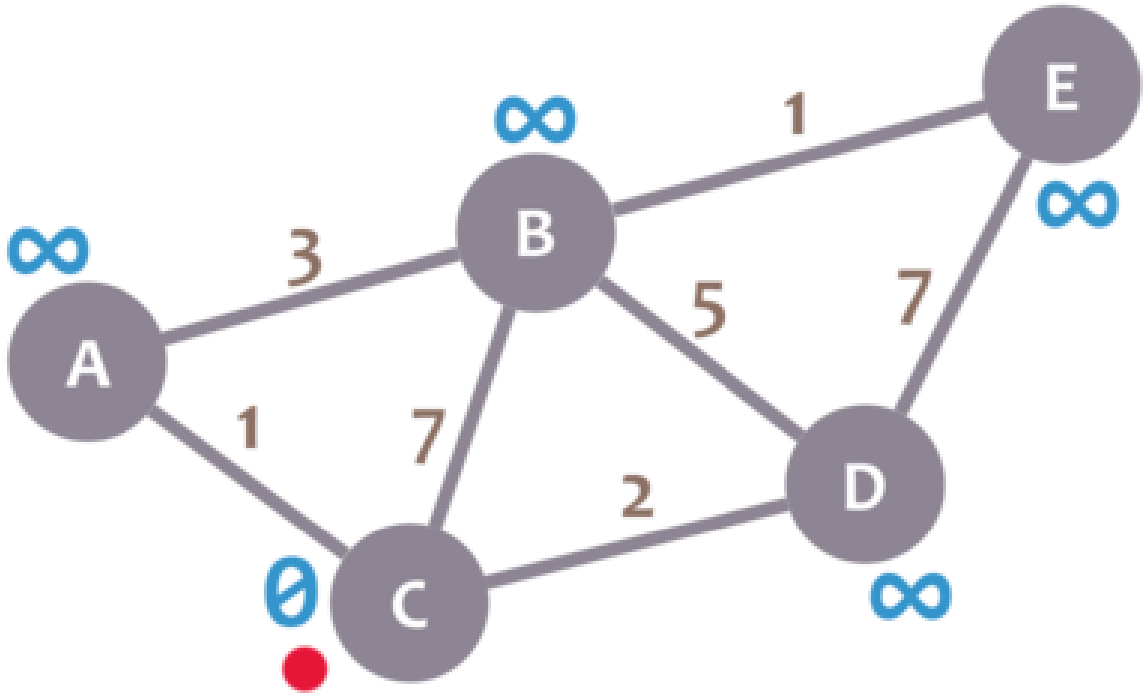


Step: 8

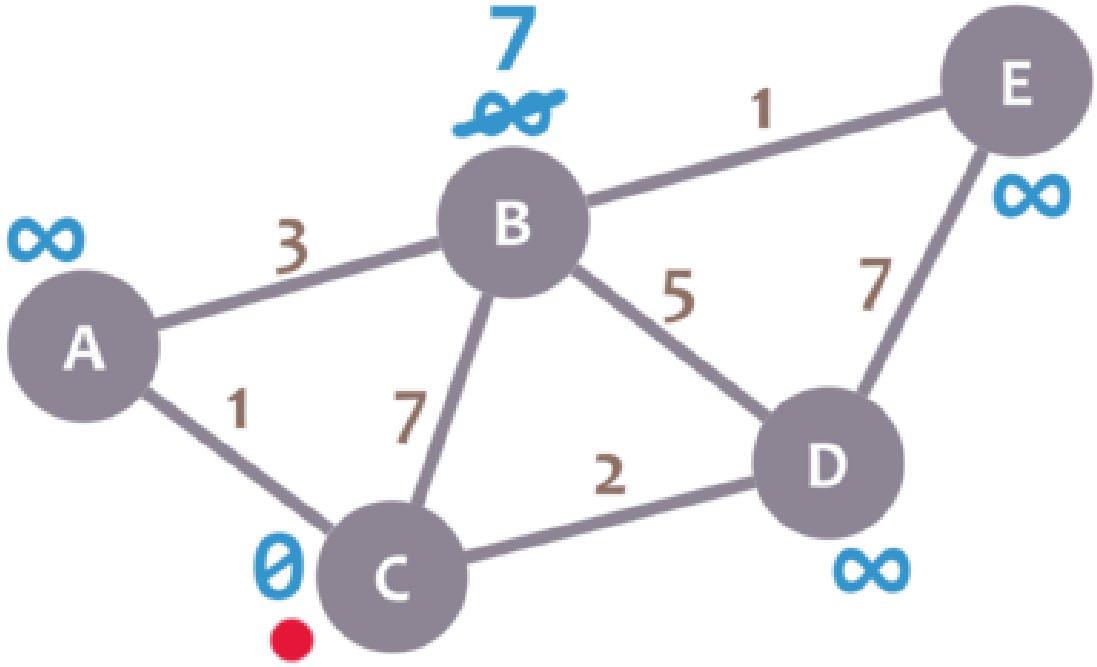
Let's calculate the shortest path between node C and the other nodes in our graph:



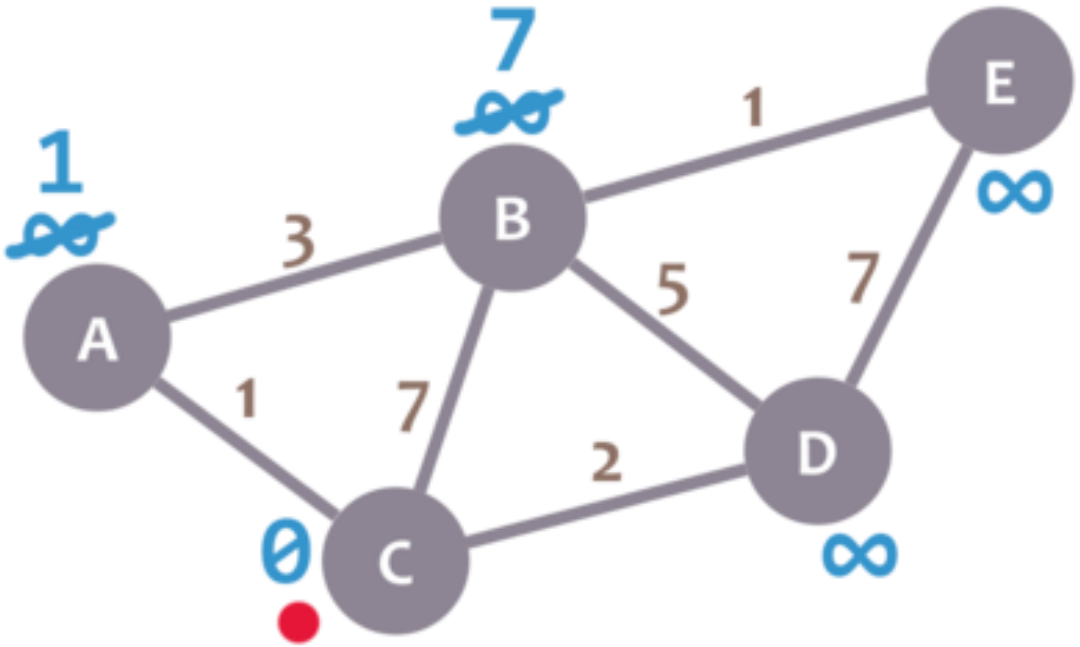
During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):



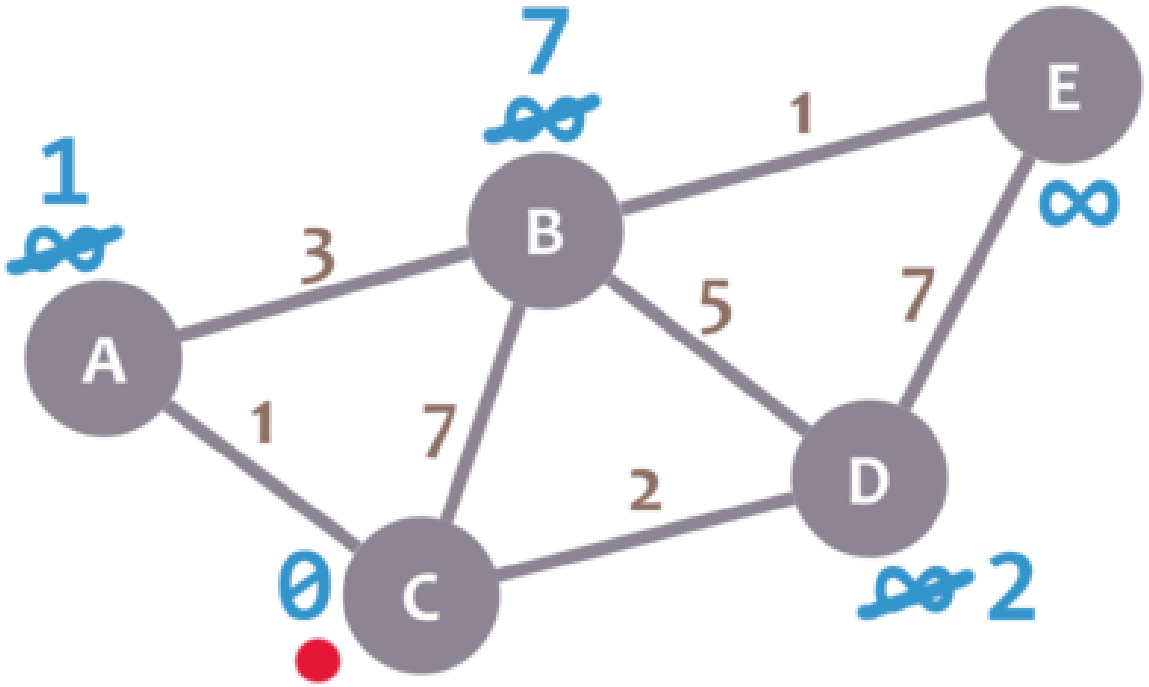
Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



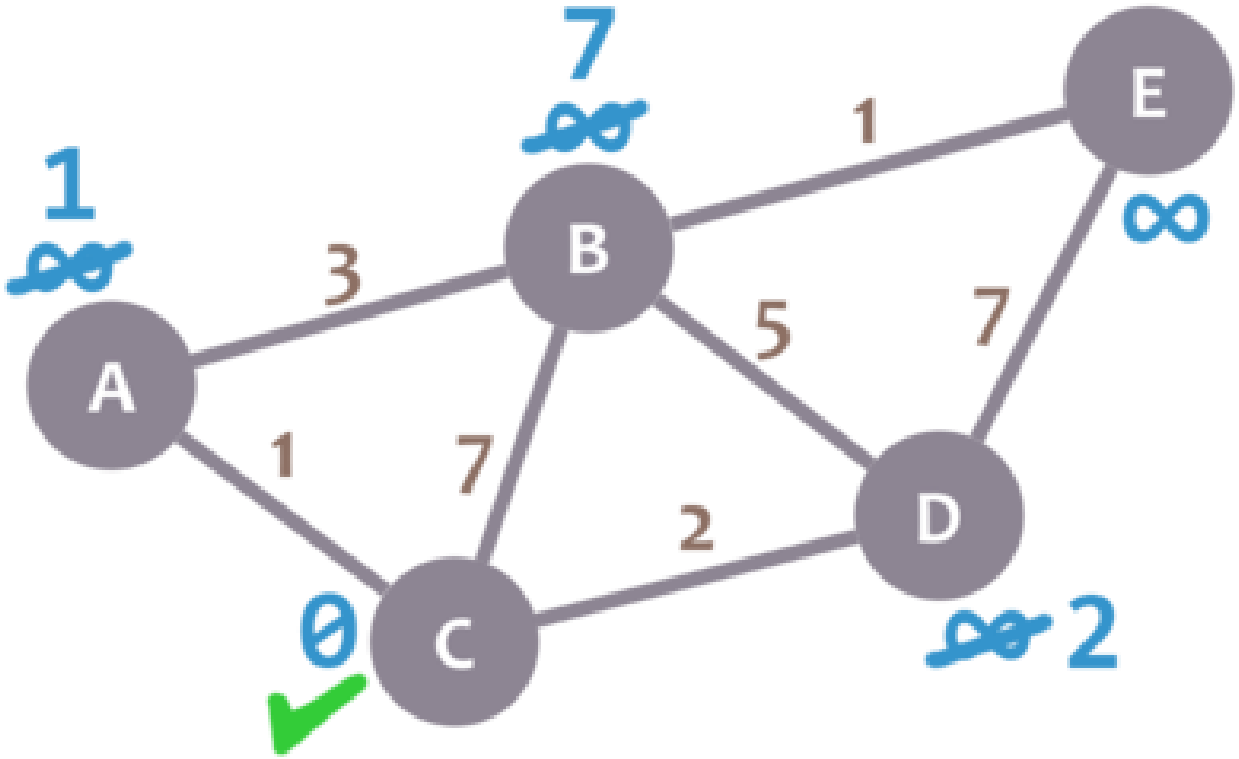
Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



