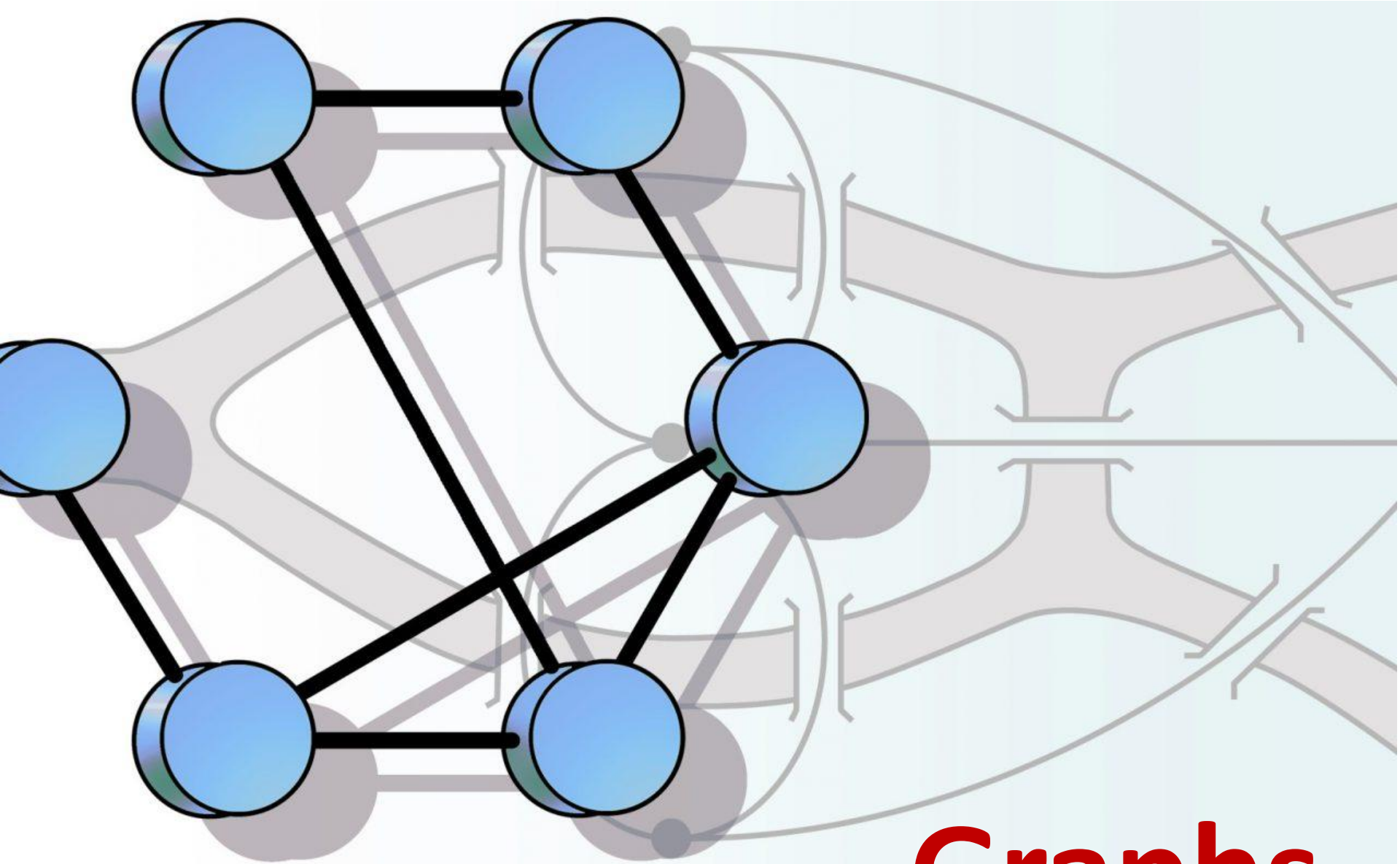
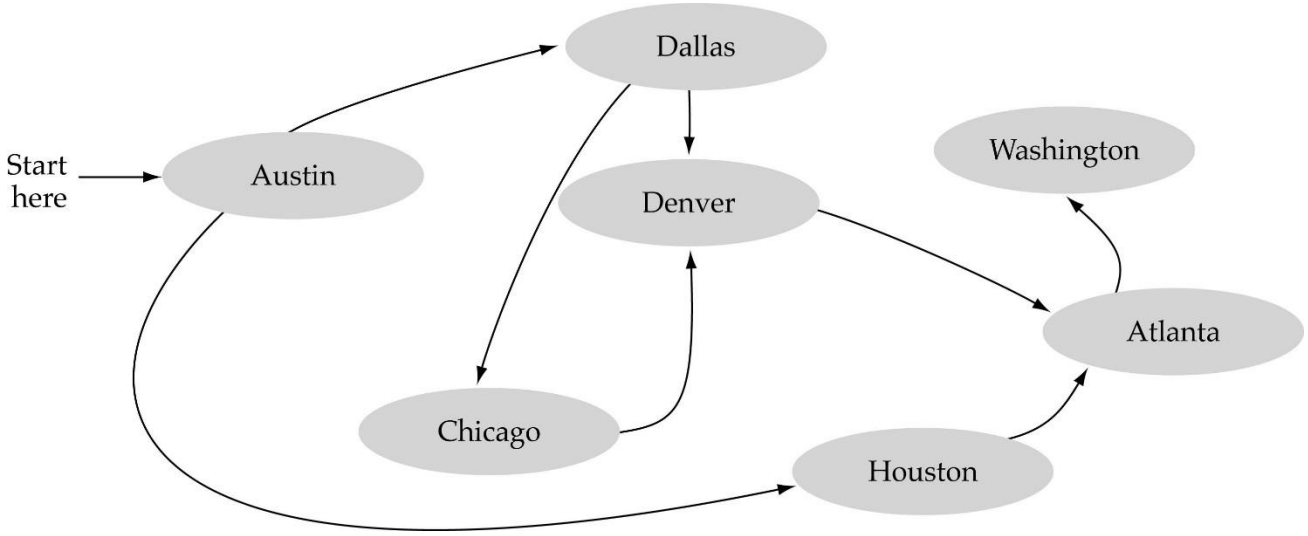


Graphs

Graph theory in mathematics means the study of graphs. Graphs are one of the prime objects of study in discrete mathematics. In general, a graph is represented as a set of vertices (nodes or points) connected by edges (arcs or line). Graphs are therefore mathematical structures used to model pairwise relations between objects. They are found on road maps, constellations, when constructing schemes and drawings. Graphs underlie many computer programs that make modern communication and technological processes possible. They contribute to the development of thinking, both logical and abstract. For example, maybe you remember this game from your childhood: connect the dots on the piece of paper to make a figure, a dog or a cat – those connections are also graphs.

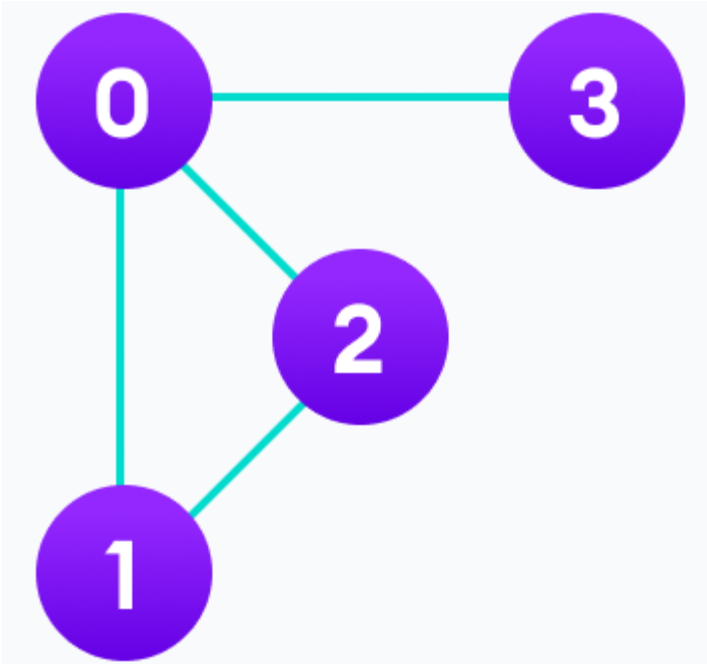
What is a graph?

- 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

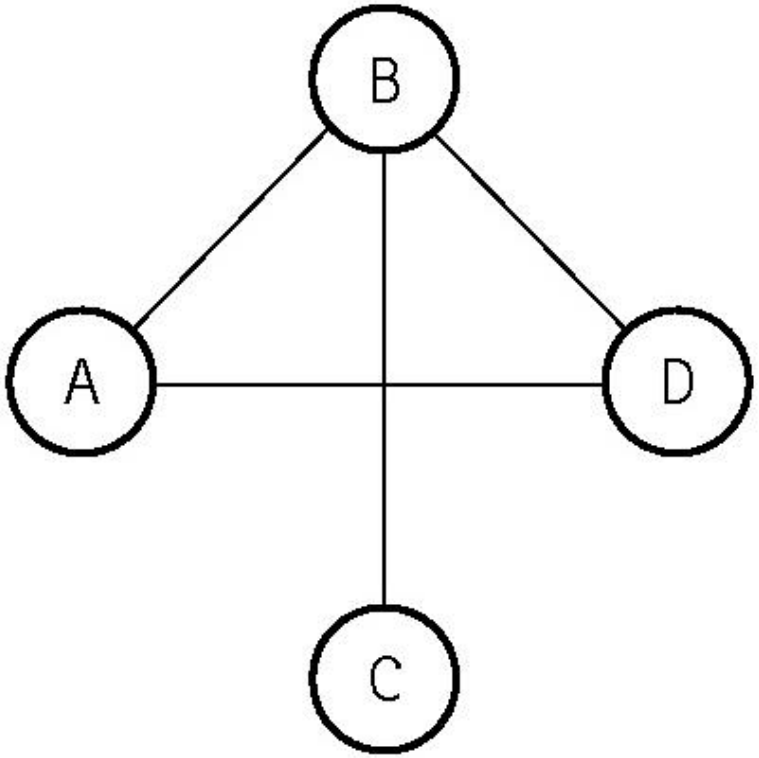
- a graph G is defined as follows:

$$G = (V,E)$$

- 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!!

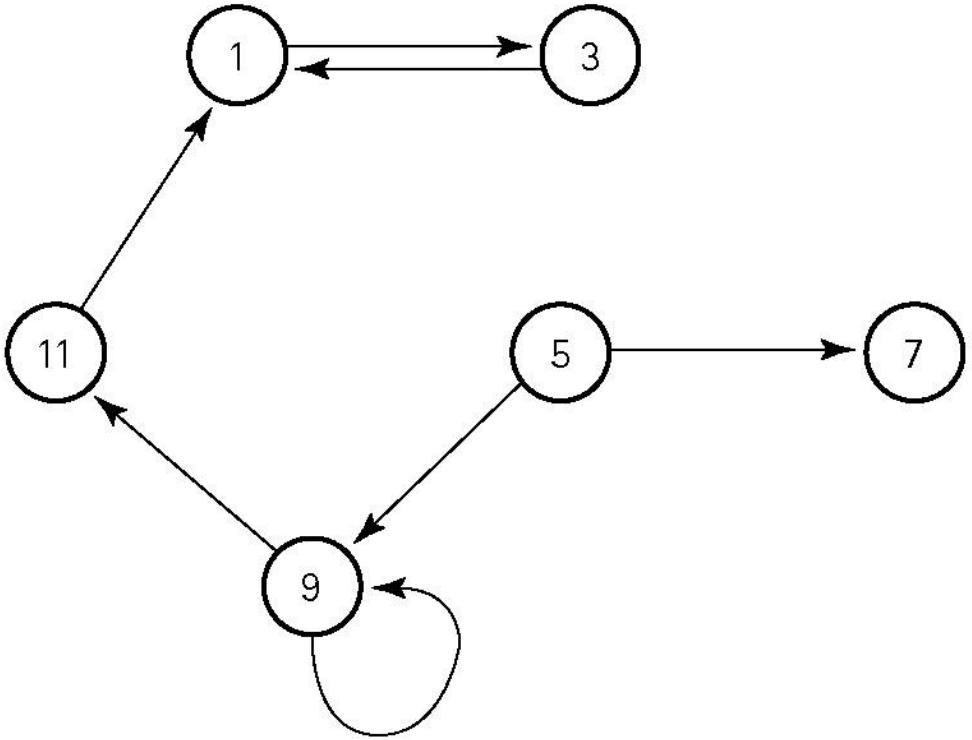
V(Graph1) = {A, B, C, D}
E(Graph1) = {(A, B), (A, D), (B, C), (B, D)}

A Directed Graph

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

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

(b) Graph2 is a directed graph.

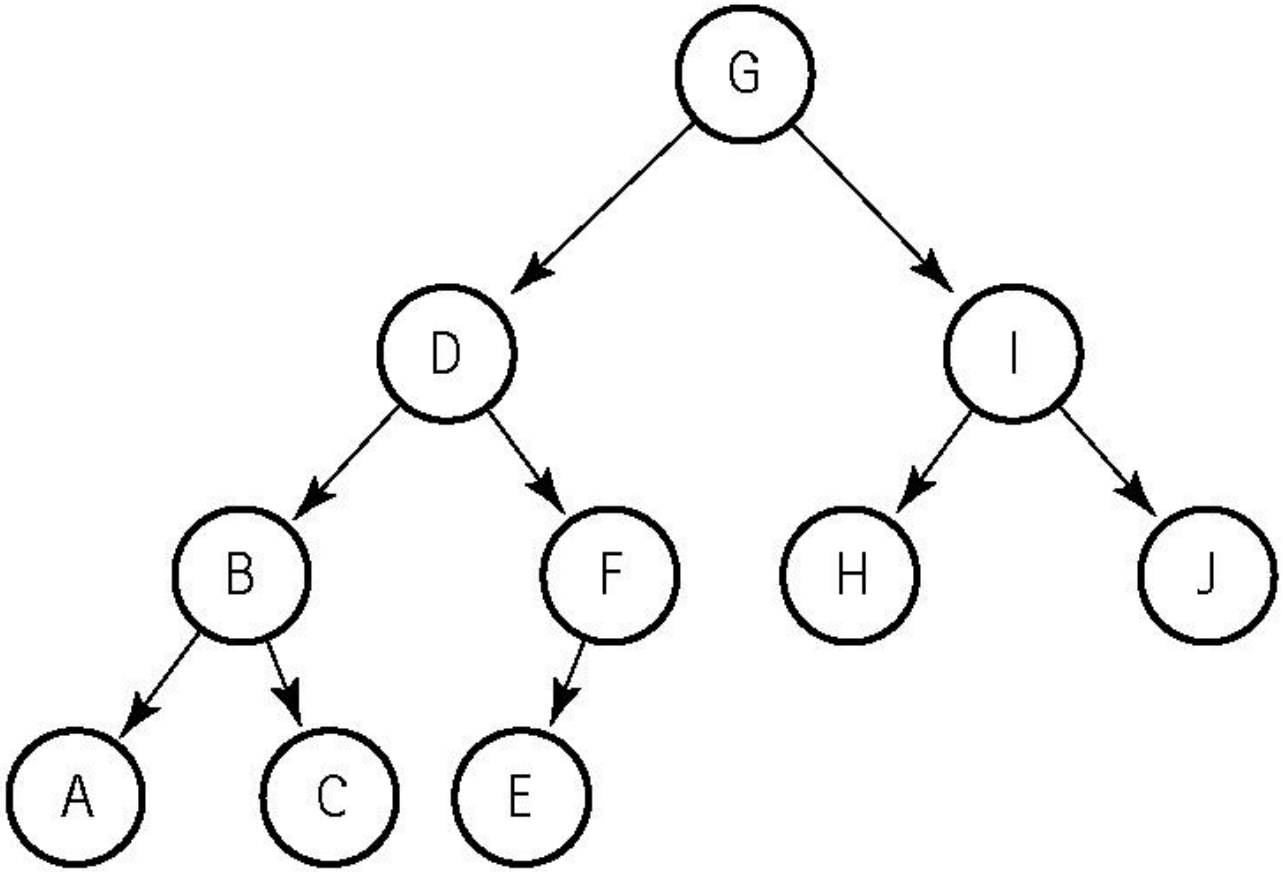


V(Graph2) = {1, 3, 5, 7, 9, 11}
E(Graph2) = {(1, 3), (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)}