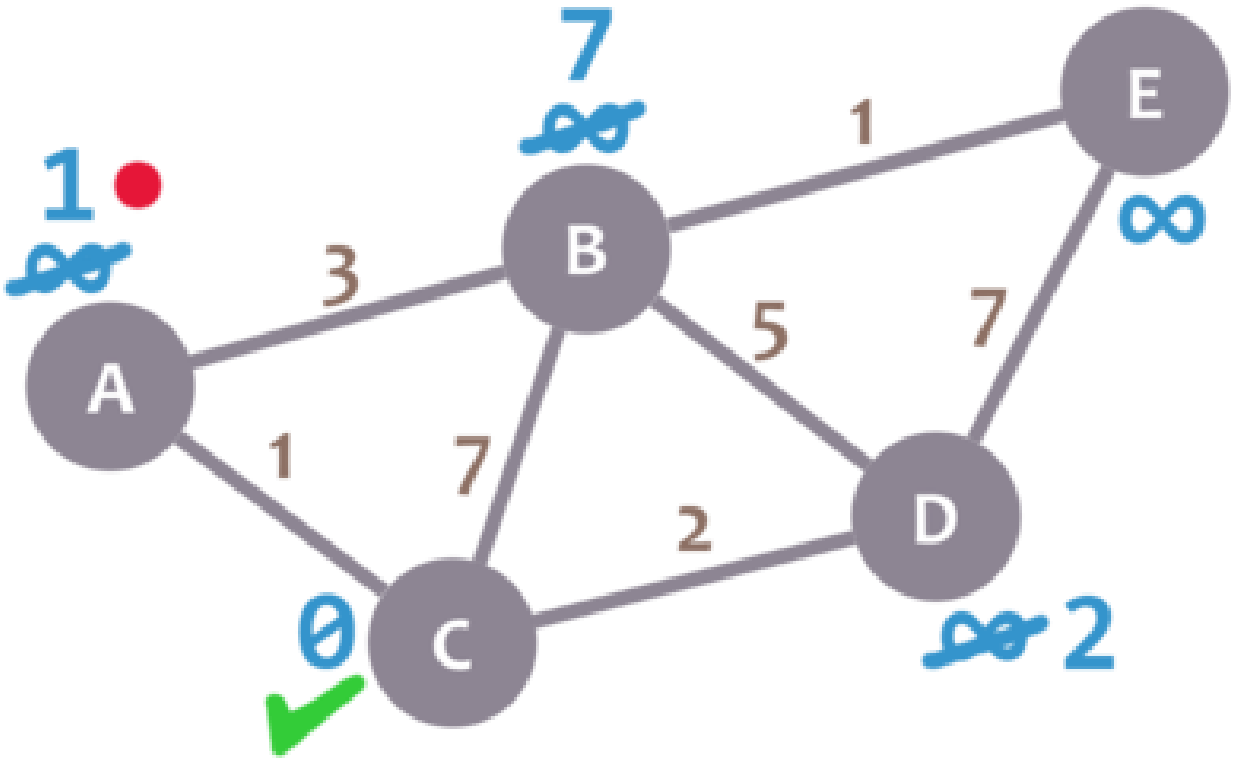
Repeat the same procedure for D:



We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:

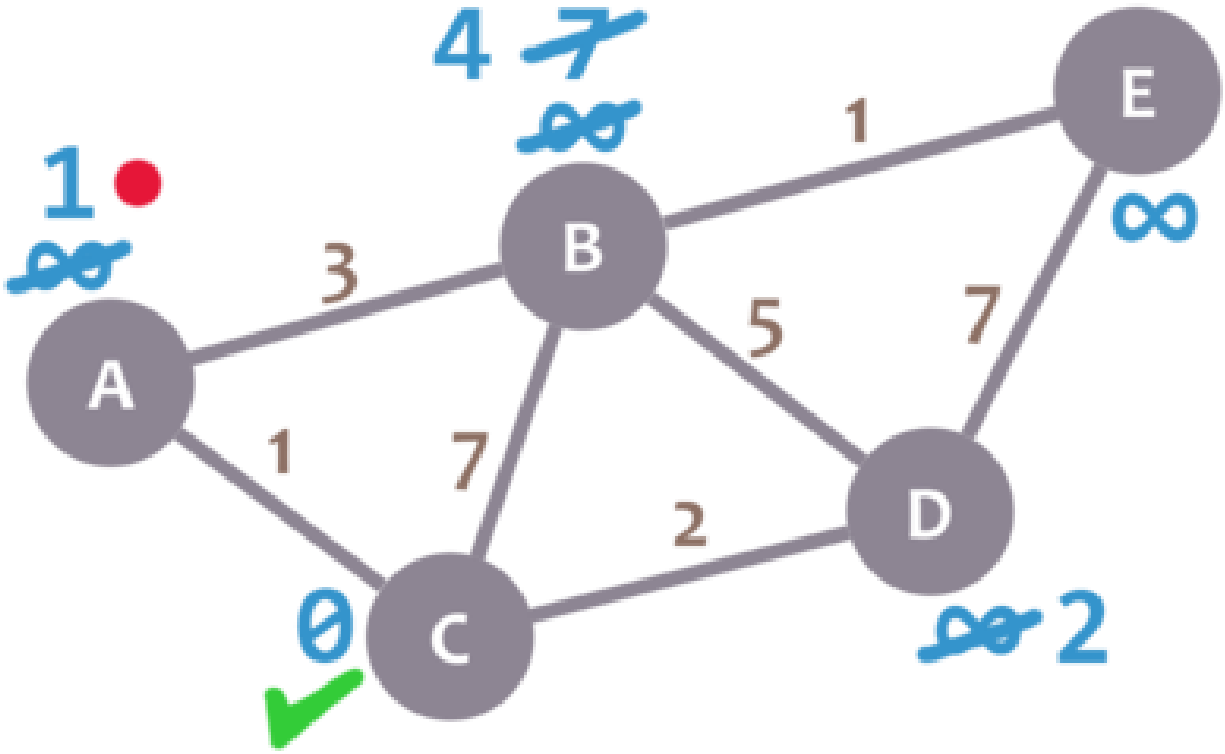


We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

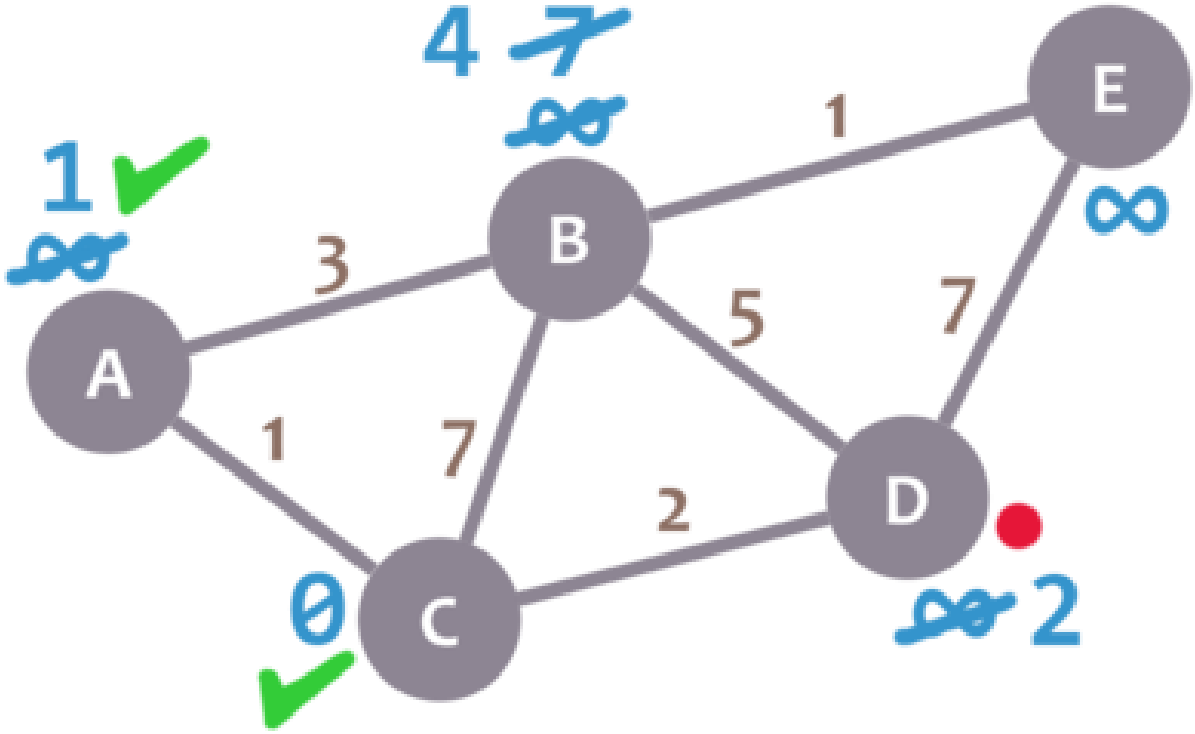


And now we repeat the algorithm. We check the neighbours of our current node, ignoring the visited nodes. This means we only check B.

For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



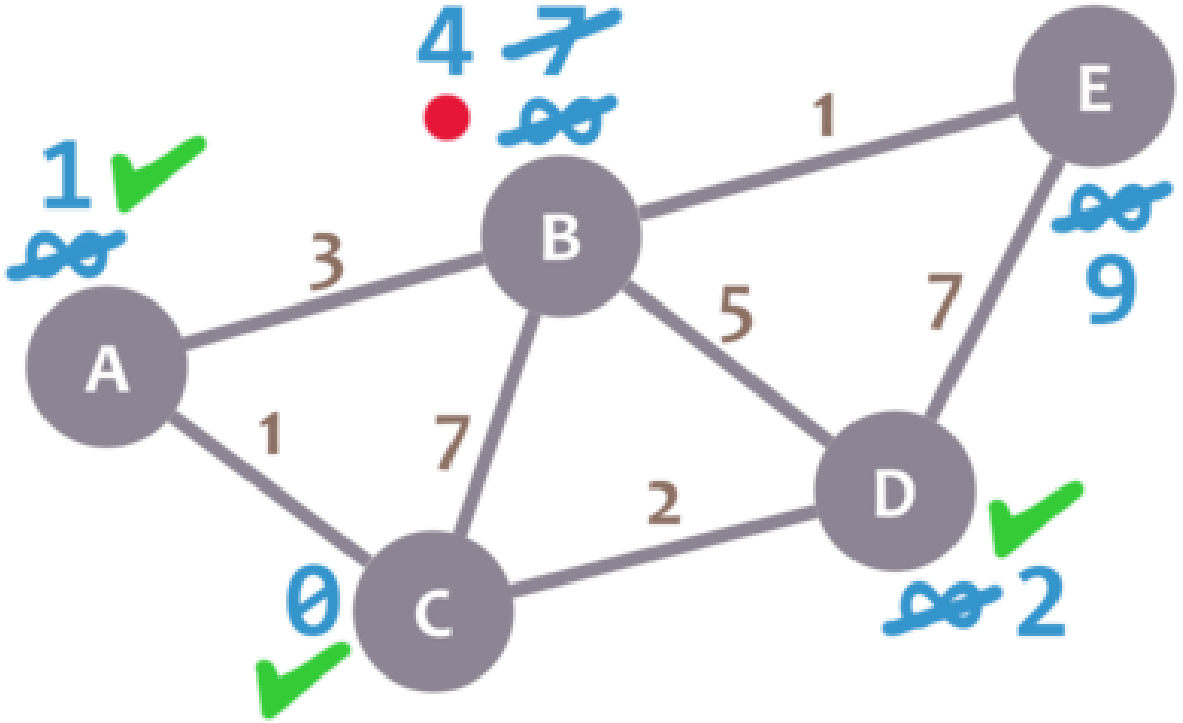
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



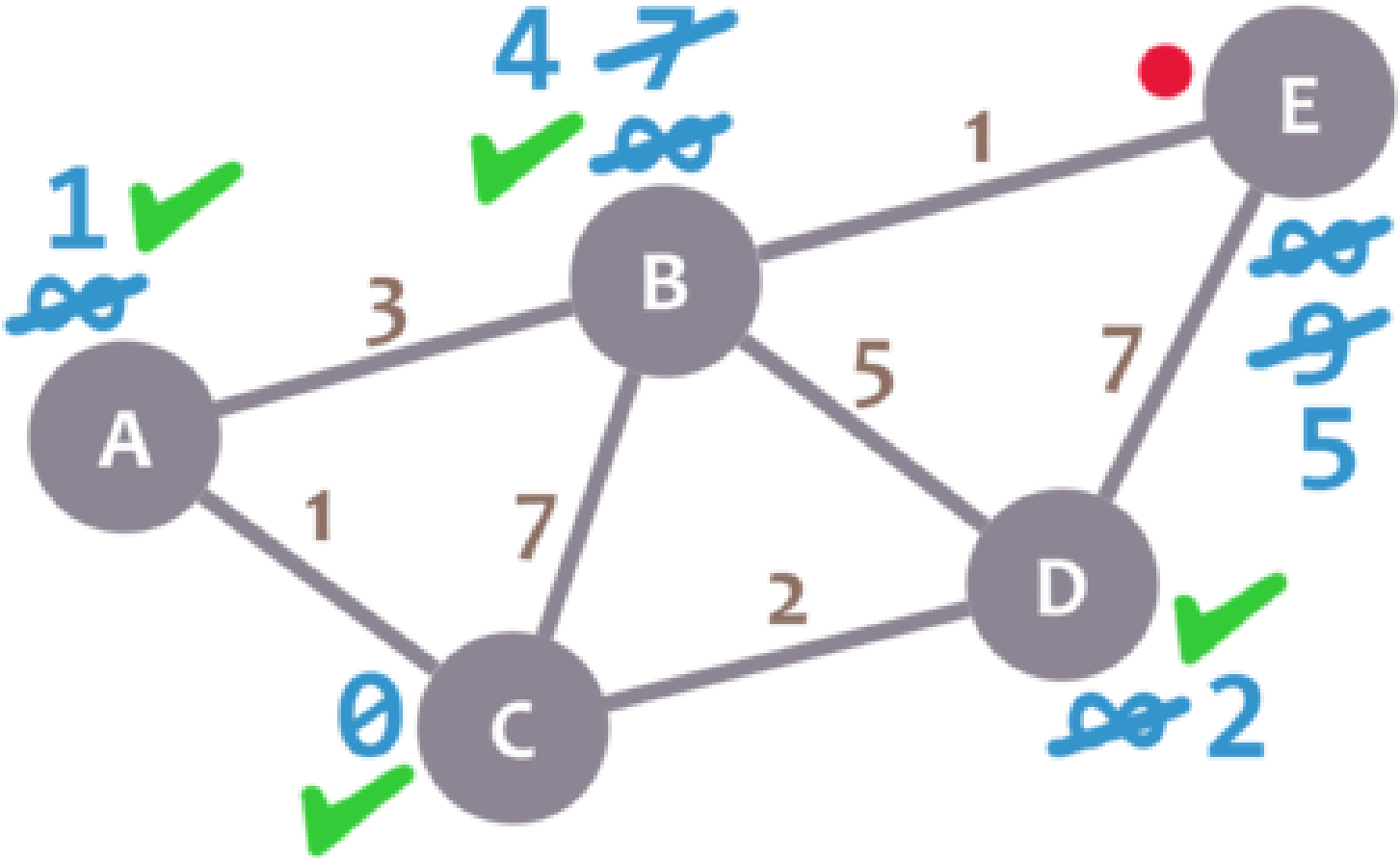
We repeat the algorithm again. This time, we check B and E.

For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

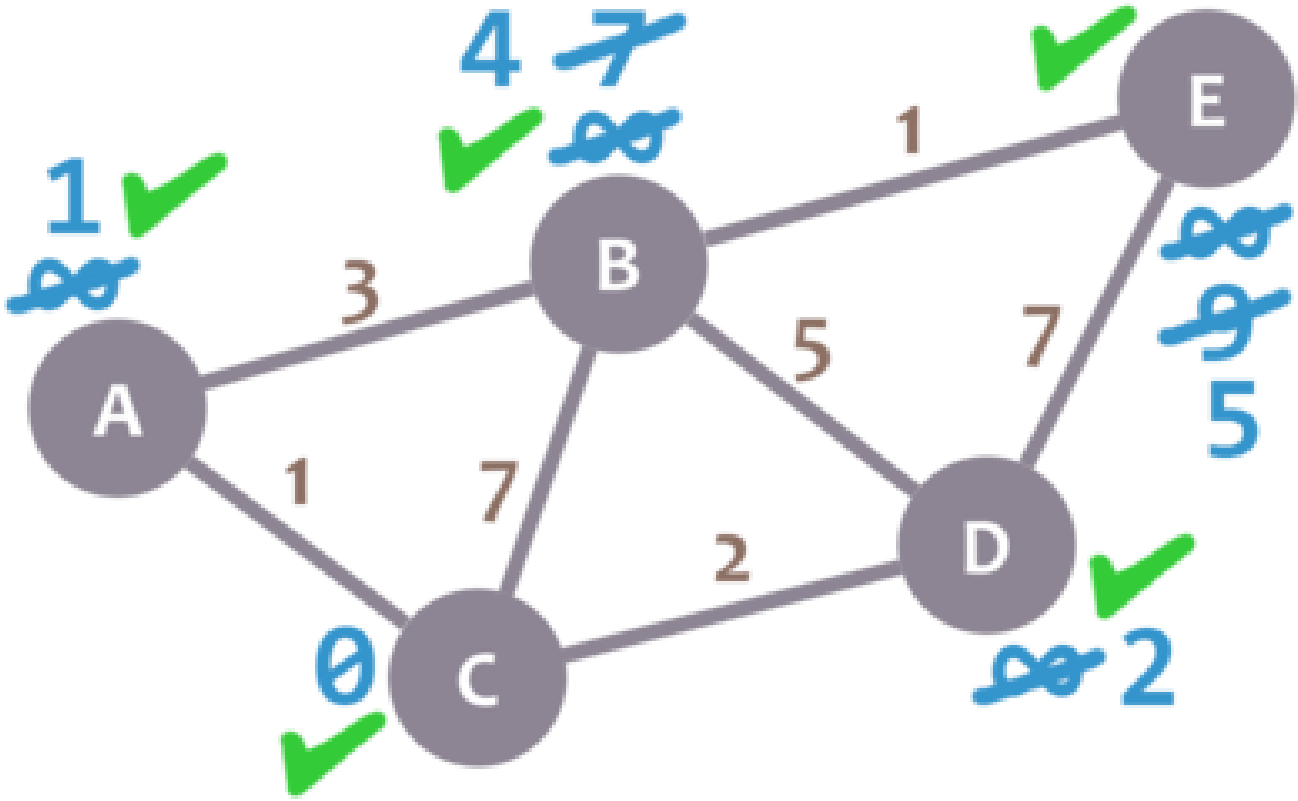
We mark D as visited and set our current node to B.



Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)

Description of the algorithm:

1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node C.
3. For each neighbour N of your current node C: add the current distance of C with the weight of the edge connecting C-N. If it's smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. If there are non-visited nodes, go to step 2.

Dijkstra's Algorithm Applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

All-Pairs Shortest Paths

It aims to figure out the shortest path from each vertex v to every other u. Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical regarding memory consumption, so these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in u's row and v's column should be the weight of the shortest path from u to v.

Three approaches for improvement:

Algorithm	Cost
Matrix Multiplication	O (V3 logV)
Floyd-Warshall	O (V3)
Johnson O	(V2 log?V+VE)

Unlike the single-source algorithms, which assume an adjacency list representation of the graph, most of the algorithm uses an adjacency matrix representation. The input is a n x n matrix W representing the edge weights of an n-vertex directed graph G = (V, E). That is, W = (w_{ij}), where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

- Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.
- Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.
- **Optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.

Let the vertices of G be V = {1, 2.....n} and consider a subset {1, 2.....k} of vertices for some k. For any pair of vertices i, j ∈ V, considered all paths from i to j whose intermediate vertices are all drawn from {1, 2.....k}, and let p be a minimum weight path from amongst them. The Floyd-Warshall algorithm exploits a link between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2.....k-1}. The link depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2.....k-1}. Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k-1} is also the shortest path i to j with all intermediate vertices in the set {1, 2.....k}.

If k is an intermediate vertex of path p, then we break p down into

$i \rightarrow k \rightarrow j$.

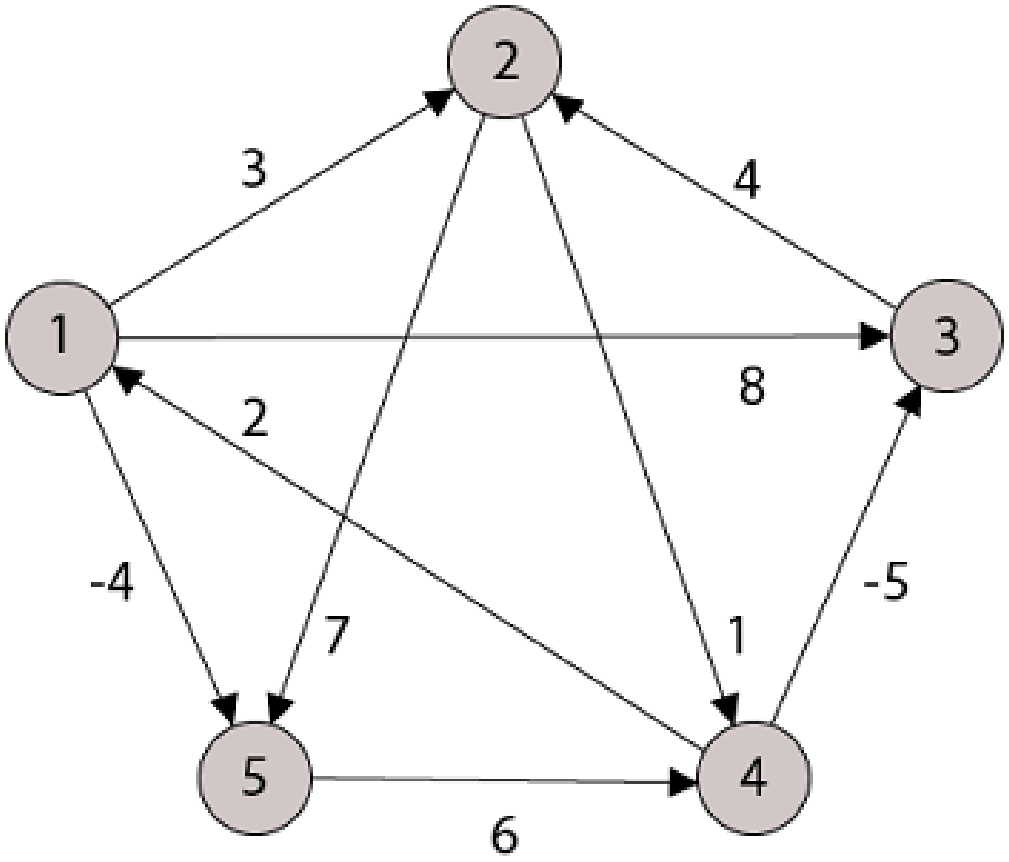
Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k}.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

The strategy adopted by the Floyd-Warshall algorithm is **Dynamic Programming**. The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O (1) time. The algorithm thus runs in time $\theta(n^3)$.