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)}

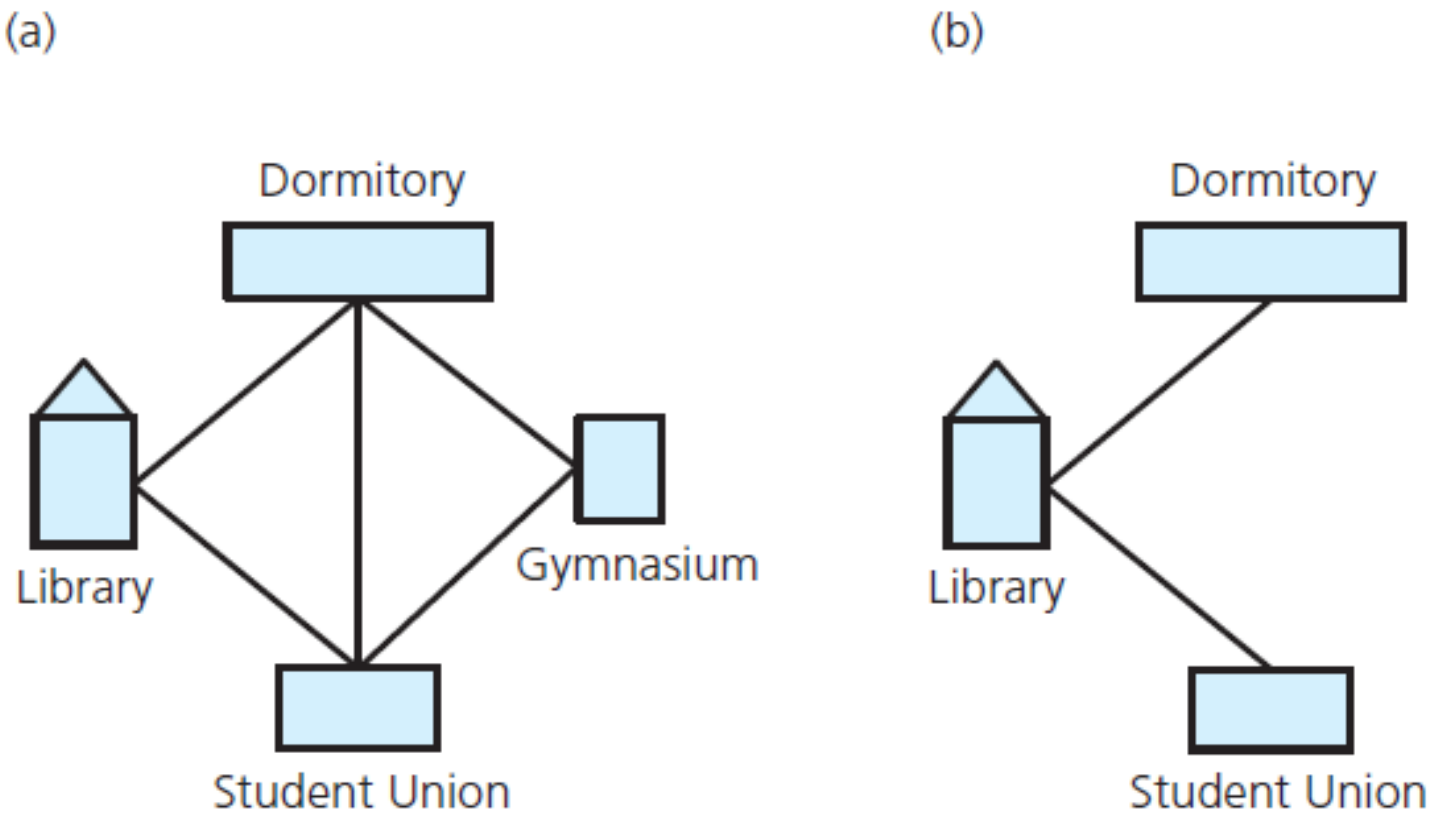
Terminology

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

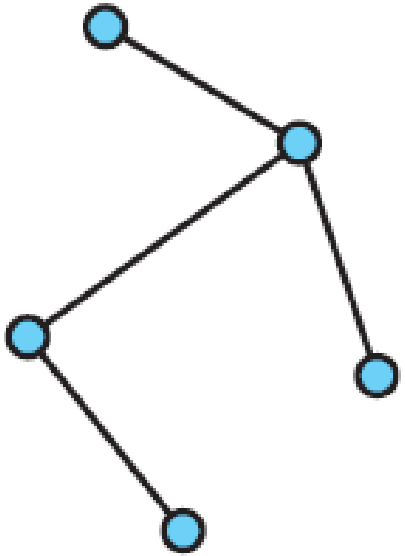
Terminology

(a) A campus map as a graph

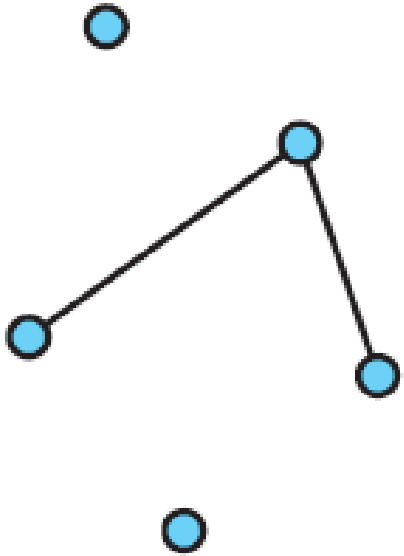
(b) A subgraph



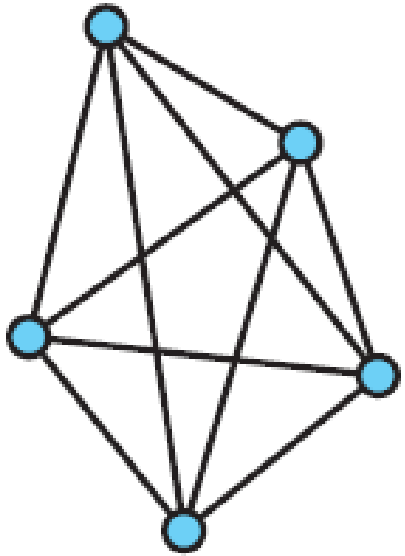
Terminology



(a)



(b)



(c)

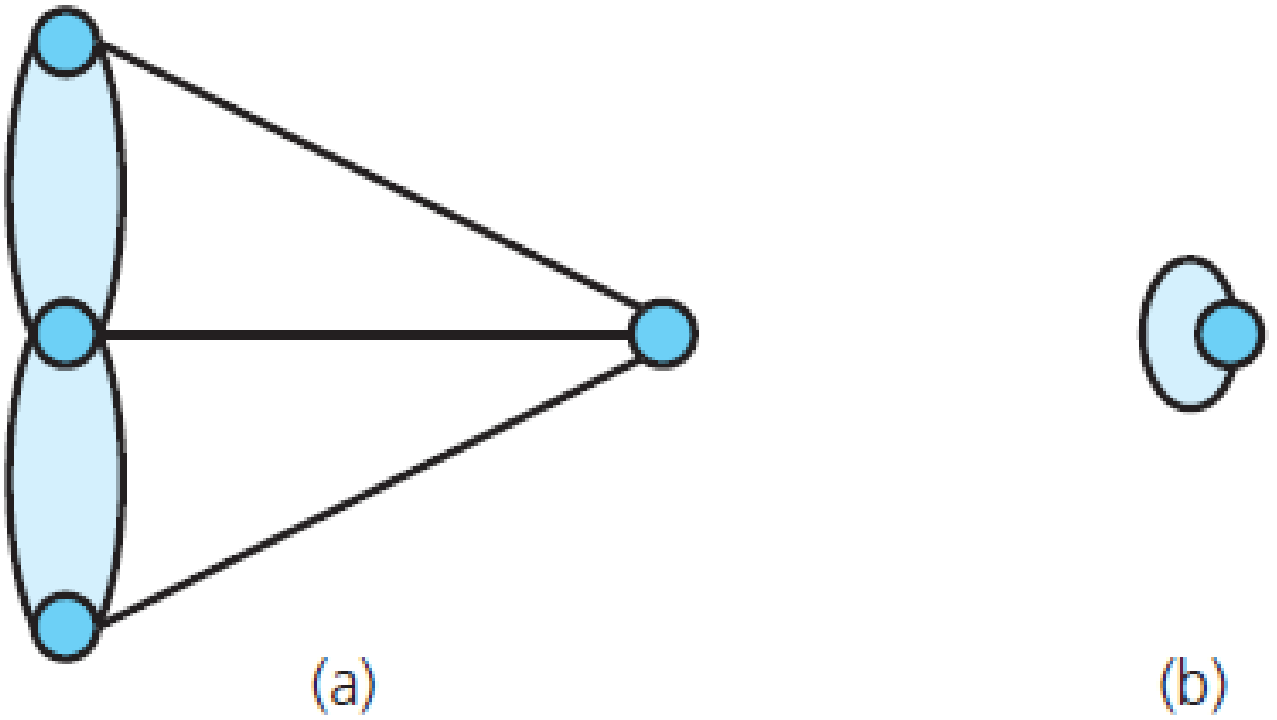
Graphs that are

(a) connected

(b) disconnected and

(c) complete

Terminology

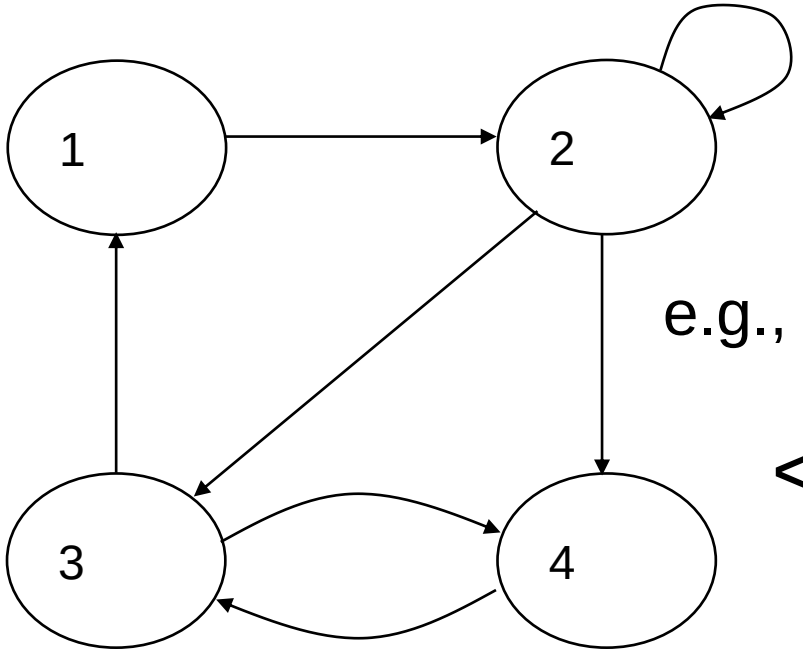


(a) A multigraph is not a simple graph;

(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

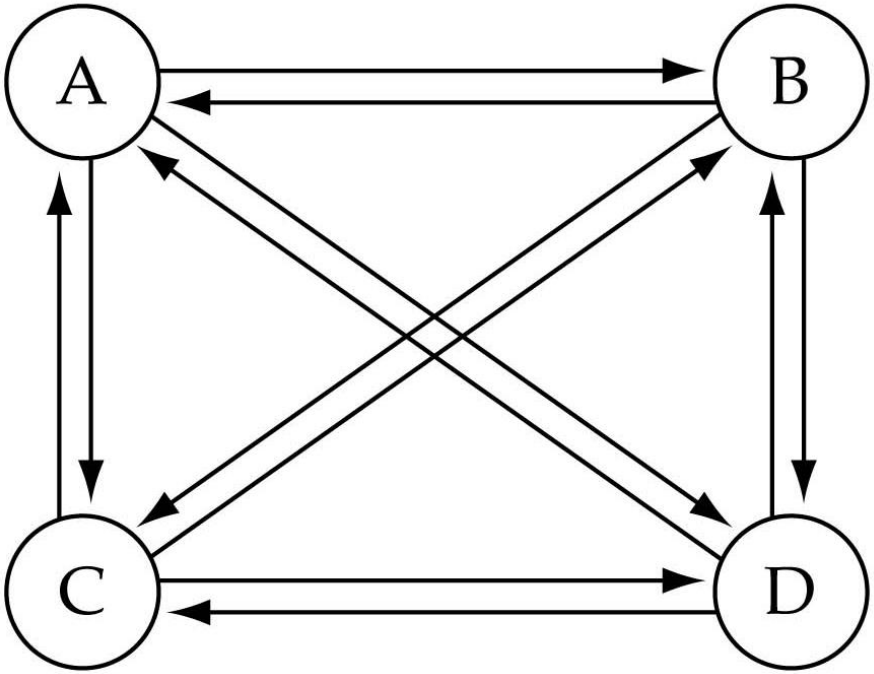
<1, 2, 3, 4>

Terminology

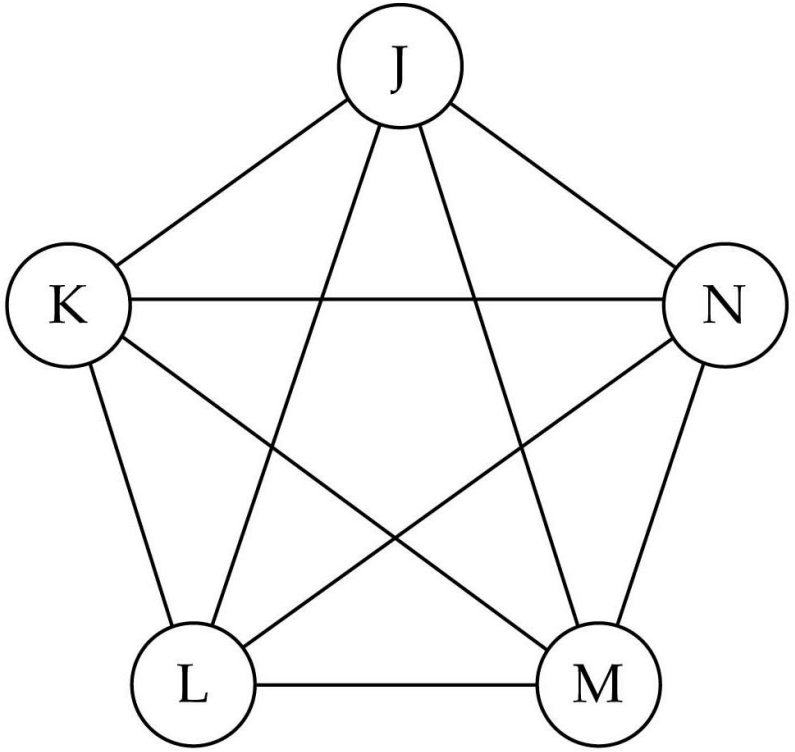
- **Simple path**: passes through vertex only once
- **Cycle**: a path that begins and ends at same vertex
- **Simple cycle**: cycle that does not pass through other vertices more than once
- **Connected graph**: each pair of distinct vertices has a path between them

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

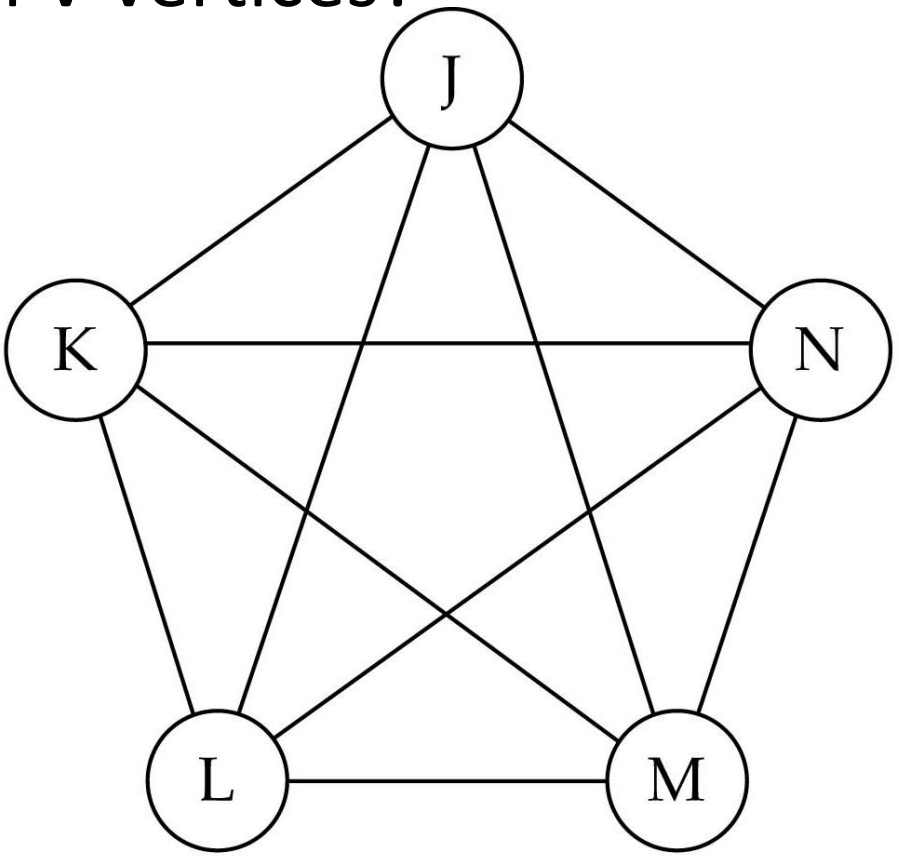
Terminology

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.

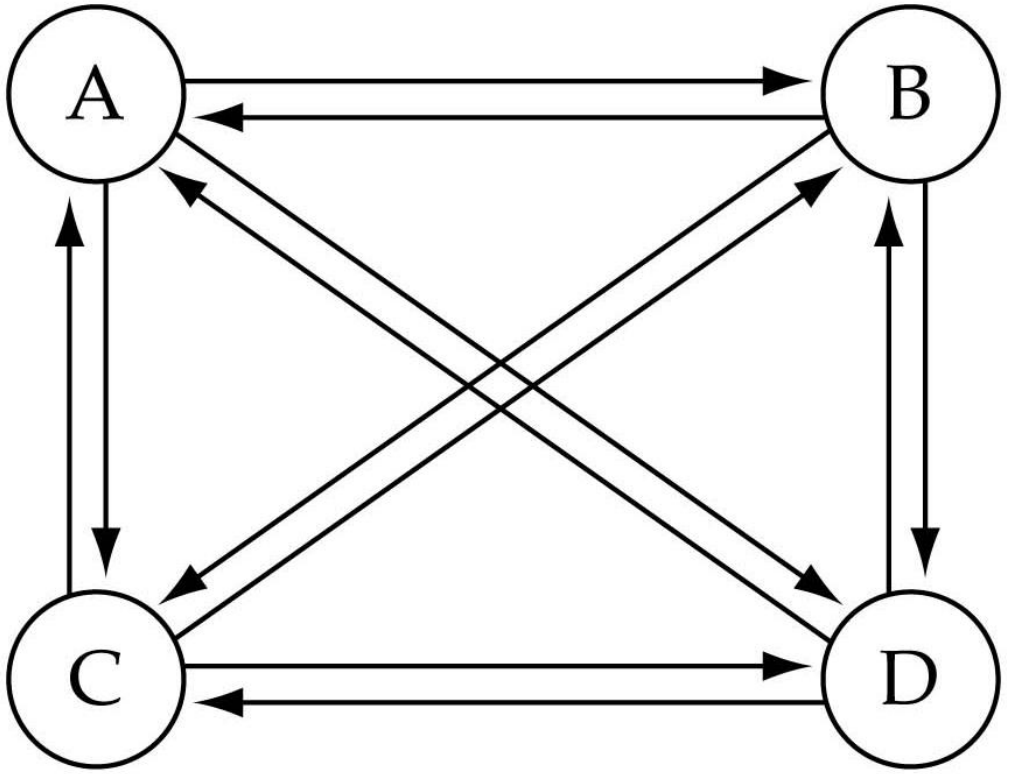
Terminology

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

E=V * (V-1)

or

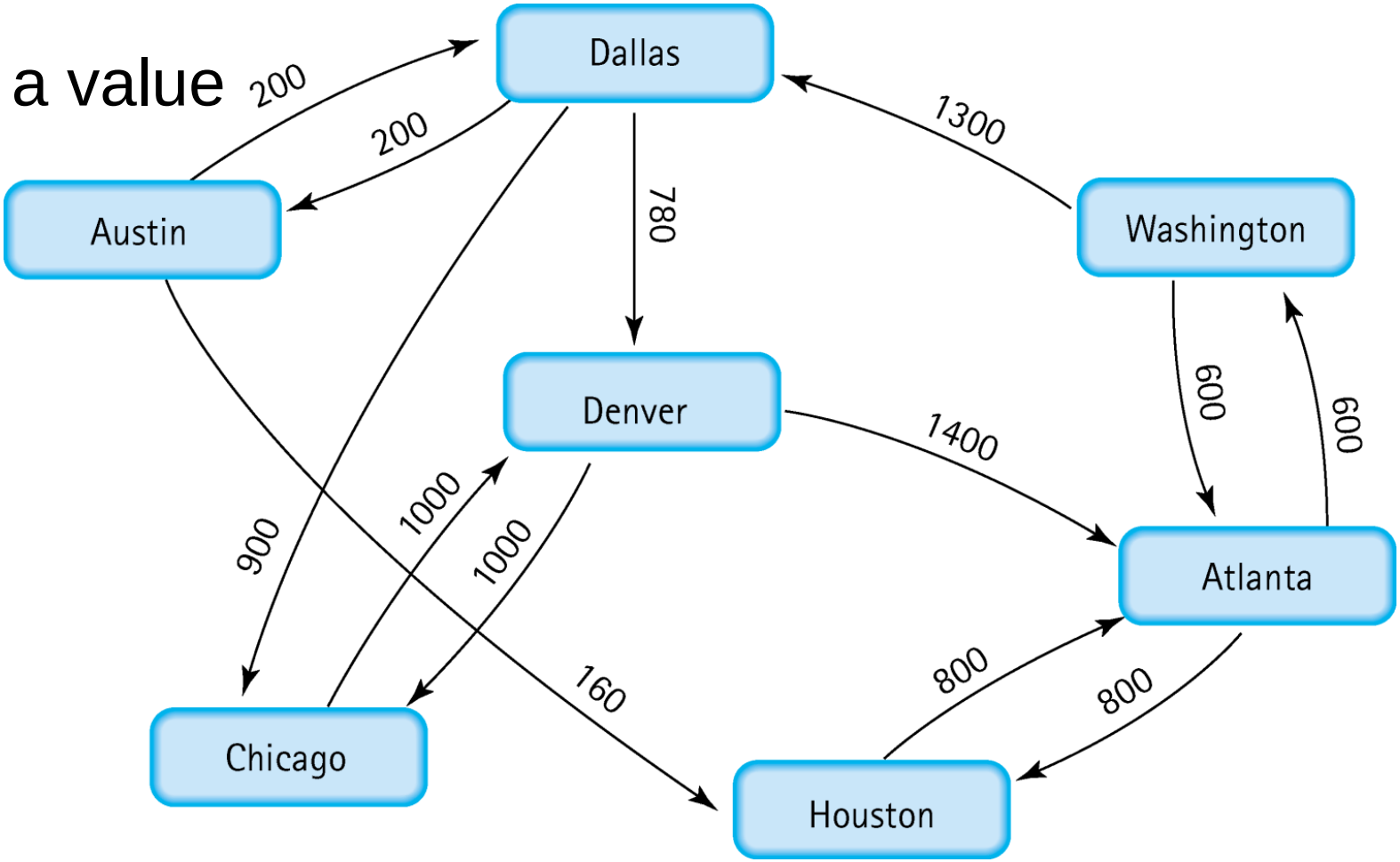
O(V^2)



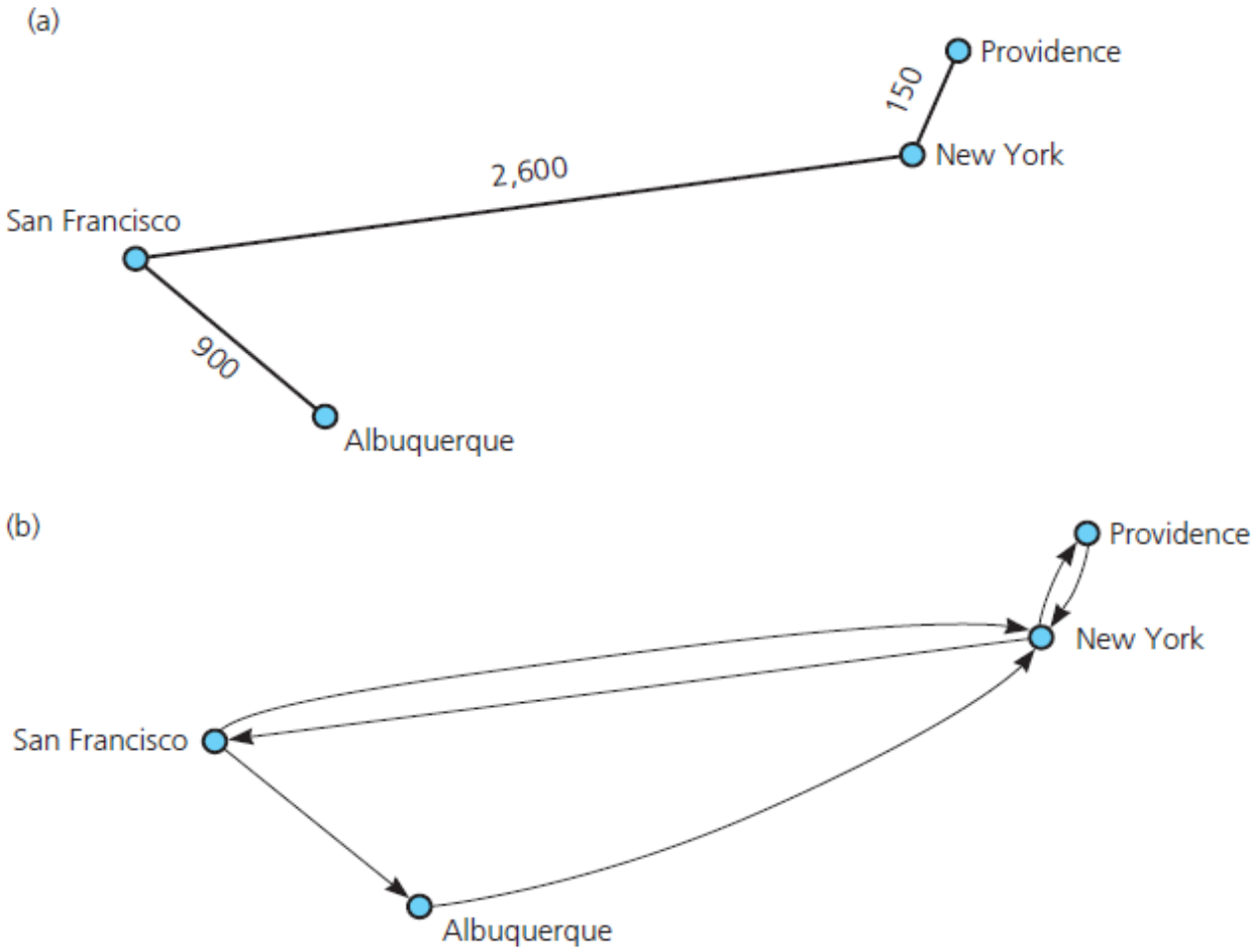
(a) Complete directed graph.

A Weighted Graph

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



Terminology



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

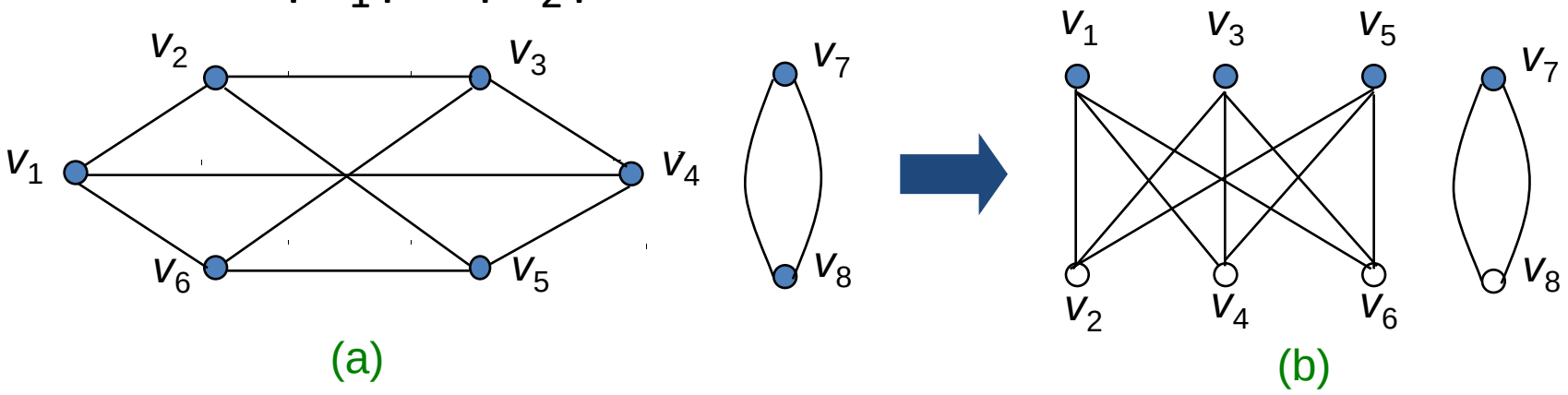
Bipartite Graph

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.