Apply Floyd-Warshall algorithm for constructing the shortest path. Show that matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.



Solution:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Step (i) When k = 0

$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(0)} = \begin{matrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 3 & \text{NIL} \\ 4 & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

Step (ii) When k =1

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(1)} = \min (d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)})$$

$$d_{14}^{(1)} = \min (\infty, 0 + \infty) = \infty$$

$$d_{15}^{(1)} = \min (d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$$

$$d_{15}^{(1)} = \min (-4, 0 + -4) = -4$$

$$d_{21}^{(1)} = \min (d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$$

$$d_{21}^{(1)} = \min (\infty, \infty + 0) = \infty$$

$$d_{23}^{(1)} = \min (d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$$

$$d_{23}^{(1)} = \min ((\infty, \infty + 8) = \infty$$

$$d_{31}^{(1)} = \min (d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$$

$$d_{31}^{(1)} = \min (\infty, \infty + 0) = \infty$$

$$d_{35}^{(1)} = \min (d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$$

$$d_{35}^{(1)} = \min (\infty, \infty + (-4)) = \infty$$

$$d_{42}^{(1)} = \min (d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)})$$

$$d_{42}^{(1)} = \min (\infty, 2 + 3) = 5$$

$$d_{43}^{(1)} = \min (d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$$

$$d_{43}^{(1)} = \min (\infty, 2 + 8) = 10$$

$$d_{45}^{(1)} = \min (d_{45}^{(0)}, d_{41}^{(0)} + d_{15}^{(0)})$$

$$d_{45}^{(1)} = \min (\infty, 2 + (-4)) = -2$$

$$d_{51}^{(1)} = \min (d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$$

$$d_{51}^{(1)} = \min (\infty, \infty + 0) = \infty$$

$$D_{ij}^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$\pi^{(1)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iii) When k = 2

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{14}^{(2)} = \min\left(d_{14}^{(1)}, d_{12}^{(1)} + d_{24}^{(1)}\right)$$

$$d_{14}^{(2)} = \min(\infty, 3 + 1) = 4$$

$$d_{21}^{(2)} = \min\left(d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)}\right)$$

$$d_{21}^{(2)} = \min(\infty, 0 + \infty) = \infty$$

$$d_{34}^{(2)} = \min\left(d_{34}^{(1)}, d_{32}^{(1)} + d_{24}^{(1)}\right)$$

$$d_{34}^{(2)} = \min(-5, 4 + 1) = -5$$

$$d_{35}^{(2)} = \min\left(d_{35}^{(1)}, d_{32}^{(1)} + d_{25}^{(1)}\right)$$

$$d_{35}^{(2)} = \min(\infty, 4 + 7) = 11$$

$$d_{43}^{(2)} = \min\left(d_{43}^{(1)}, d_{42}^{(1)} + d_{23}^{(1)}\right)$$

$$d_{43}^{(2)} = \min(10, 5 + \infty) = 10$$

$$D_{ij}^{(2)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(2)} = \begin{matrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & 2 \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iv) When k = 3

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{14}^{(3)} = \min\left(d_{14}^{(2)}, d_{13}^{(2)} + d_{34}^{(2)}\right)$$

$$d_{14}^{(3)} = \min\left(4, 8 + (-5)\right) = 3$$

$$D_{ij}^{(3)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(3)} = \begin{matrix} NIL & 1 & 1 & 3 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & 2 \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (v) When k = 4

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{21}^{(4)} = \min\left(d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)}\right)$$

$$d_{21}^{(4)} = \min(\infty, 1 + 2) = 3$$

$$d_{23}^{(4)} = \min\left(d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)}\right)$$

$$d_{23}^{(4)} = \min(\infty, 1 + 10) = 11$$

$$d_{25}^{(4)} = \min\left(d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)}\right)$$

$$d_{25}^{(4)} = \min(7, 1 + (-2)) = -1$$

$$d_{31}^{(4)} = \min\left(d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)}\right)$$

$$d_{31}^{(4)} = \min(\infty, -5 + 2) = -3$$

$$d_{32}^{(4)} = \min\left(d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)}\right)$$

$$d_{32}^{(4)} = \min(4, -5 + 5) = 0$$

$$D_{ij}^{(4)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ 3 & 0 & 11 & 1 & -1 \\ -3 & 0 & 0 & -5 & -7 \\ 2 & 5 & 10 & 0 & -2 \\ 8 & 11 & 16 & 6 & 0 \end{matrix} \qquad \pi^{(4)} = \begin{matrix} NIL & 1 & 1 & 3 & 1 \\ 4 & NIL & 4 & 2 & 2 \\ 4 & 4 & NIL & 3 & 4 \\ 4 & 1 & 1 & NIL & 1 \\ 4 & 4 & 4 & 5 & NIL \end{matrix}$$

Step (vi) When k = 5

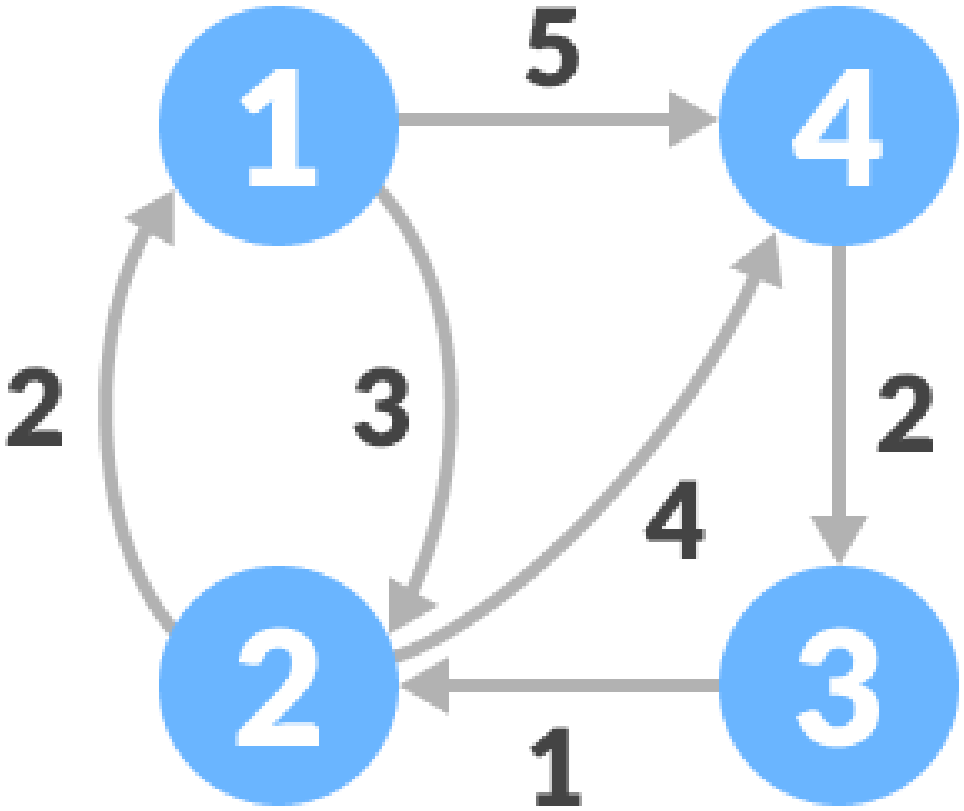
$$D_{ij}^{(5)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ 3 & 0 & 11 & 1 & -1 \\ -3 & 0 & 0 & -5 & -7 \\ 2 & 5 & 10 & 0 & -2 \\ 8 & 11 & 16 & 6 & 0 \end{matrix} \qquad \pi^{(5)} = \begin{matrix} NIL & 1 & 1 & 5 & 1 \\ 4 & NIL & 4 & 2 & 4 \\ 4 & 4 & NIL & 3 & 4 \\ 4 & 1 & 1 & NIL & 1 \\ 4 & 4 & 4 & 5 & NIL \end{matrix}$$

TRANSITIVE- CLOSURE (G)

1. n ← |V[G]|
2. for i ← 1 to n
3. do for j ← 1 to n
4. do if i = j or (i, j) ∈ E [G]
5. the ← 1
6. else ← 0
7. for k ← 1 to n
8. do for i ← 1 to n
9. do for j ← 1 to n
10. dod $_{ij}^{(k)}$ ←
11. Return $T^{(n)}$.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



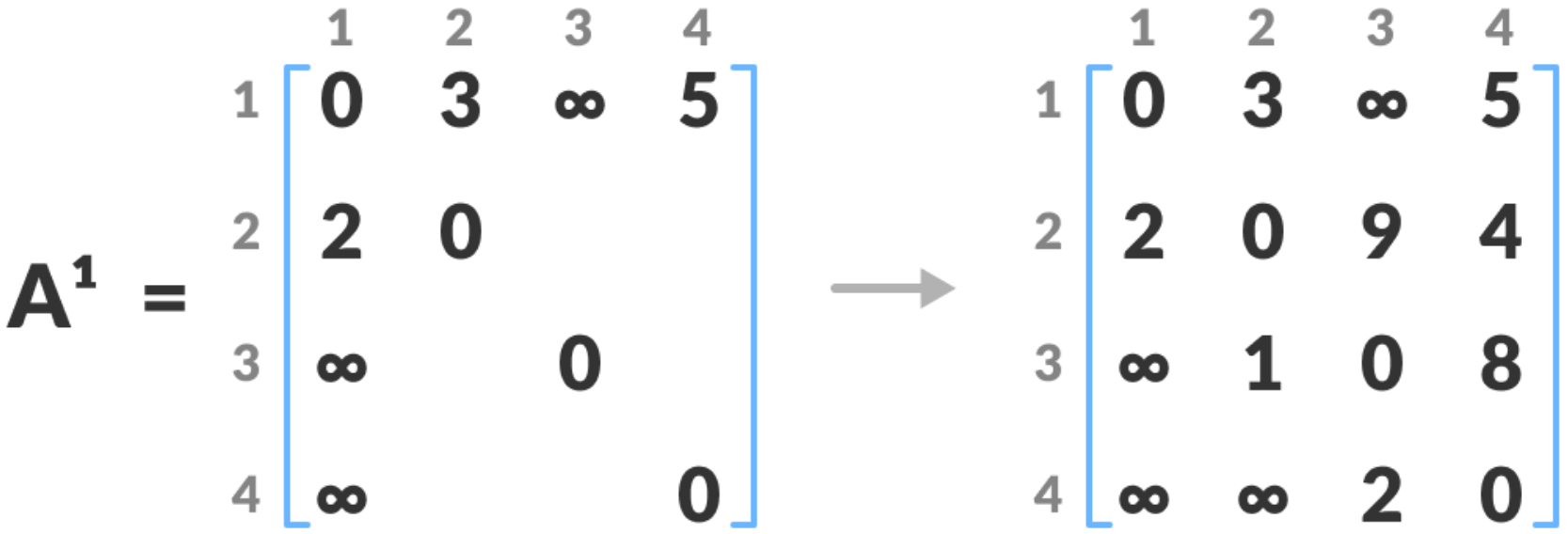
Initial graph

1. Follow the steps below to find the shortest path between all the pairs of vertices.
2. Create a matrix A^1 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.
3. Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Fill each cell with the distance between ith and jth vertex

- Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.
- Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).
- That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].
- In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

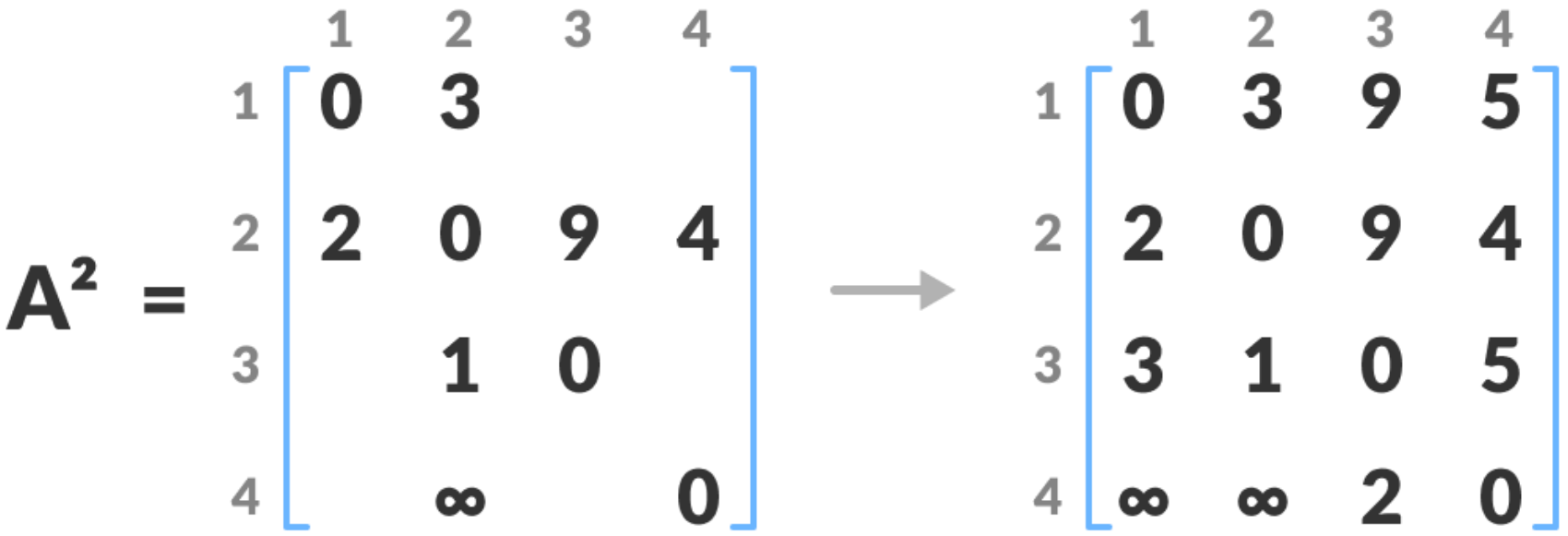


Calculate the distance from the source vertex to destination vertex through this vertex k

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

Similarly, A^2 is created using A^3. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

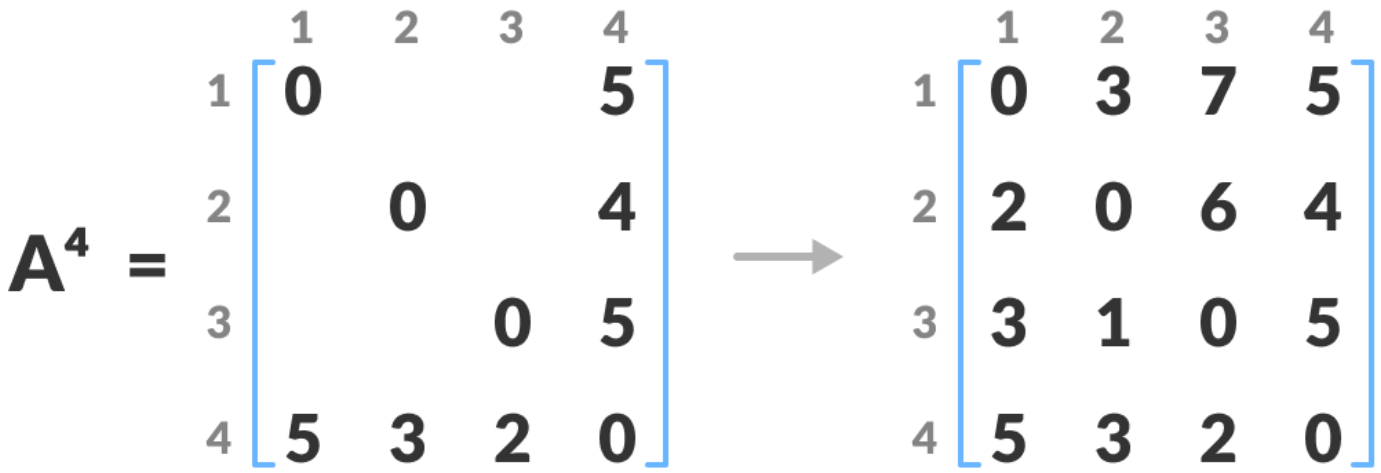


Calculate the distance from the source vertex to destination vertex through this vertex 2

Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 3



Calculate the distance from the source vertex to destination vertex through this vertex 4

A^4 gives the shortest path between each pair of vertices.

Floyd Warshall Algorithm Applications

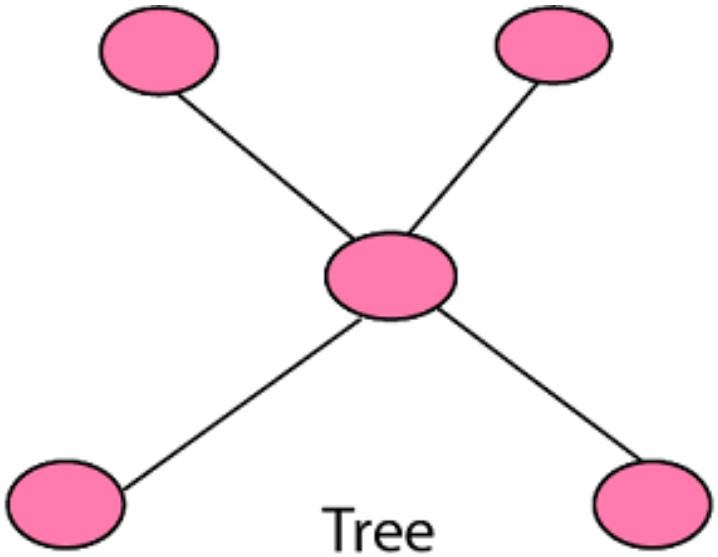
- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite

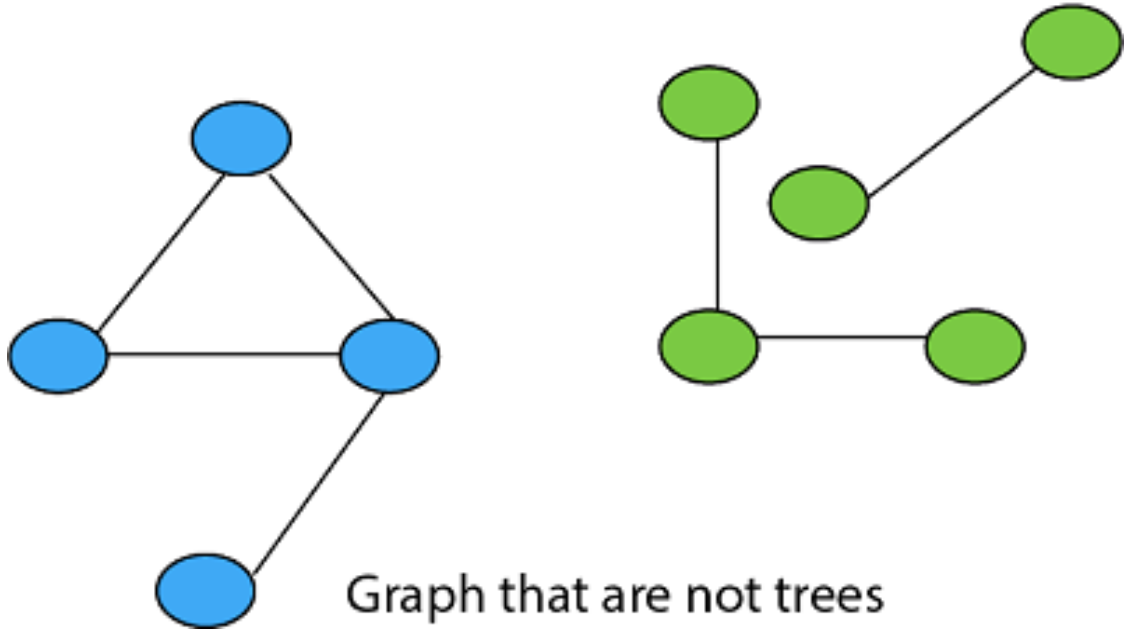
Minimum Spanning Tree

Tree:

A tree is a graph with the following properties:

1. The graph is connected (can go from anywhere to anywhere)
2. There are no cyclic (Acyclic)

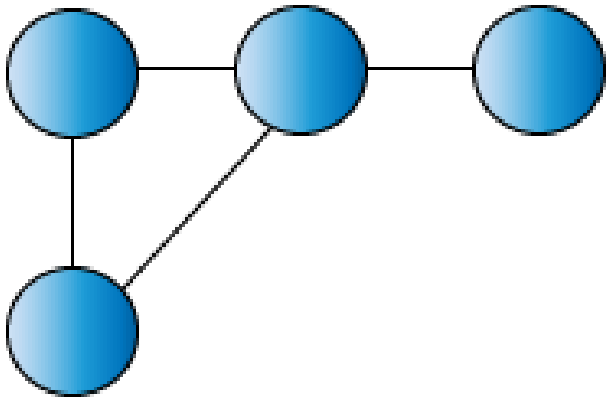

Tree


Graph that are not trees

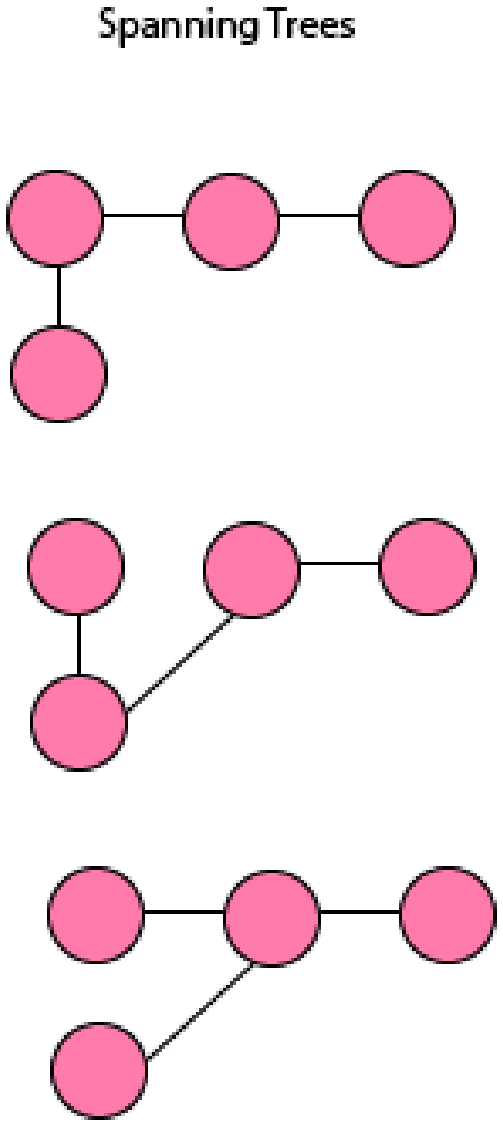
Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joined all vertices. A single graph can have many spanning trees.