Graphs as ADTs

ADT graph operations

- Test whether graph is empty.
- Get number of vertices in a graph.
- Get number of edges in a graph.
- See whether edge exists between two given vertices.
- Insert vertex in graph whose vertices have distinct values that differ from new vertex's value.
- Insert edge between two given vertices in graph.

Graphs as ADTs

ADT graph operations.

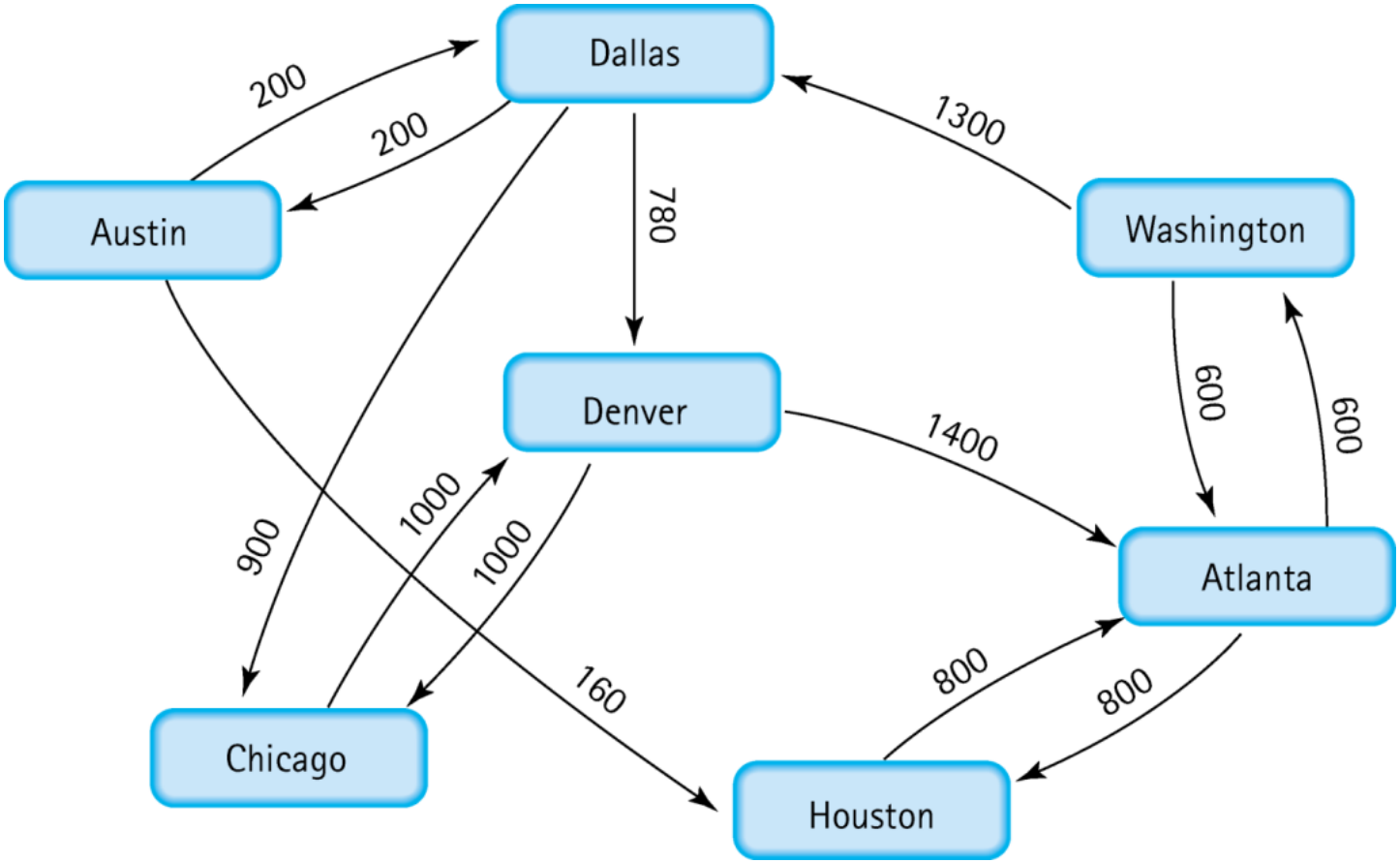
- Remove specified vertex from graph and any edges between the vertex and other vertices.
- Remove edge between two vertices in graph.
- Retrieve from graph vertex that contains given value.

- View interface for undirected, connected graphs

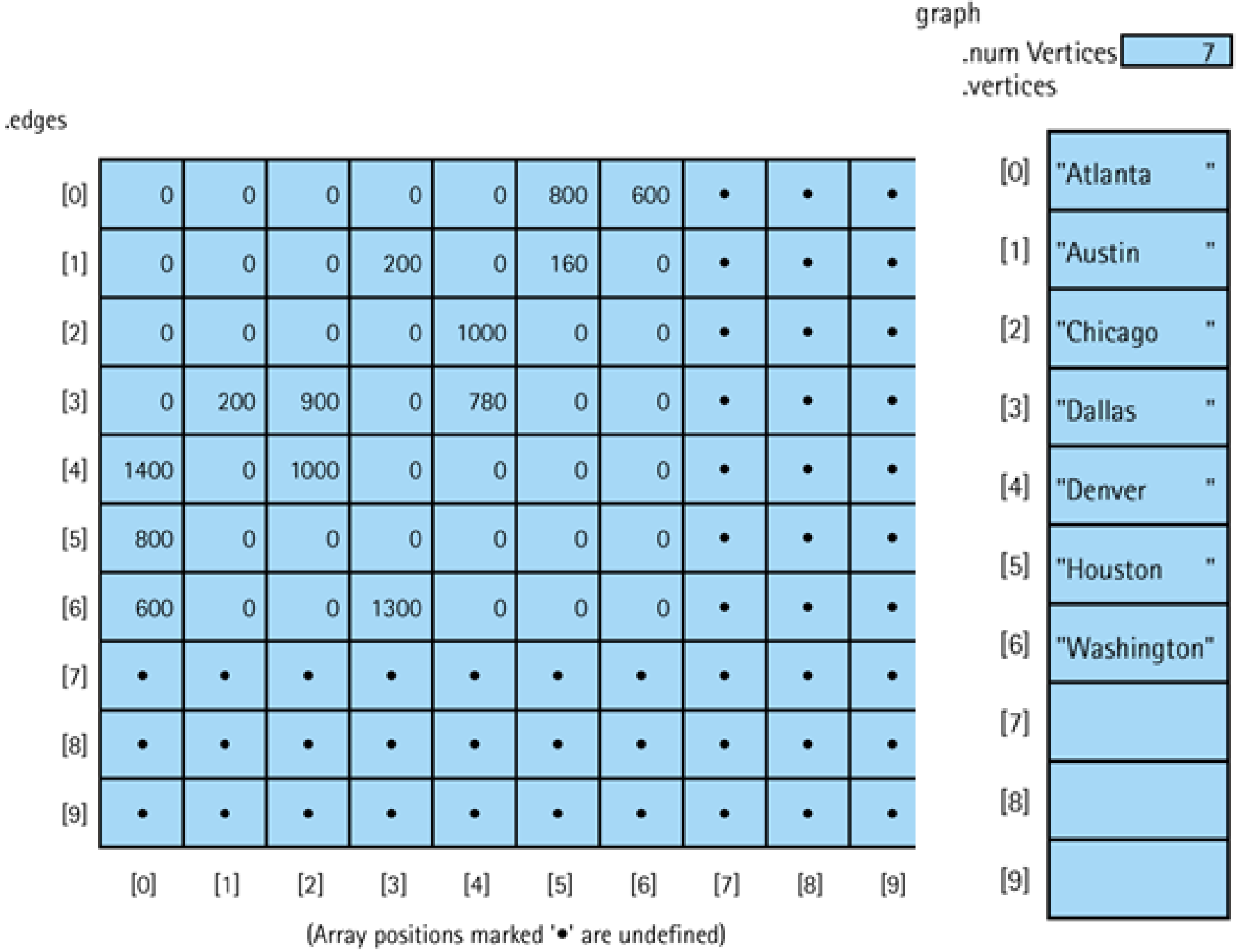
Array-Based Implementation

- 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

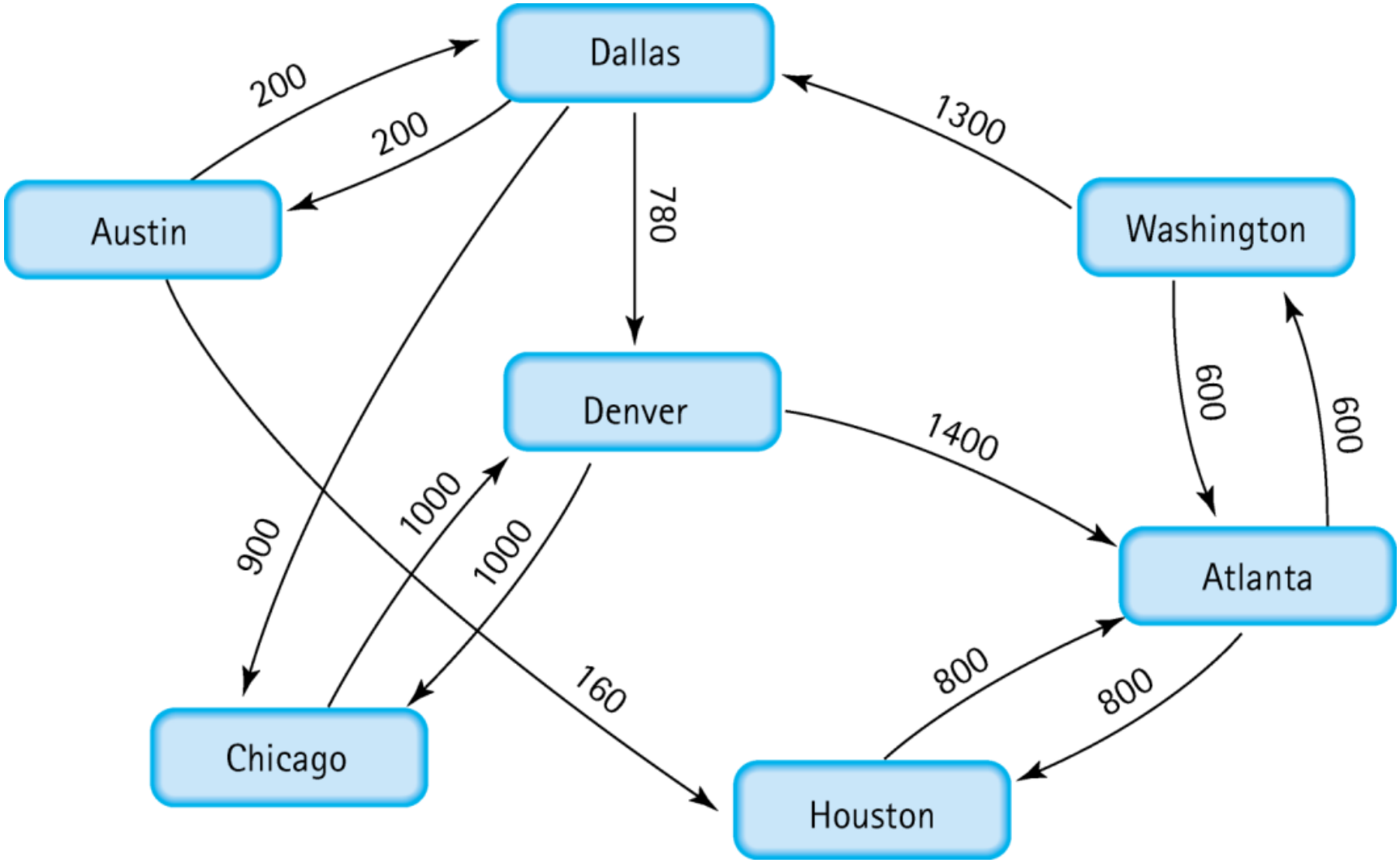


from node x ? to node x ?


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

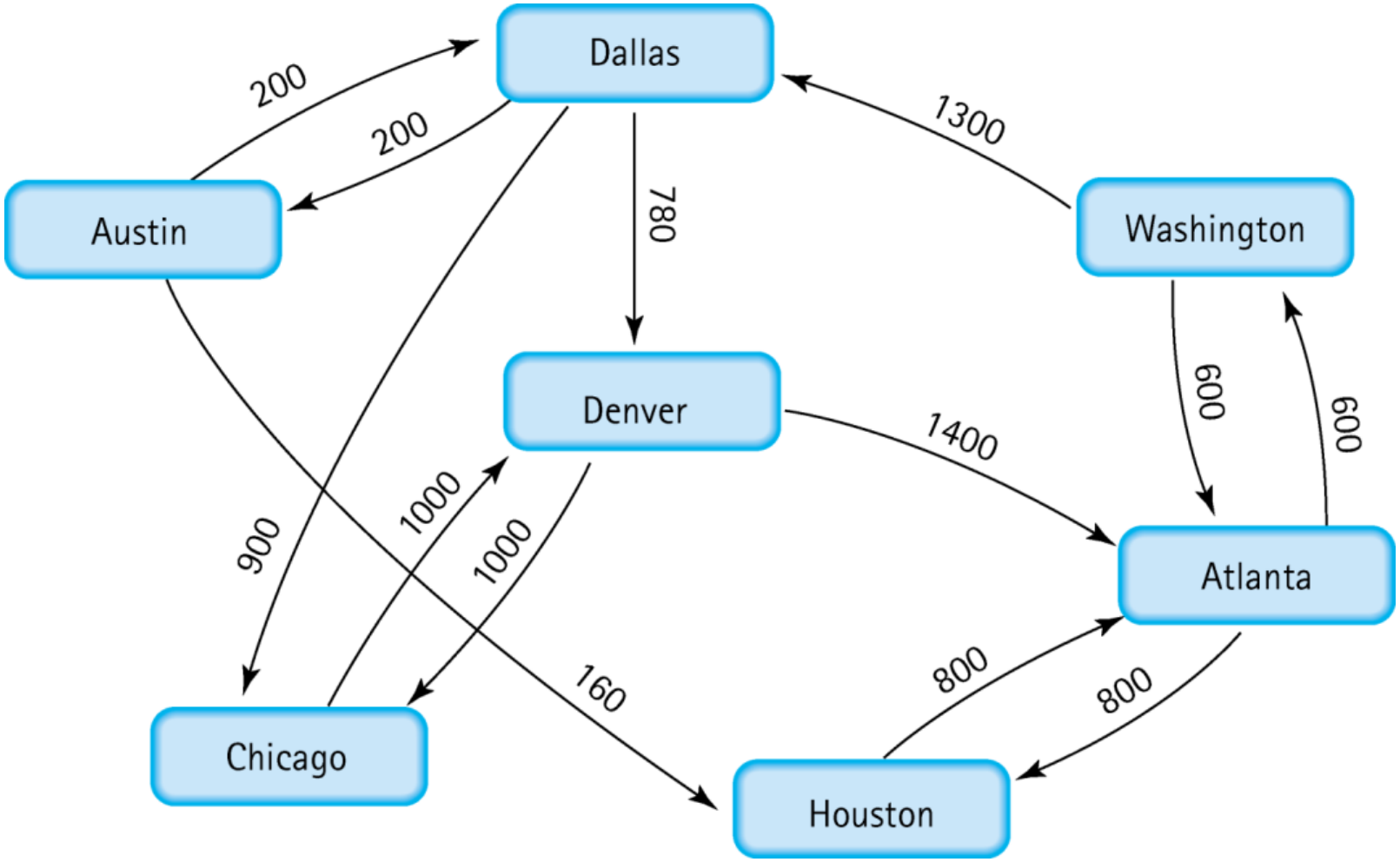
- Memory required
  - $O(V+V^2)=O(V^2)$
- Preferred when
  - The graph is **dense:** $E = O(V^2)$
- Advantage
  - Can quickly determine if there is an edge between two vertices
- Disadvantage
  - Consumes significant memory for sparse large graphs