Connected Undirected Graph



There can be multiple spanning Trees like

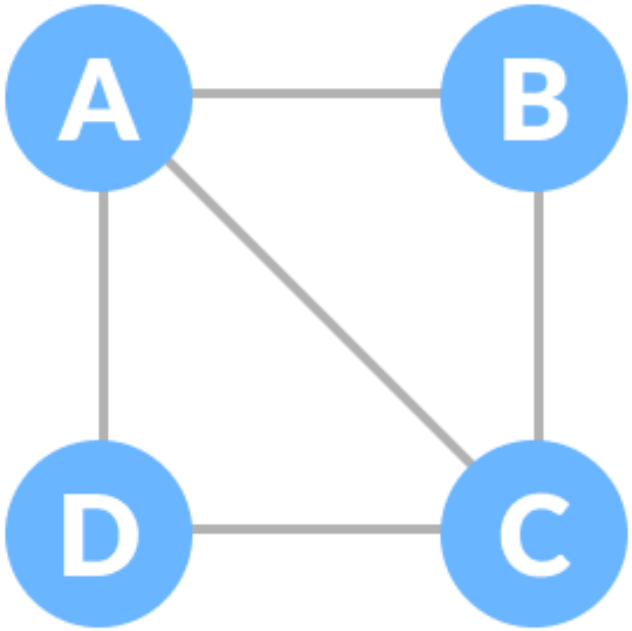


Properties of Spanning Tree:

1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.
6. Spanning Tree doesn't contain cycles.
7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

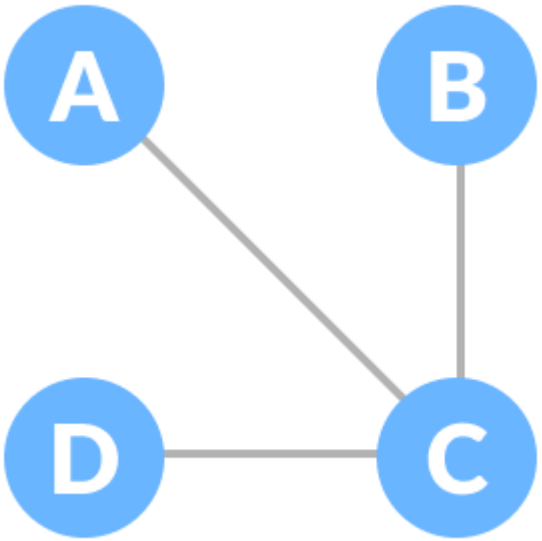
Addition of even one single edge results in the spanning tree losing its property of Acyclicity and elimination of one single edge results in its losing the property of connectivity.

An **undirected graph** is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Undirected Graph

A **connected graph** is a graph in which there is always a path from a vertex to any other vertex.



Connected Graph

Spanning tree

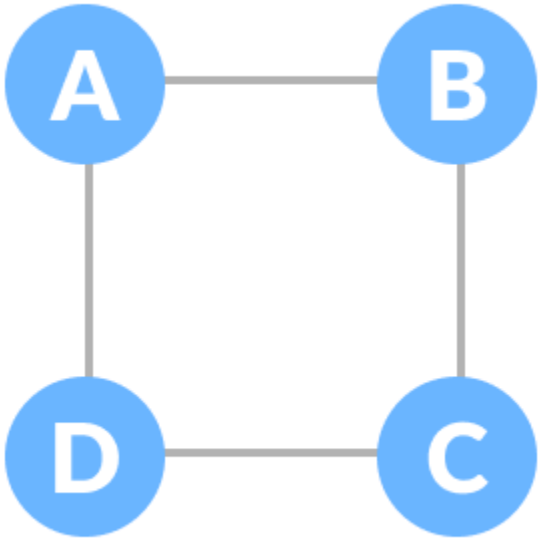
A spanning tree is a sub-graph of an undirected and a connected graph, which includes all the vertices of the graph having a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

The edges may or may not have weights assigned to them.

The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$.

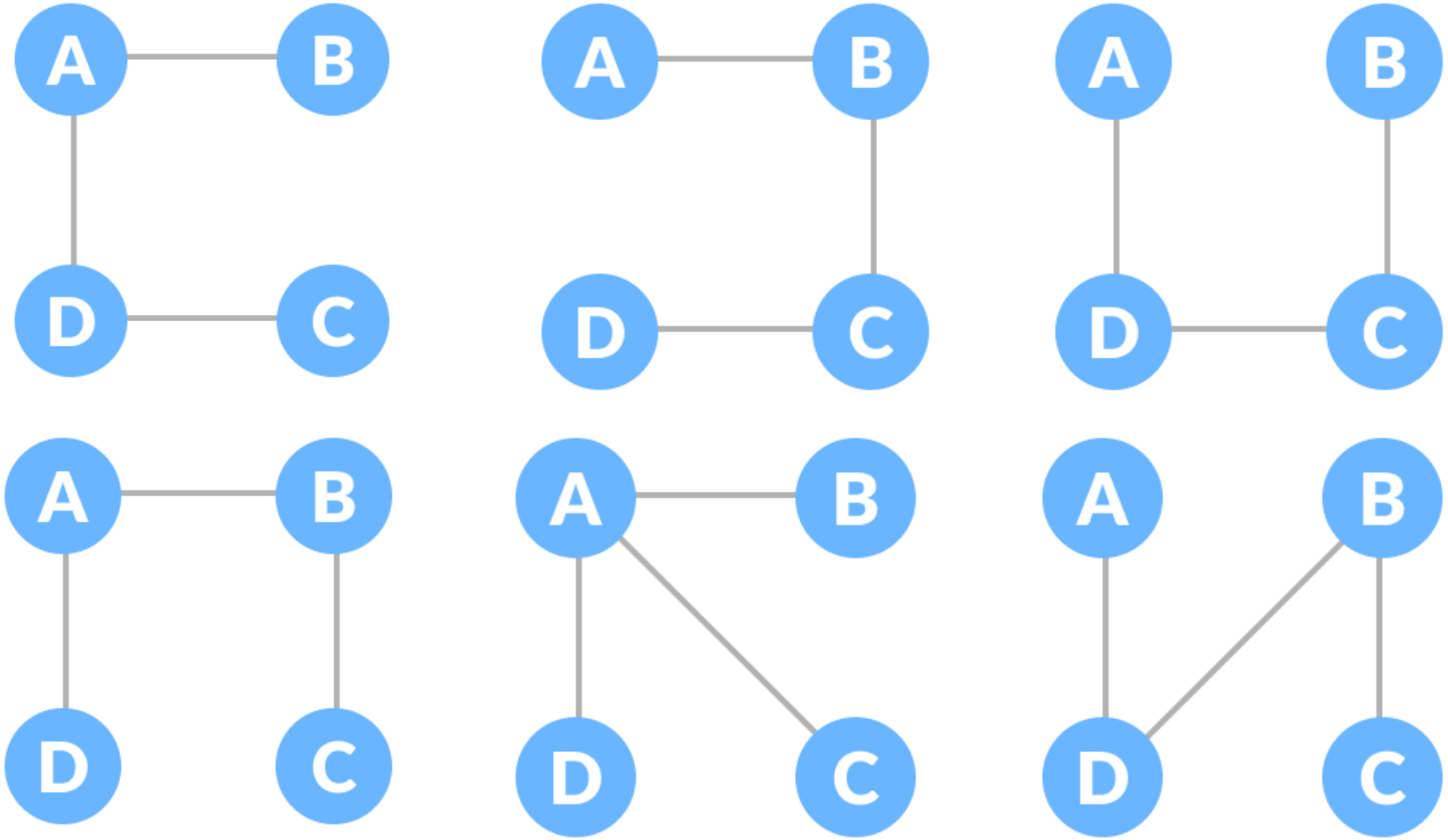
If we have n = 4, the maximum number of possible spanning trees is equal to $4^{4-2} = 16$. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

Let the original graph be:



Normal graph

Some of the possible spanning trees that can be created from the above graph are:

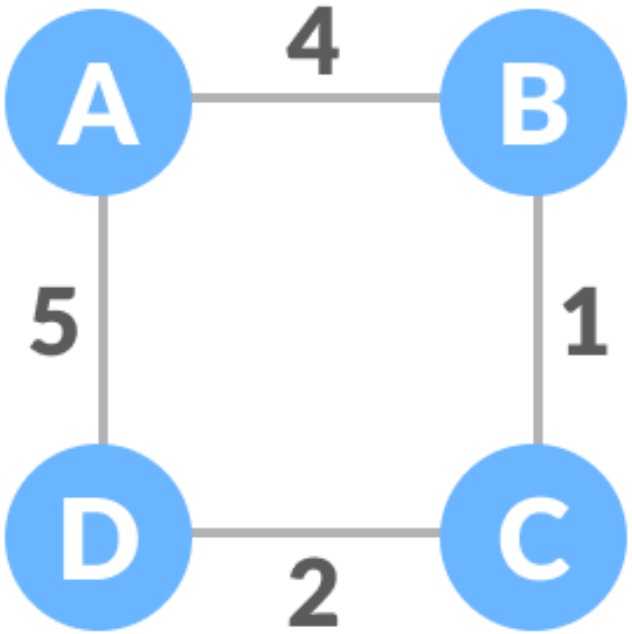


A spanning tree

Minimum Spanning Tree

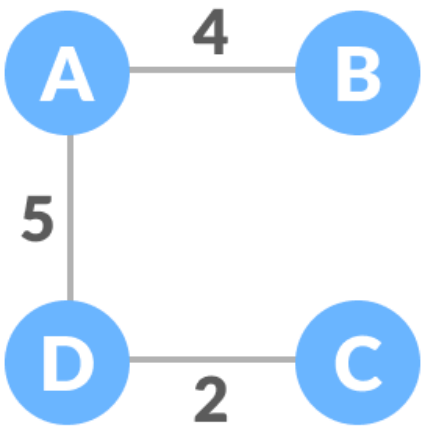
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

The initial graph is:



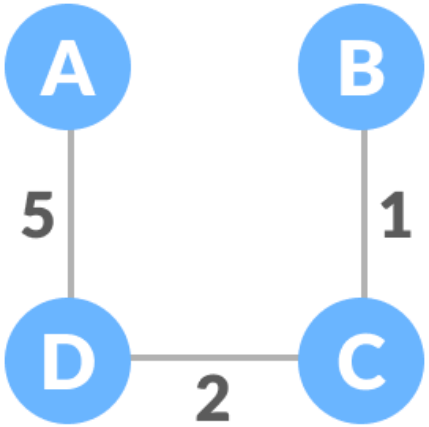
Weighted graph

The possible spanning trees from the above graph are:



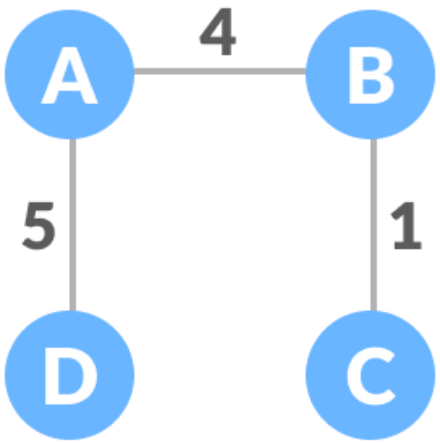
sum = 11

MST - 1



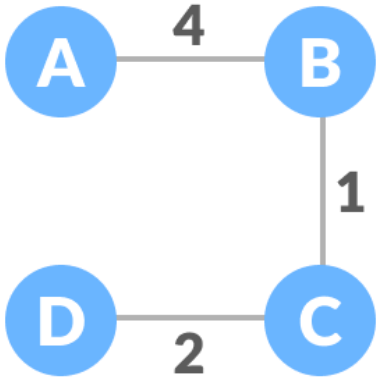
sum = 8

MST - 2



sum = 10

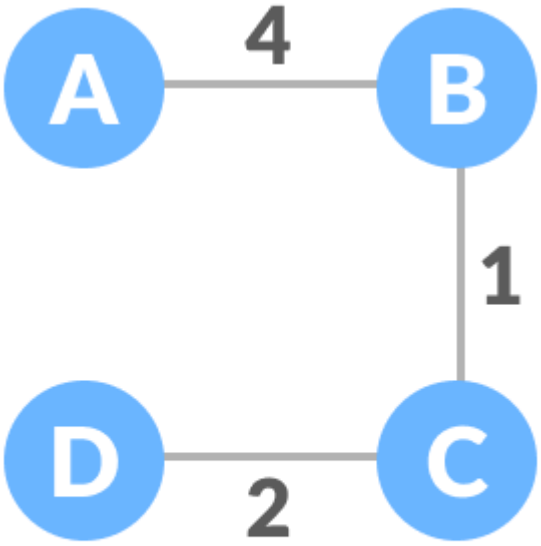
MST - 3



sum = 7

MST - 4

The minimum spanning tree from the above spanning trees is:



sum = 7

Minimum spanning tree

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

Spanning Tree Applications

1. Computer Network Routing Protocol
2. Cluster Analysis
3. Civil Network Planning

Minimum Spanning tree Applications

1. To find paths in the map
2. To design networks like telecommunication networks, water supply networks, and electrical grids.