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

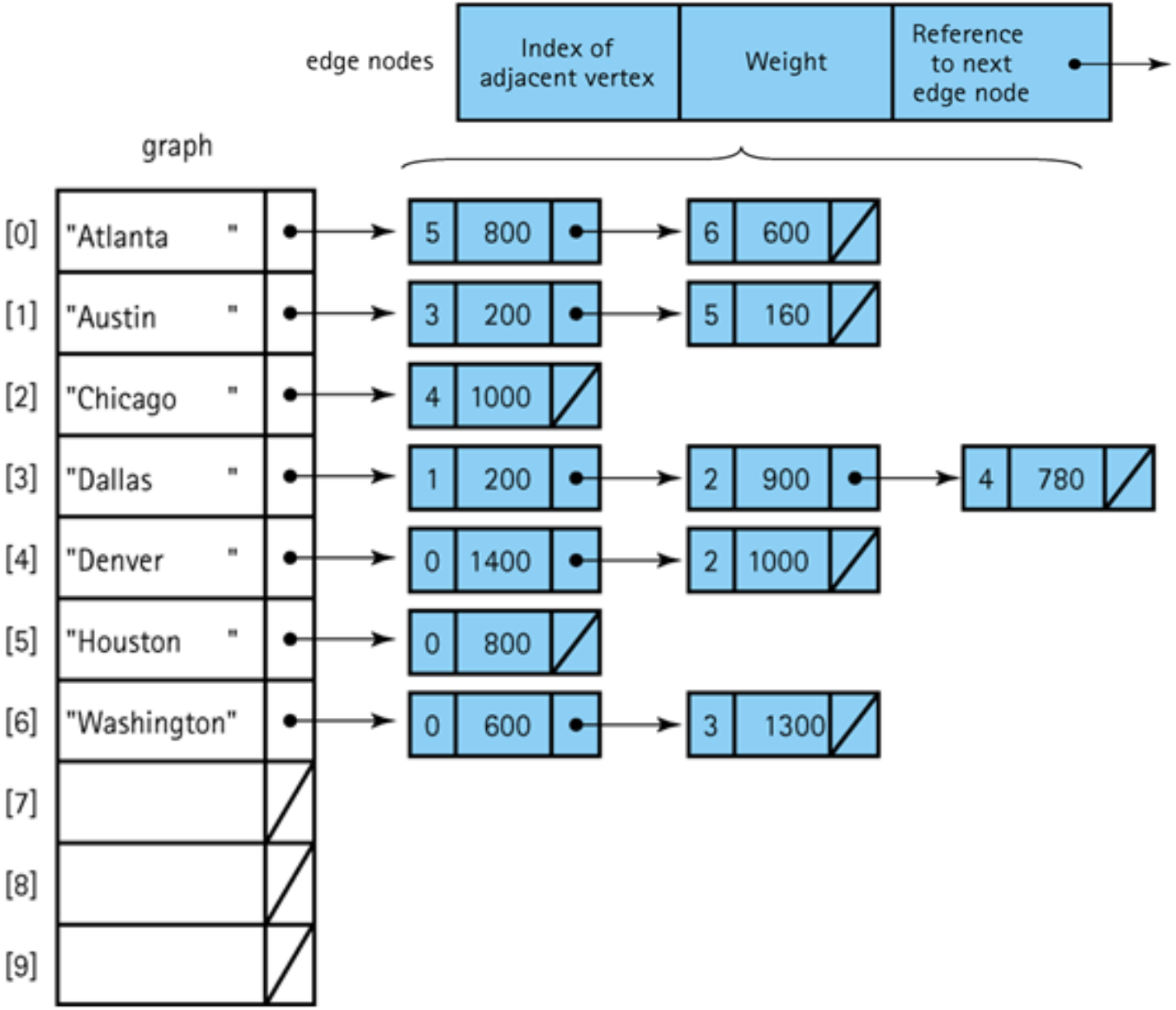
Linked Implementation

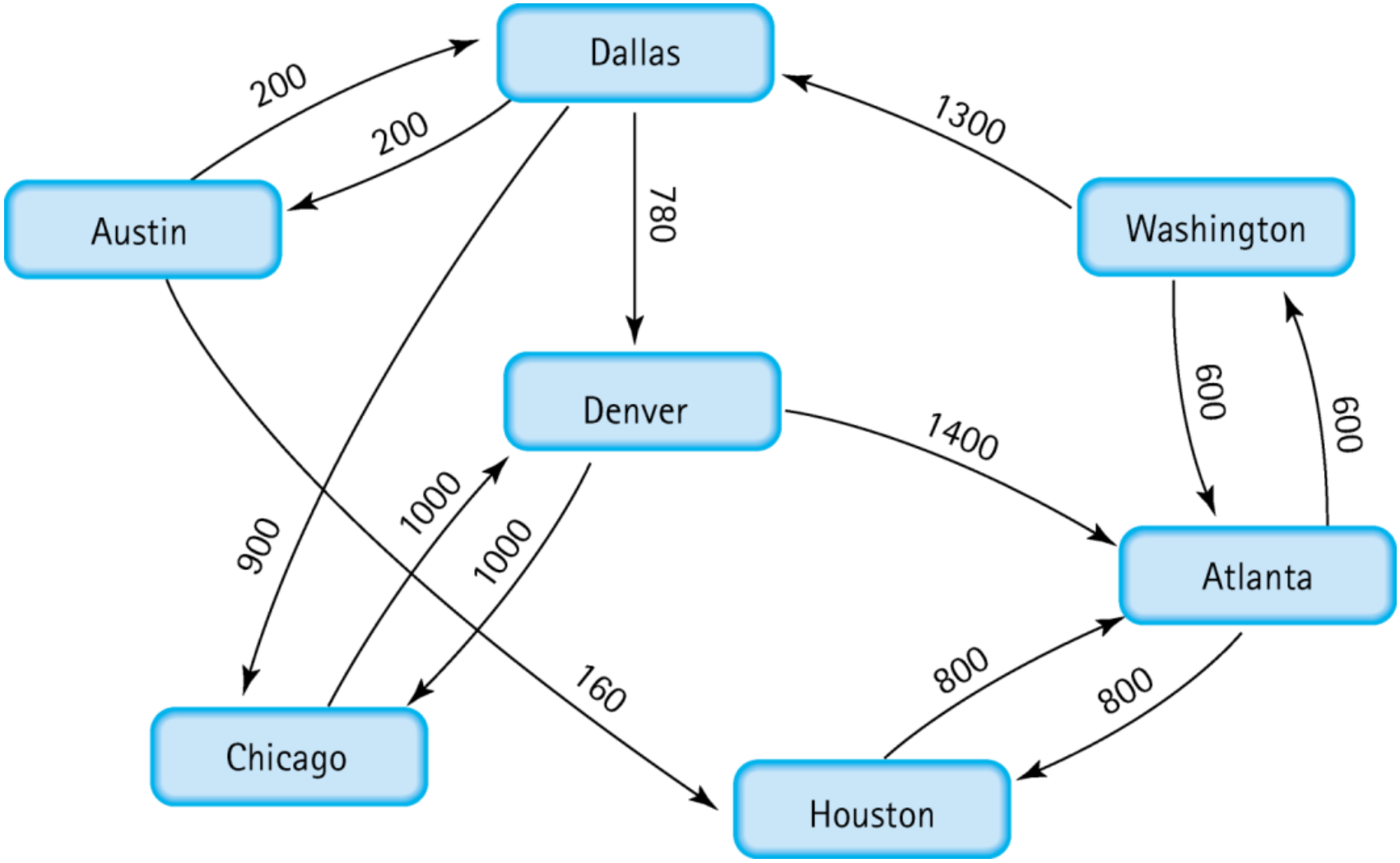
- Use a 1D array to represent the vertices
- Use a list for each vertex *v* which contains the vertices which are adjacent **from** *v* (i.e., adjacency list)
- **Adjacency List:**
  - A linked list that identifies all the vertices to which a particular vertex is connected;
    - each vertex has its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

Adjacency List Representation of Graphs



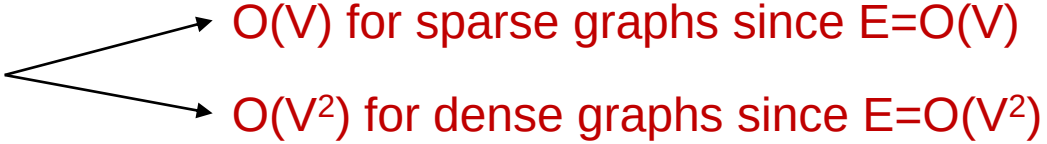
from node x ? to node x ?


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)



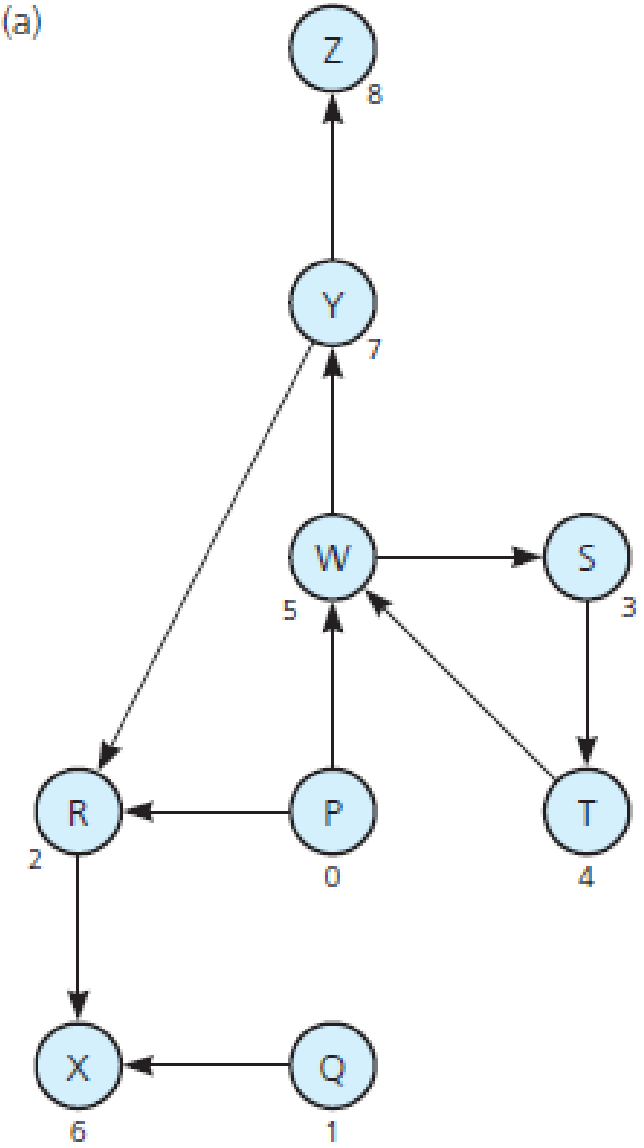
- Preferred when
  - for **sparse** graphs: E = O(V)
- Disadvantage
  - No quick way to determine the vertices adjacent to a given vertex
- Advantage
  - Can quickly determine the vertices adjacent **from** a given vertex

Implementing Graphs

(a)A directed graph and

(b)Its adjacency matrix

(a)

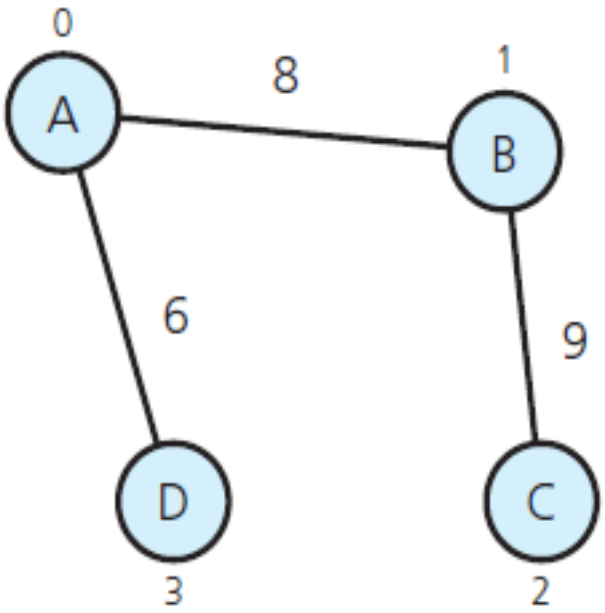


(b)

		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)



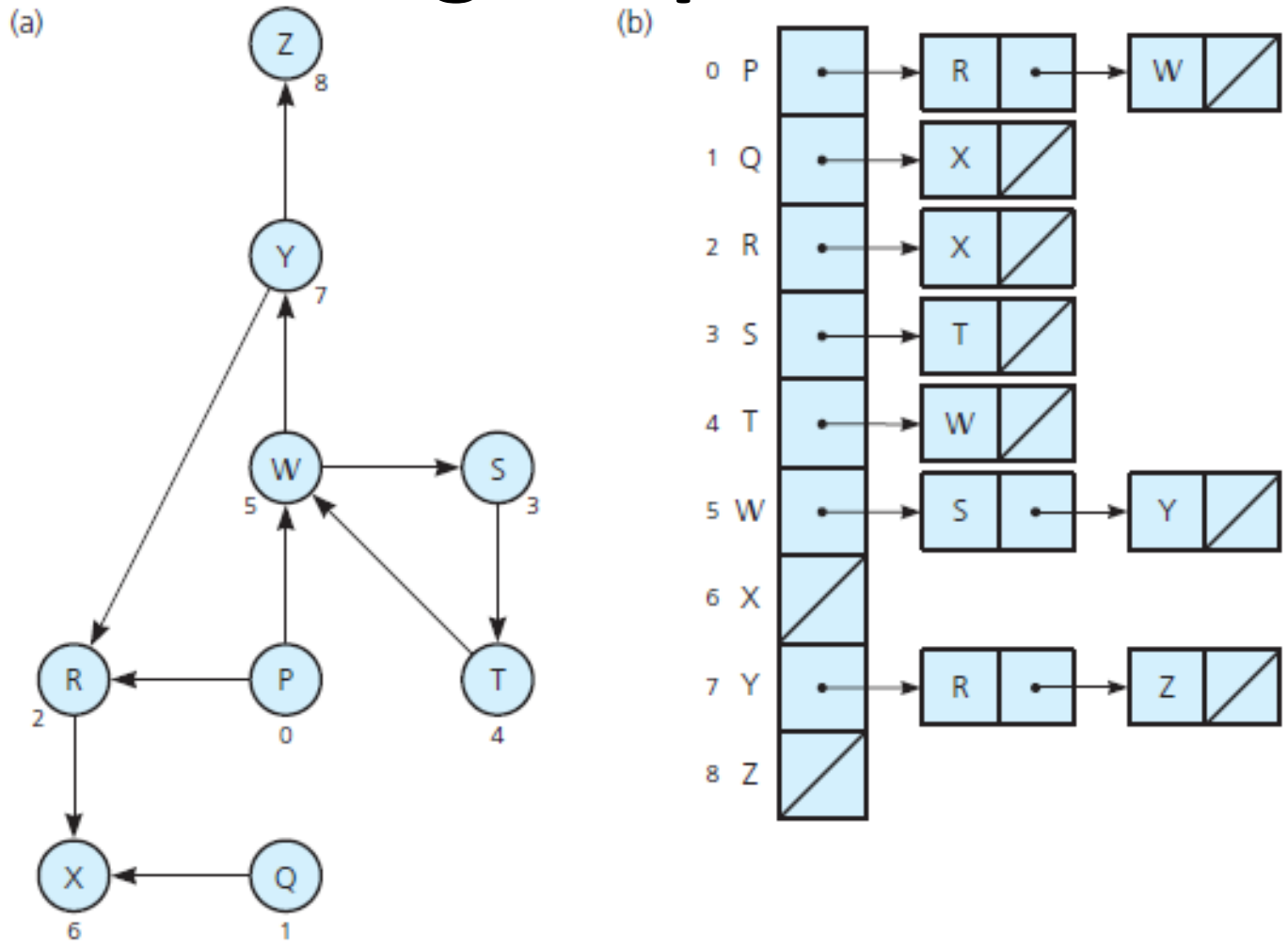
(b)

		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

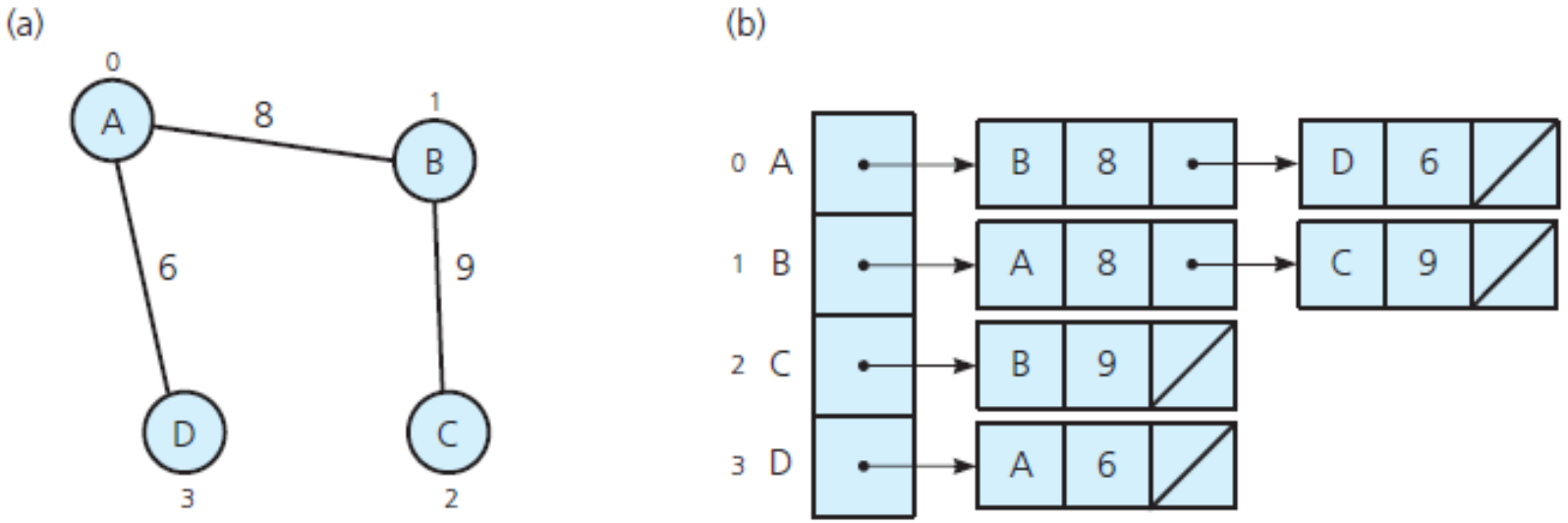
(b) Its adjacency matrix

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)

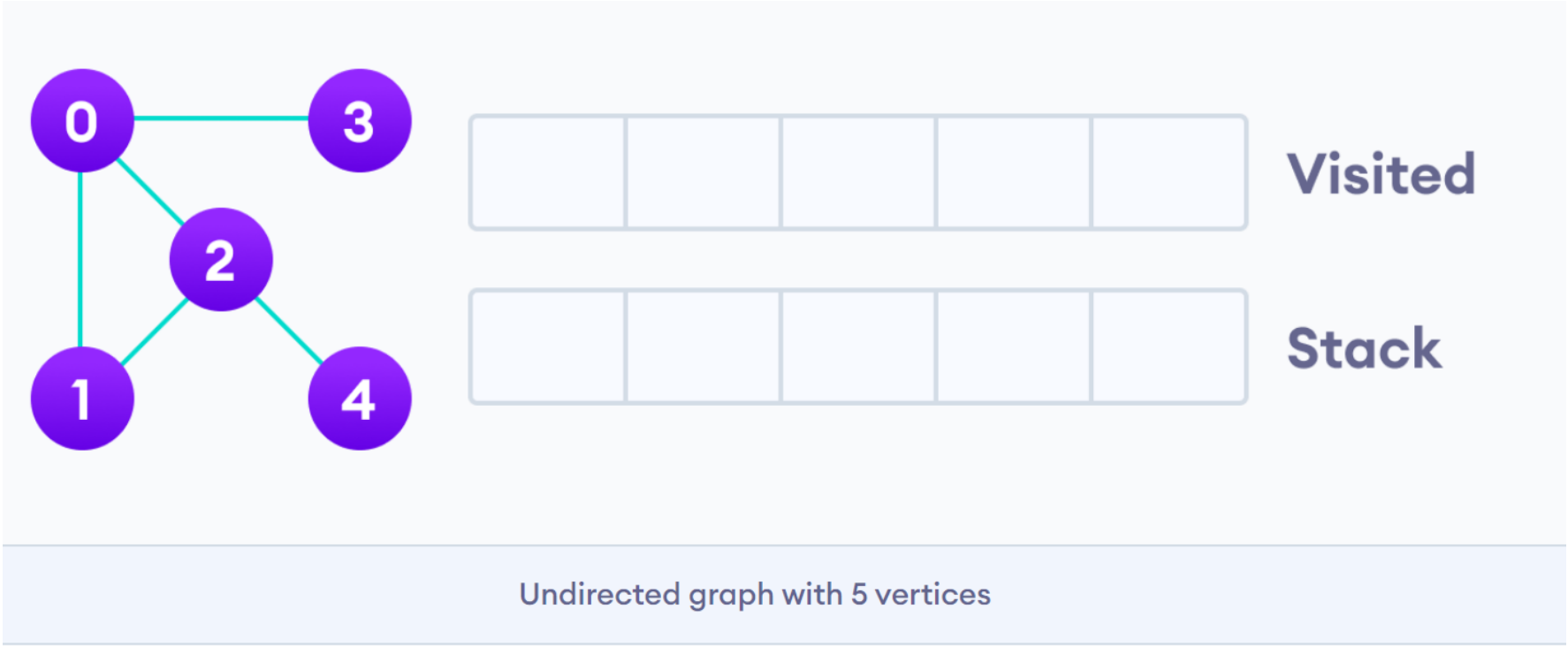
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

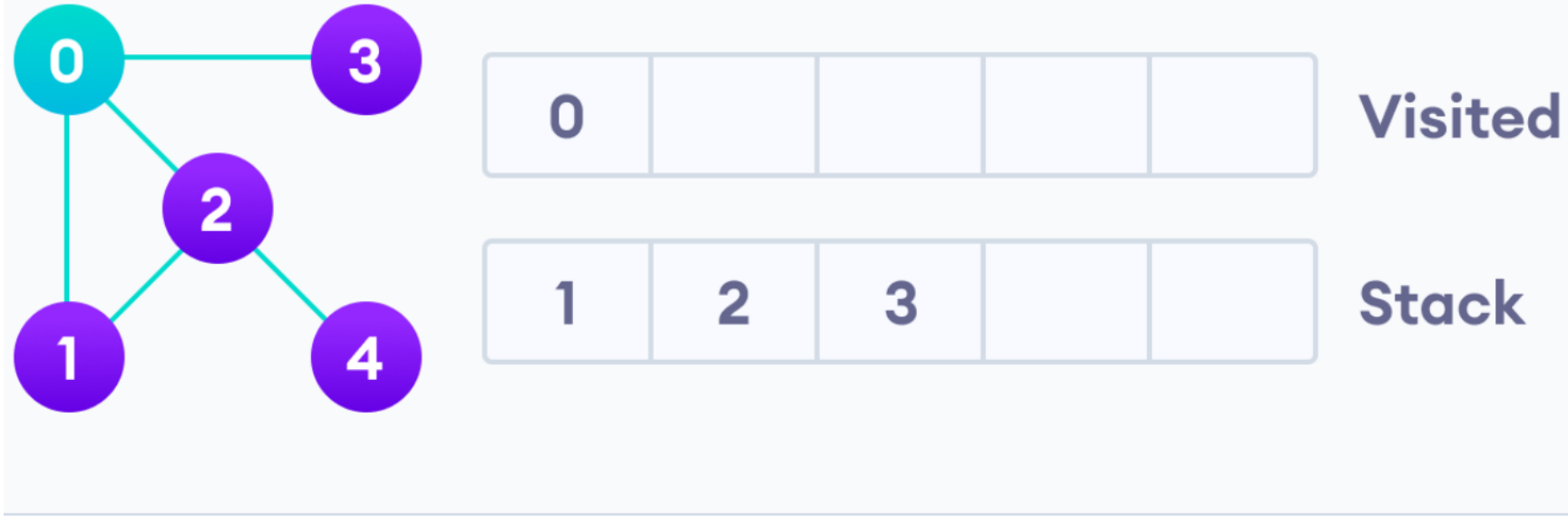
DFS example

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

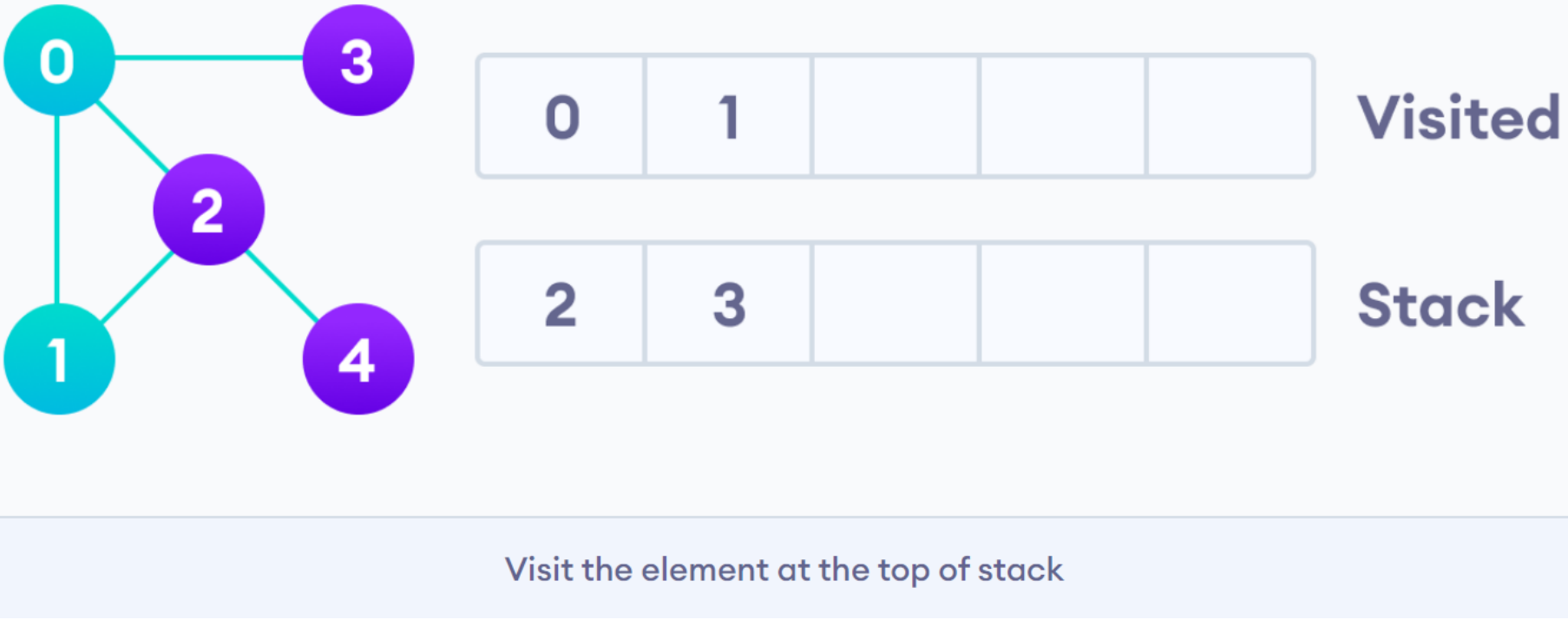


Undirected graph with 5 vertices

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.

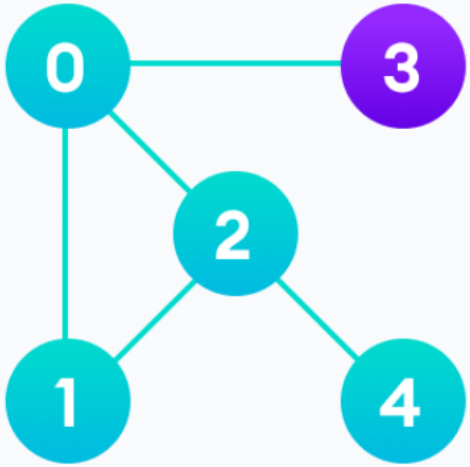


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



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.