Application of Minimum Spanning Tree

1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost.

   The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose you want to apply a set of houses with
   - Electric Power
   - Water
   - Telephone lines
   - Sewage lines

To reduce cost, you can connect houses with minimum cost spanning trees.

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

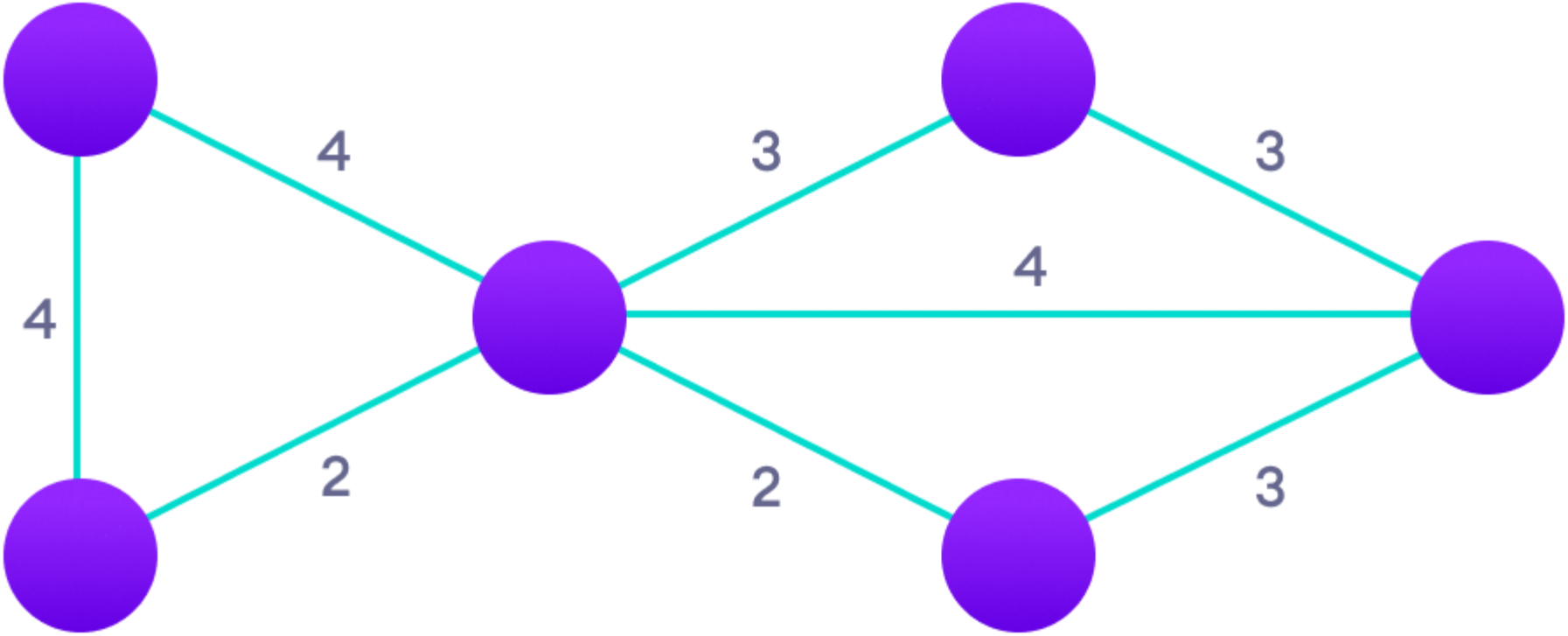
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

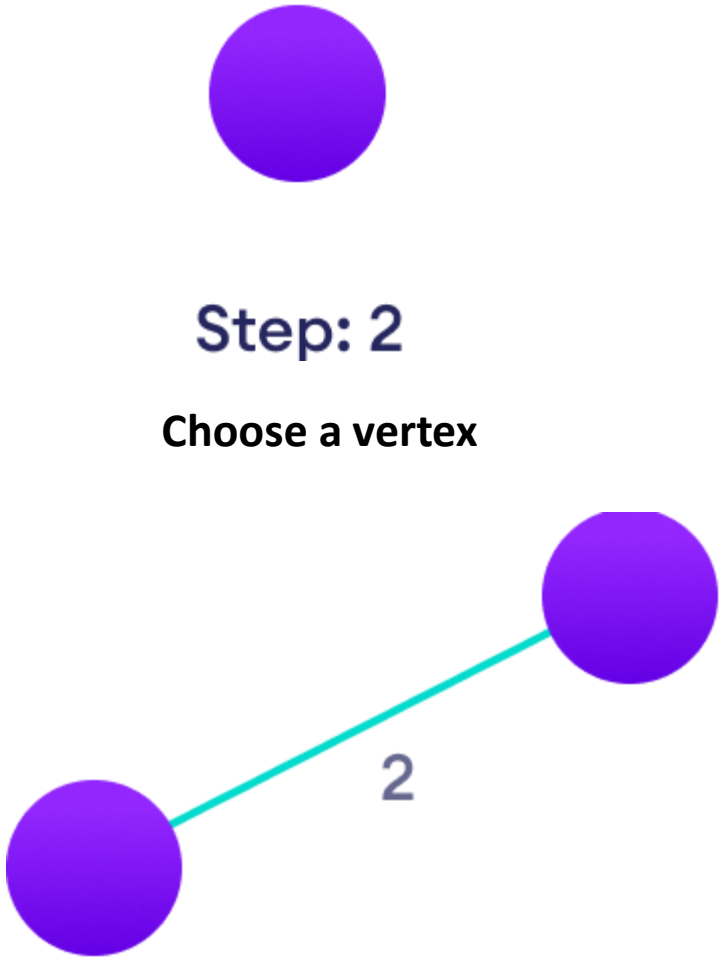
1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Example