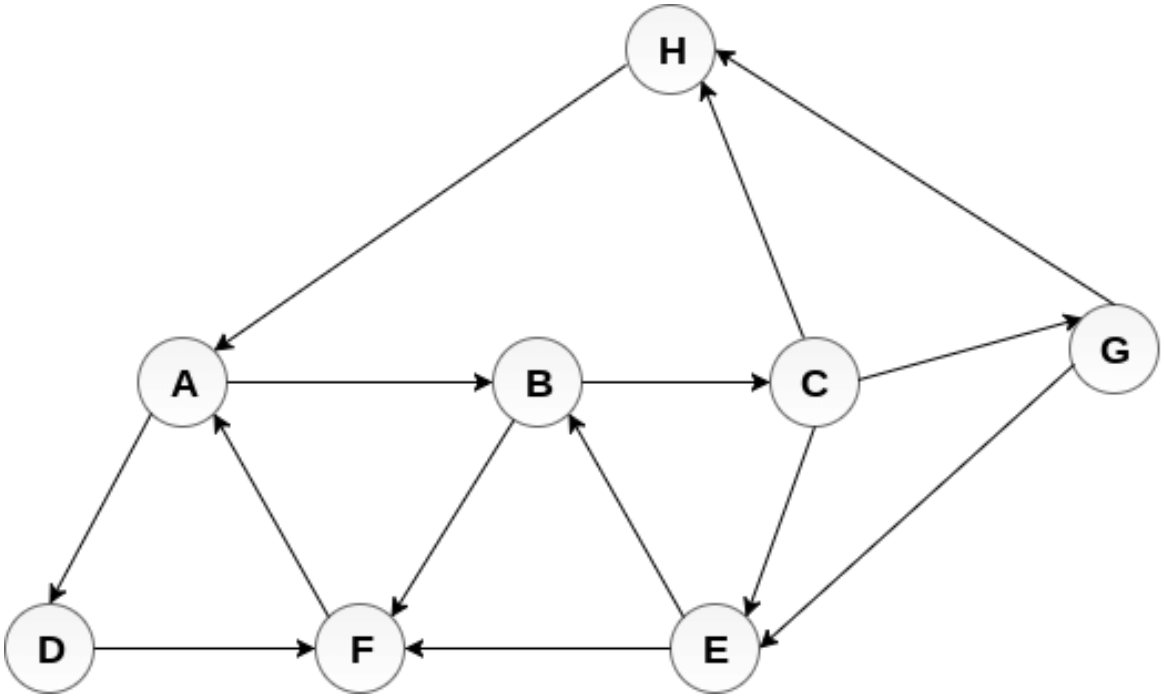
0	1	2	4		Visited
3					Stack

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

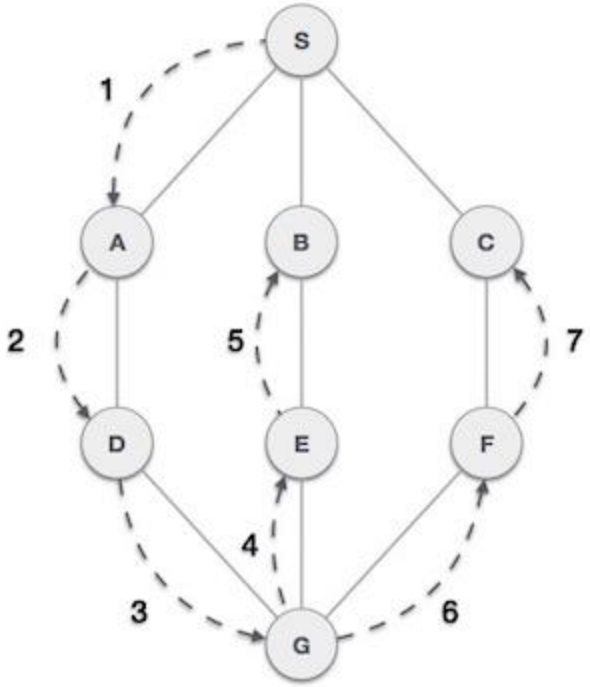
Stack :

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.

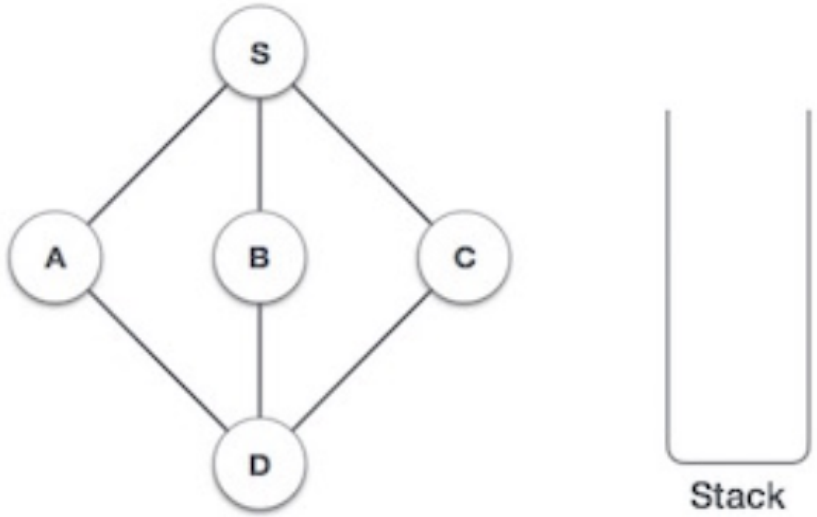
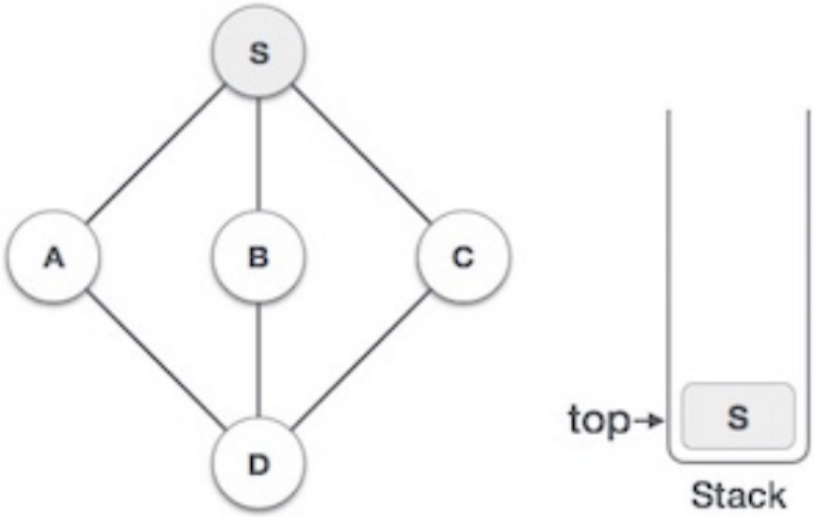


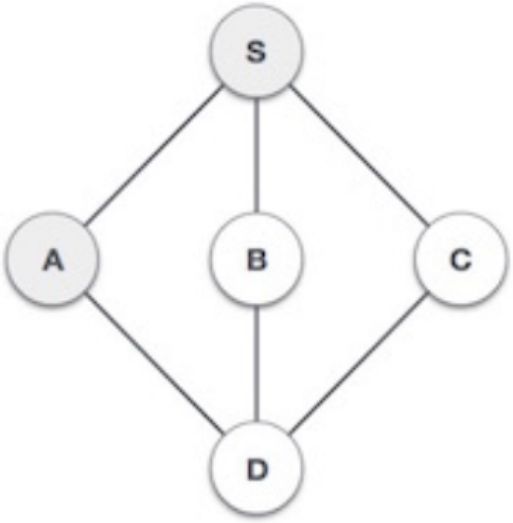
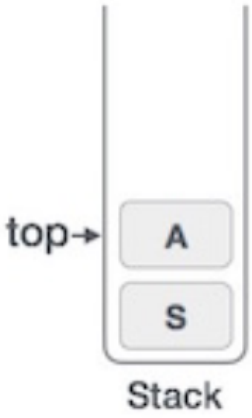
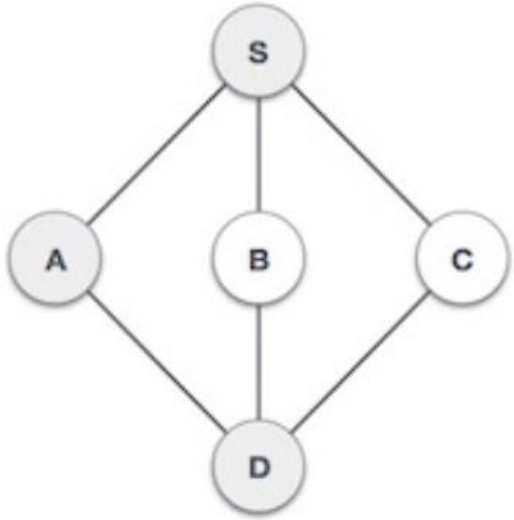
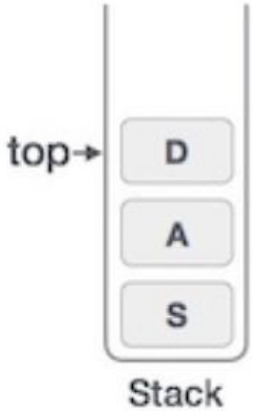
In the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

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

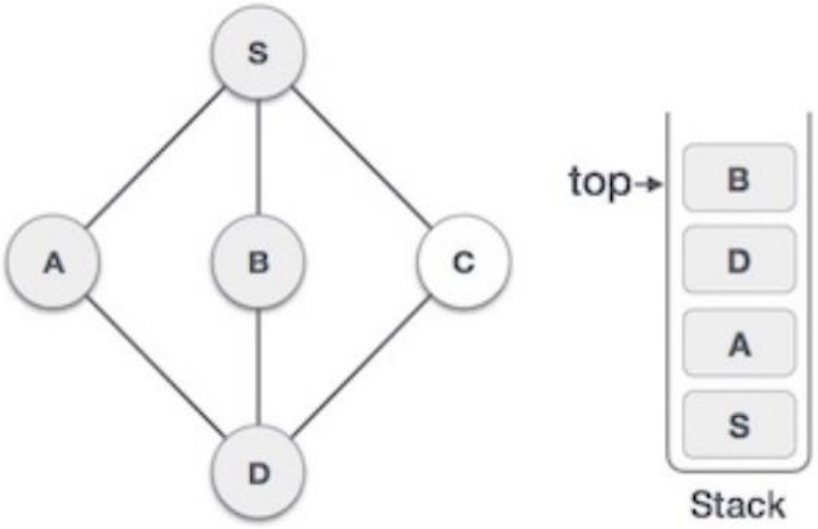
Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

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.

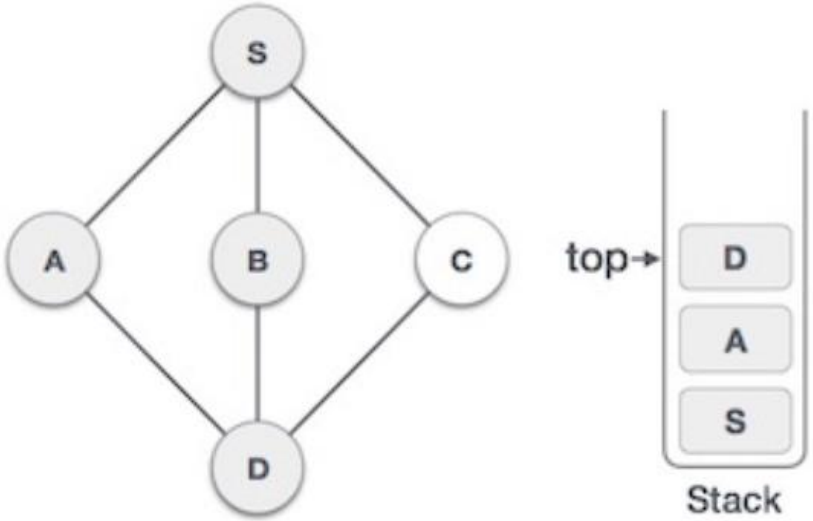
3			Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4			Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent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

5

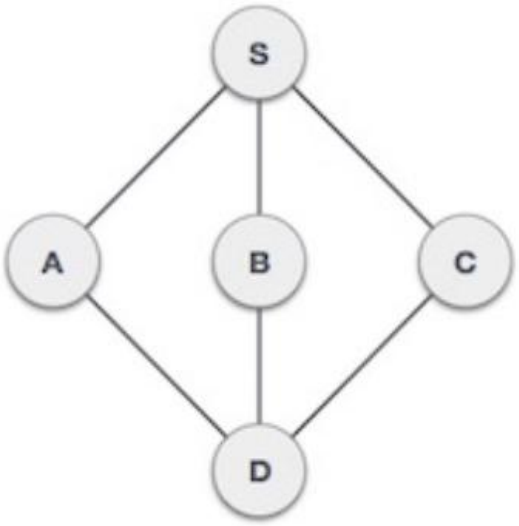
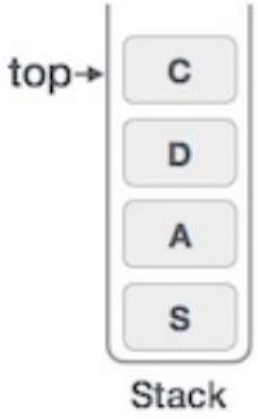


We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

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 pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
        u.visited = true
        for each v ∈ G.Adj[u]
                if v.visited == false
                        DFS(G,v)


init()
 {
        For each u ∈ G
                u.visited = false
        For each u ∈ G
                DFS(G, u)
}
```

DFS Algorithm Complexity

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

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

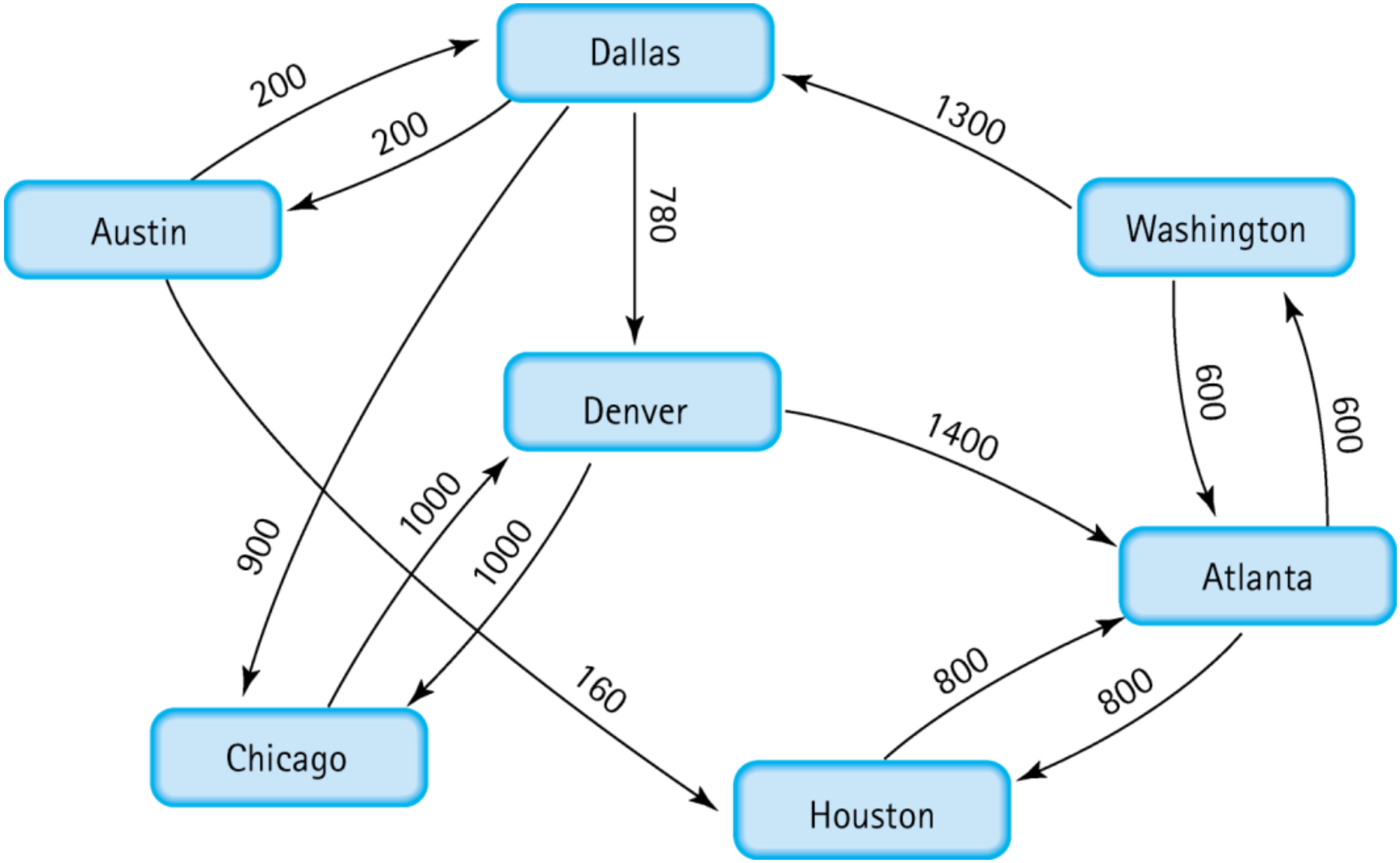
Depth-First-Search (DFS)

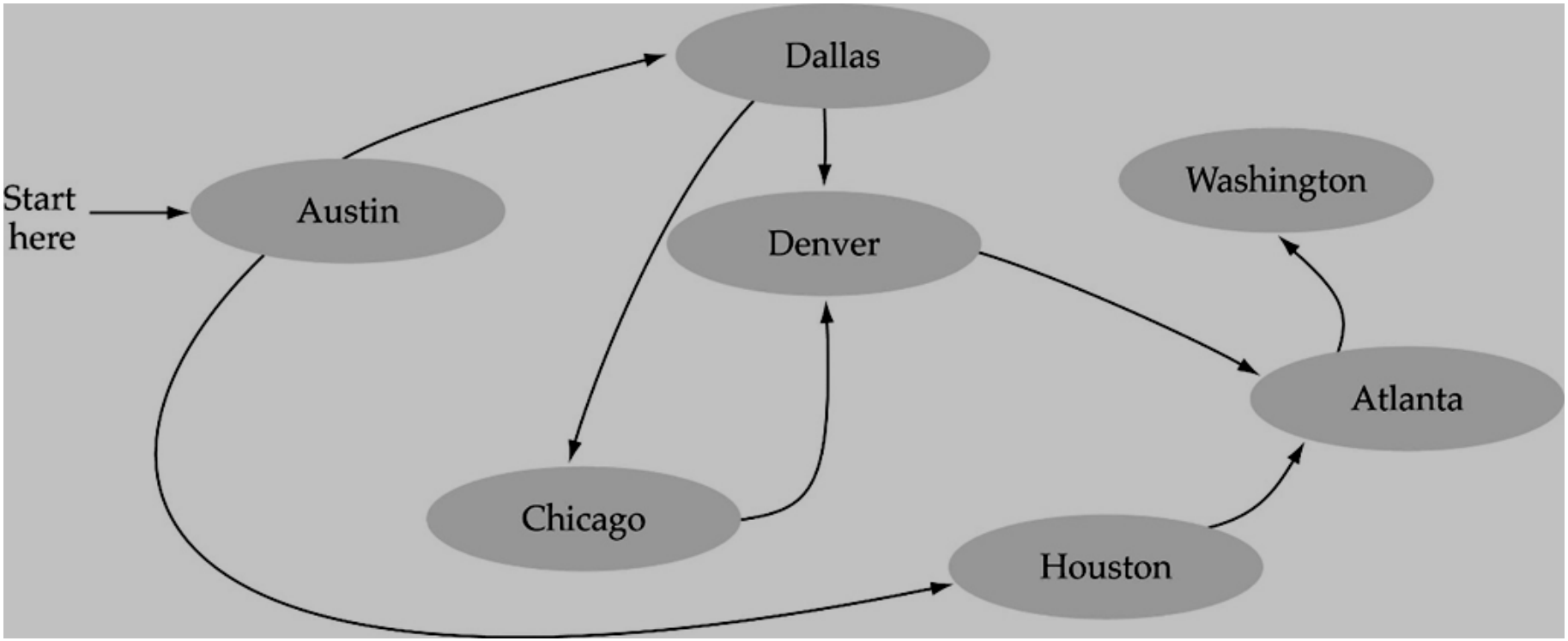
Main idea:

Travel as far as you can down a path

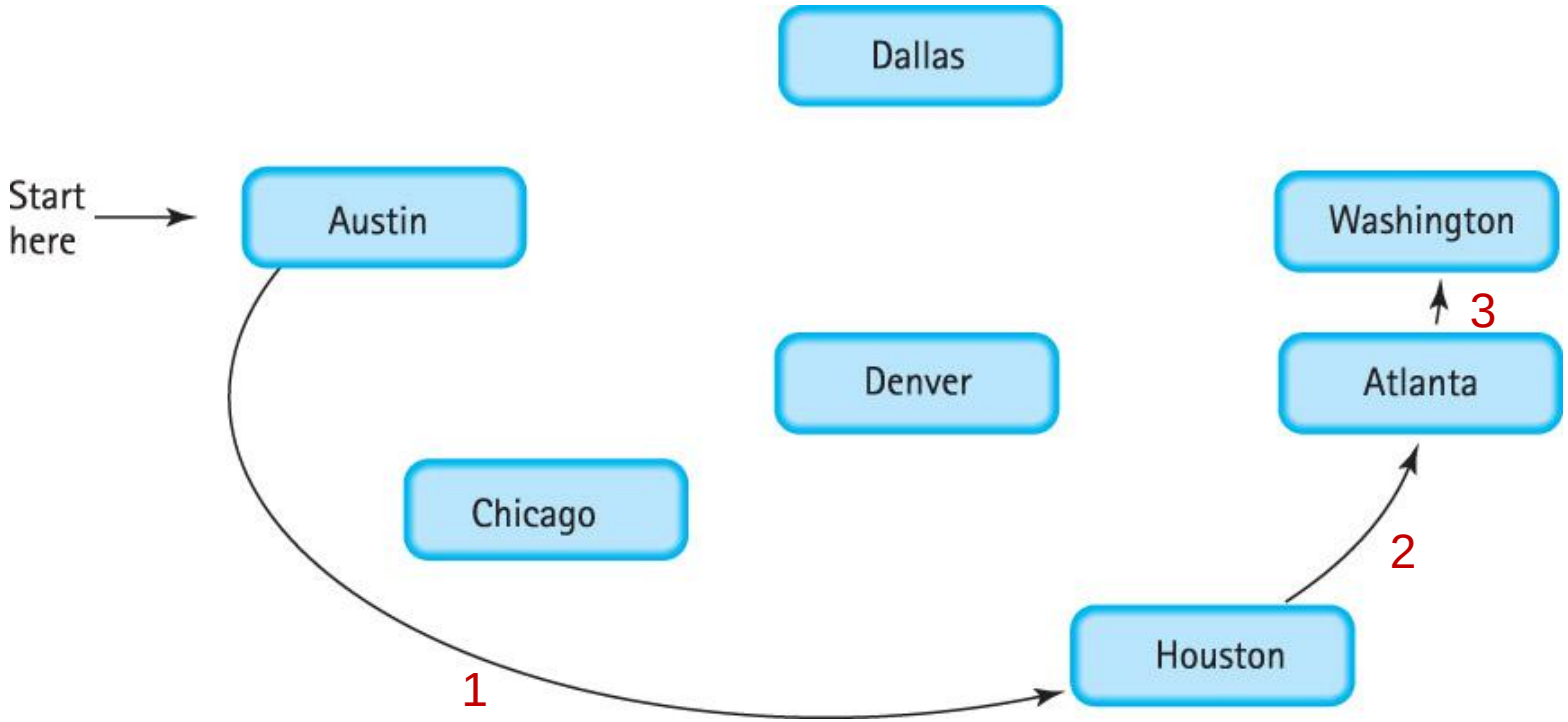
Back up **as little as possible** when you reach a "dead end“

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
Washington
Denver
Atlanta
Chicago
Houston

Depth First Search: Follow Down



DFS uses Stack !

graph

.numVertices 7

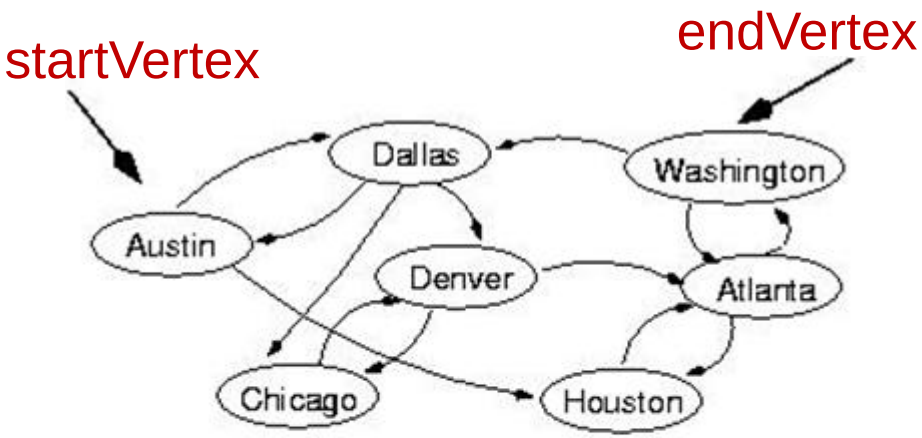
.vertices

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

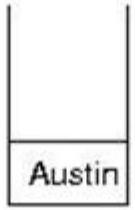
.edges

	[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]	•	•	•	•	•	•	•	•	•	•

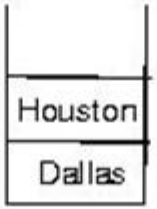
(Array positions marked '•' are undefined)

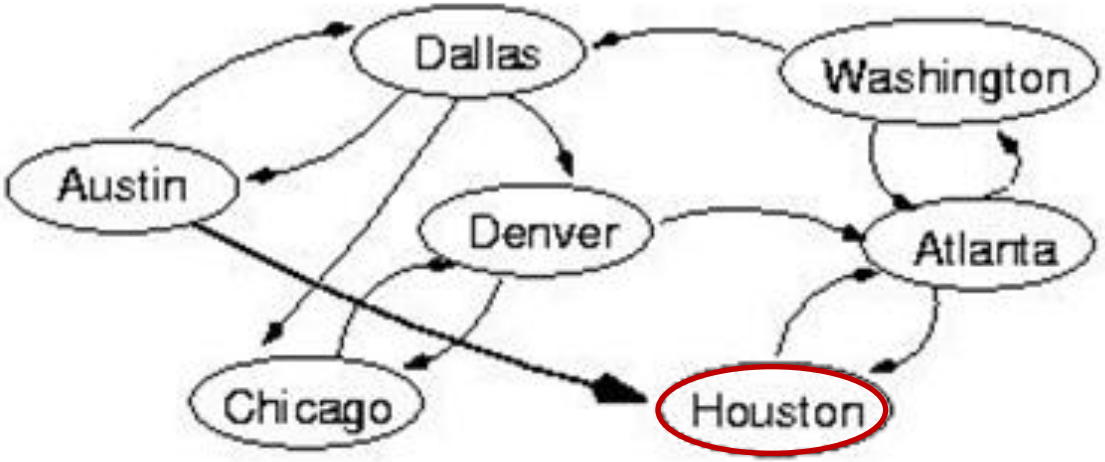


(initialization)

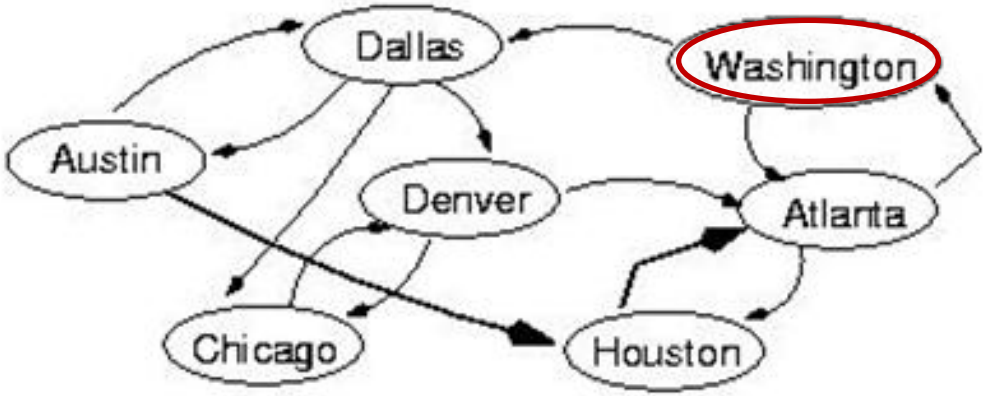


pop Austin



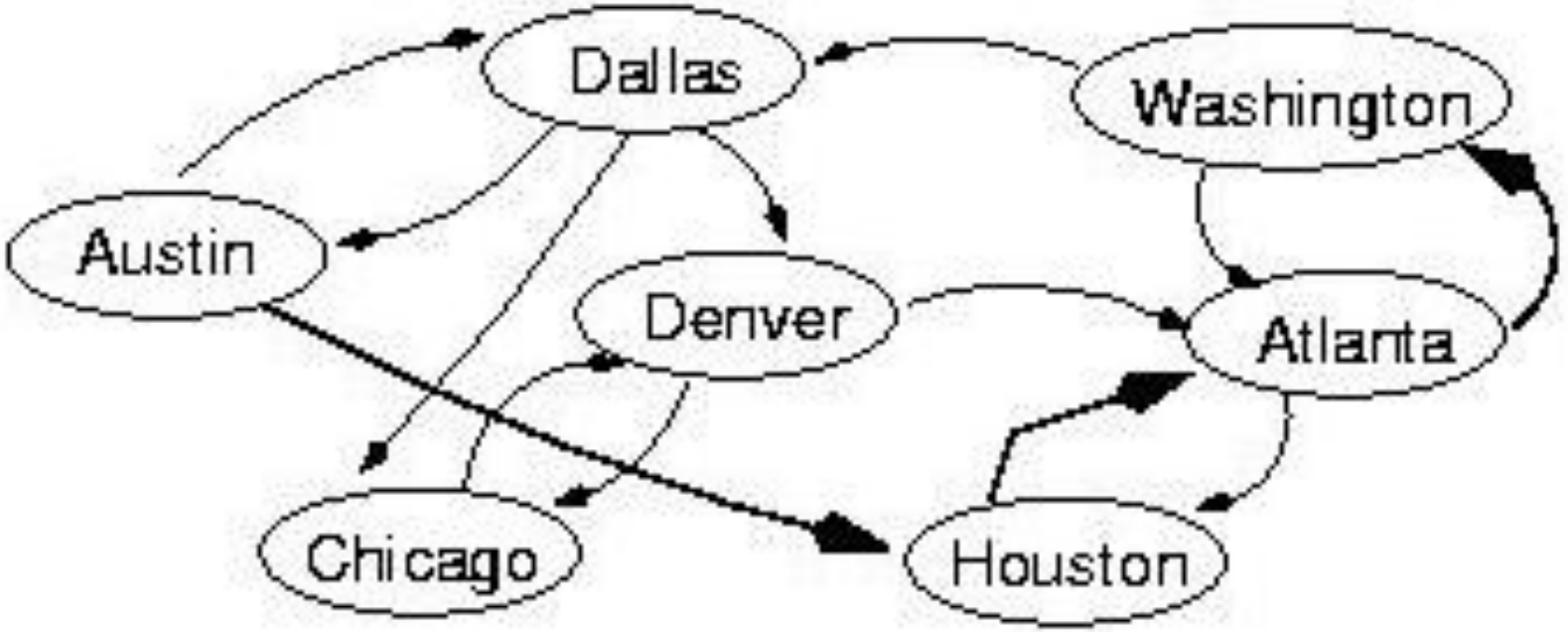


pop Houston

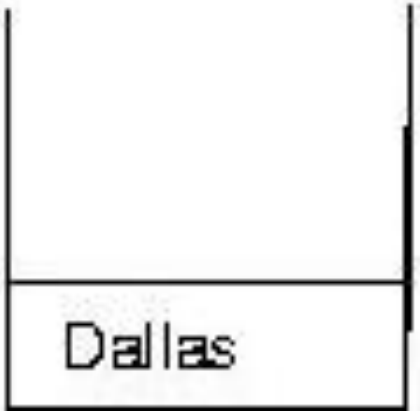


pop Atlanta

endVertex



pop Washington