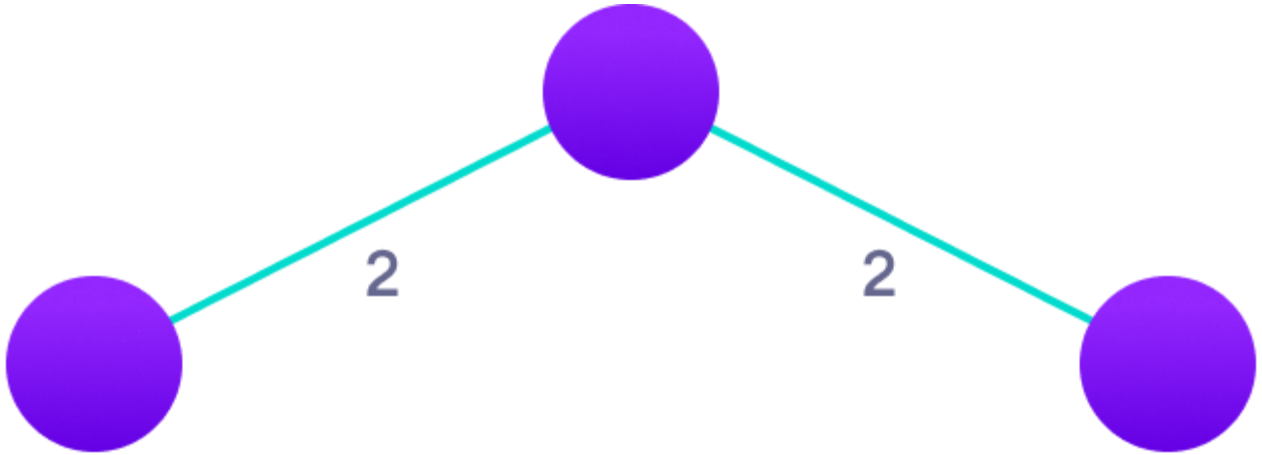


Step: 1

Start with a weighted graph

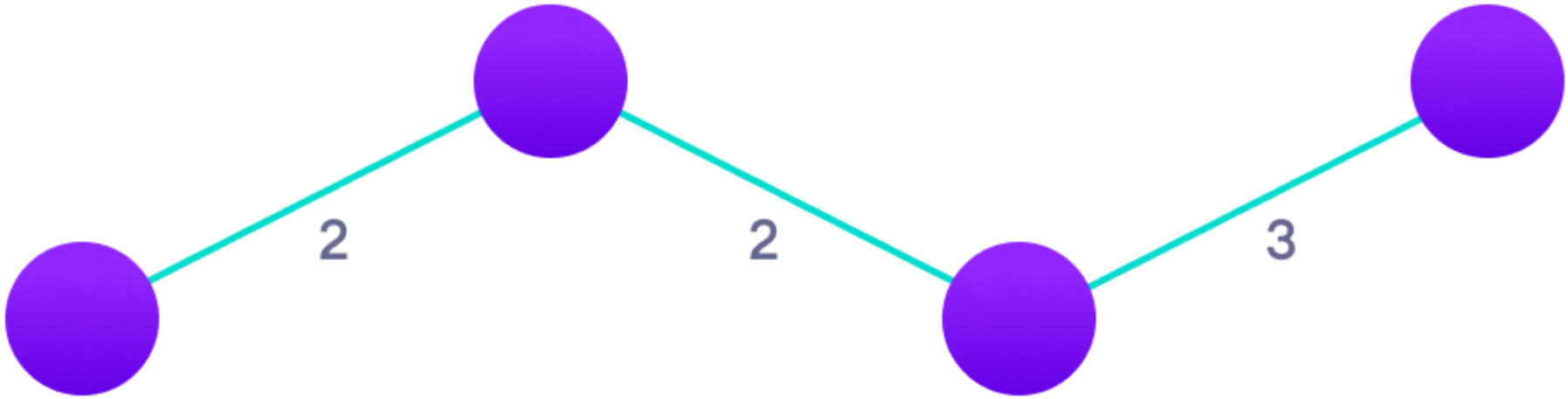


Choose the shortest edge from this vertex and add it



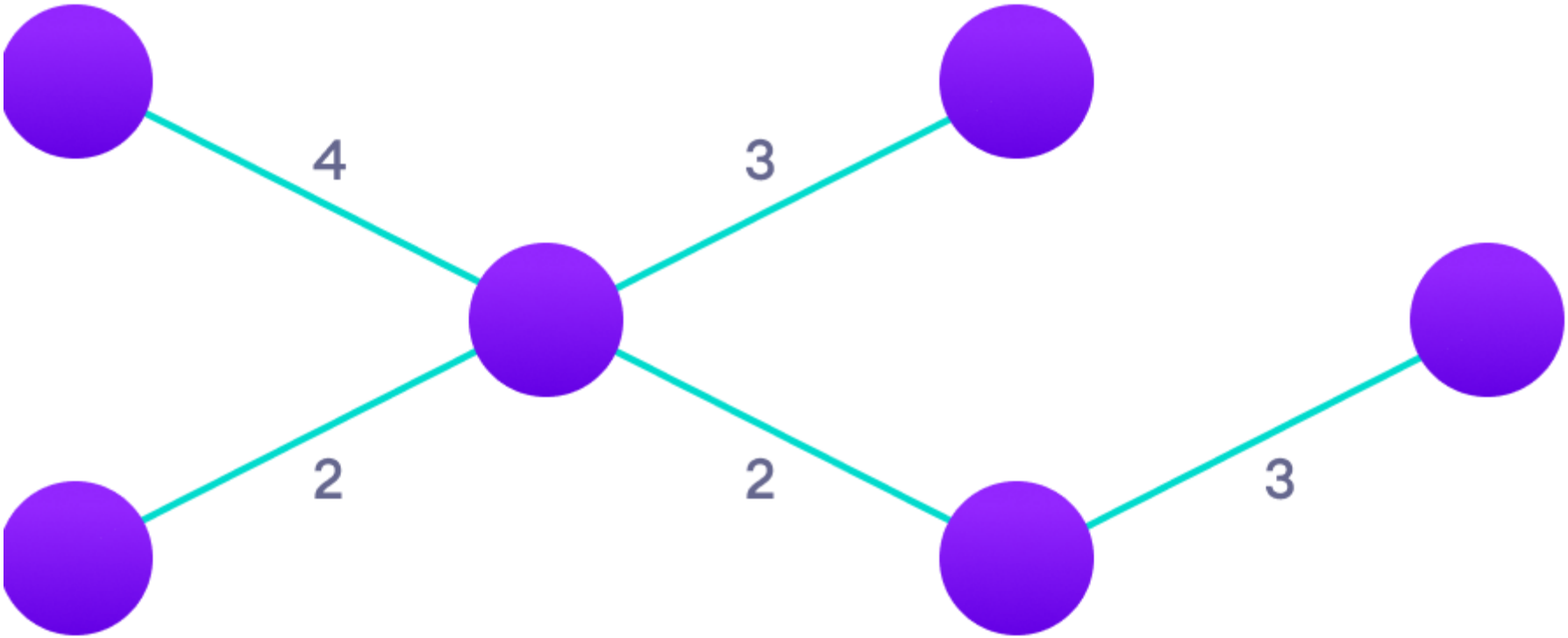
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

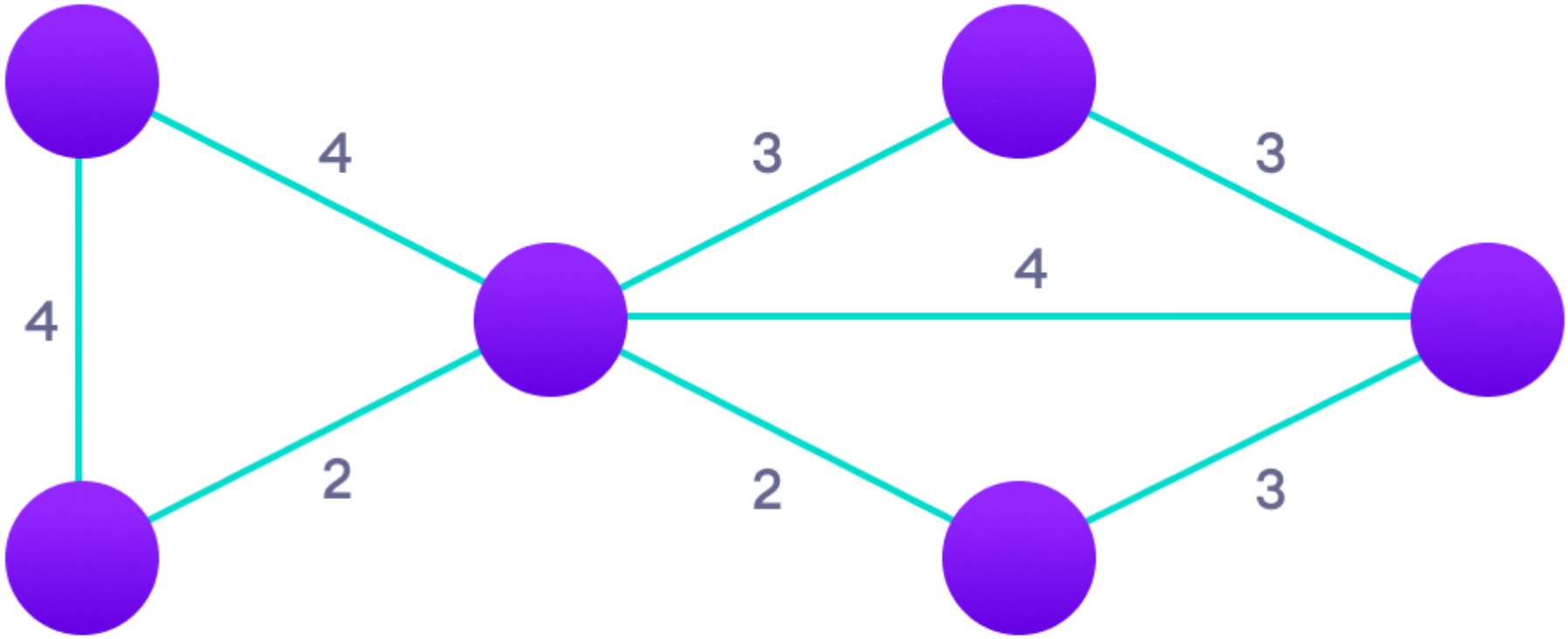
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

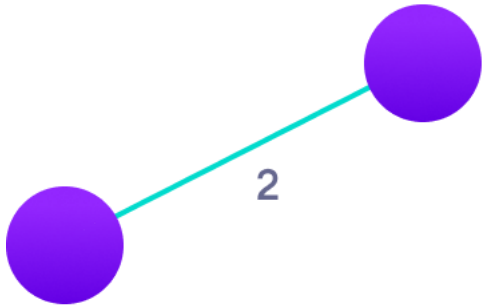
1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example



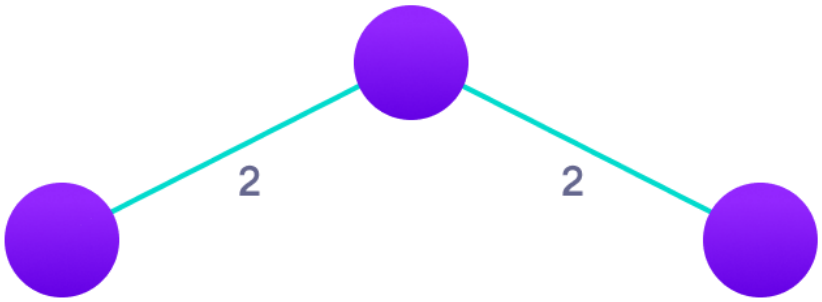
Step: 1

Start with a weighted graph



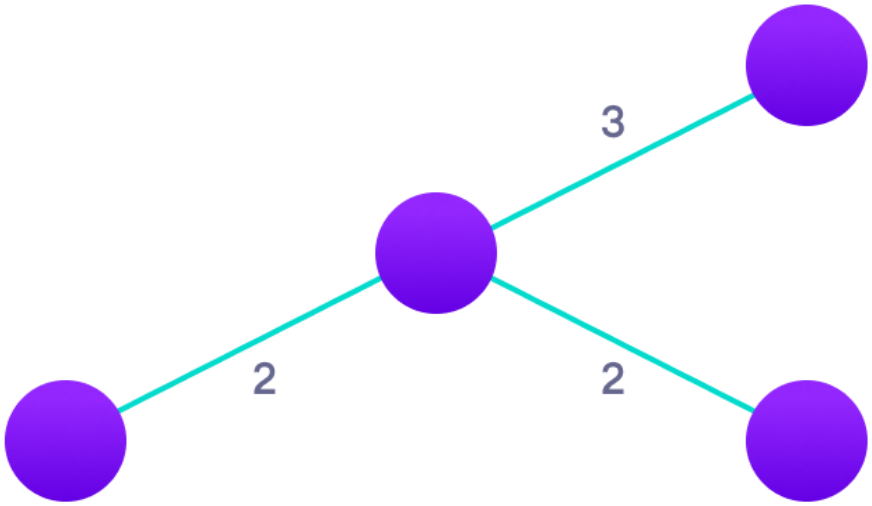
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



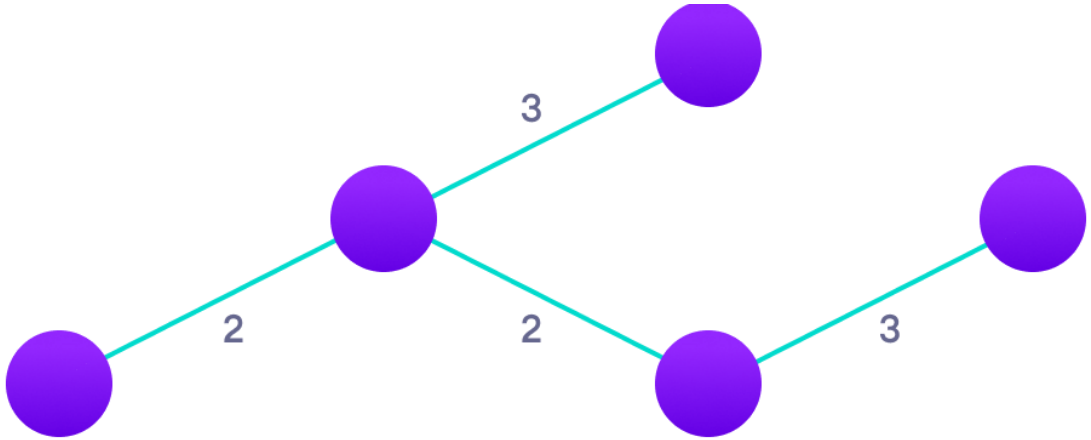
Step: 3

Choose the next shortest edge and add it



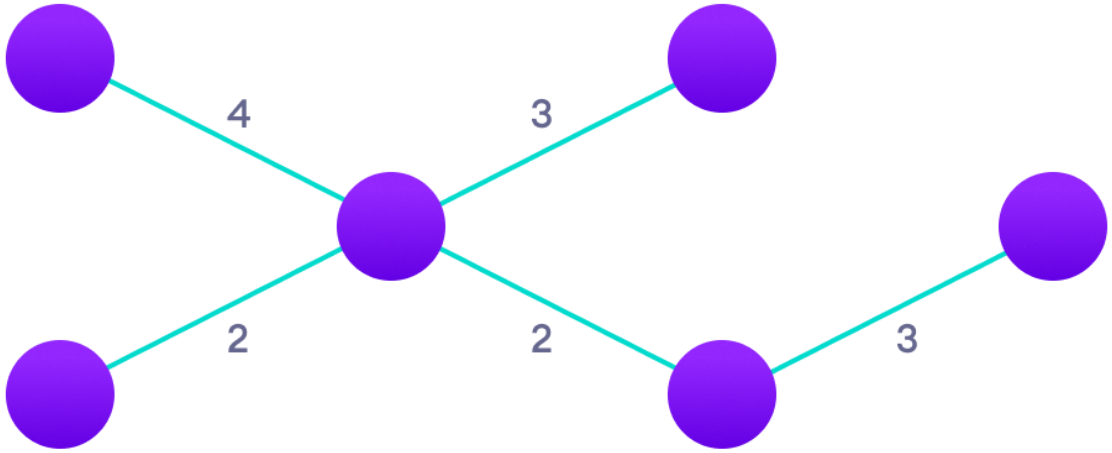
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree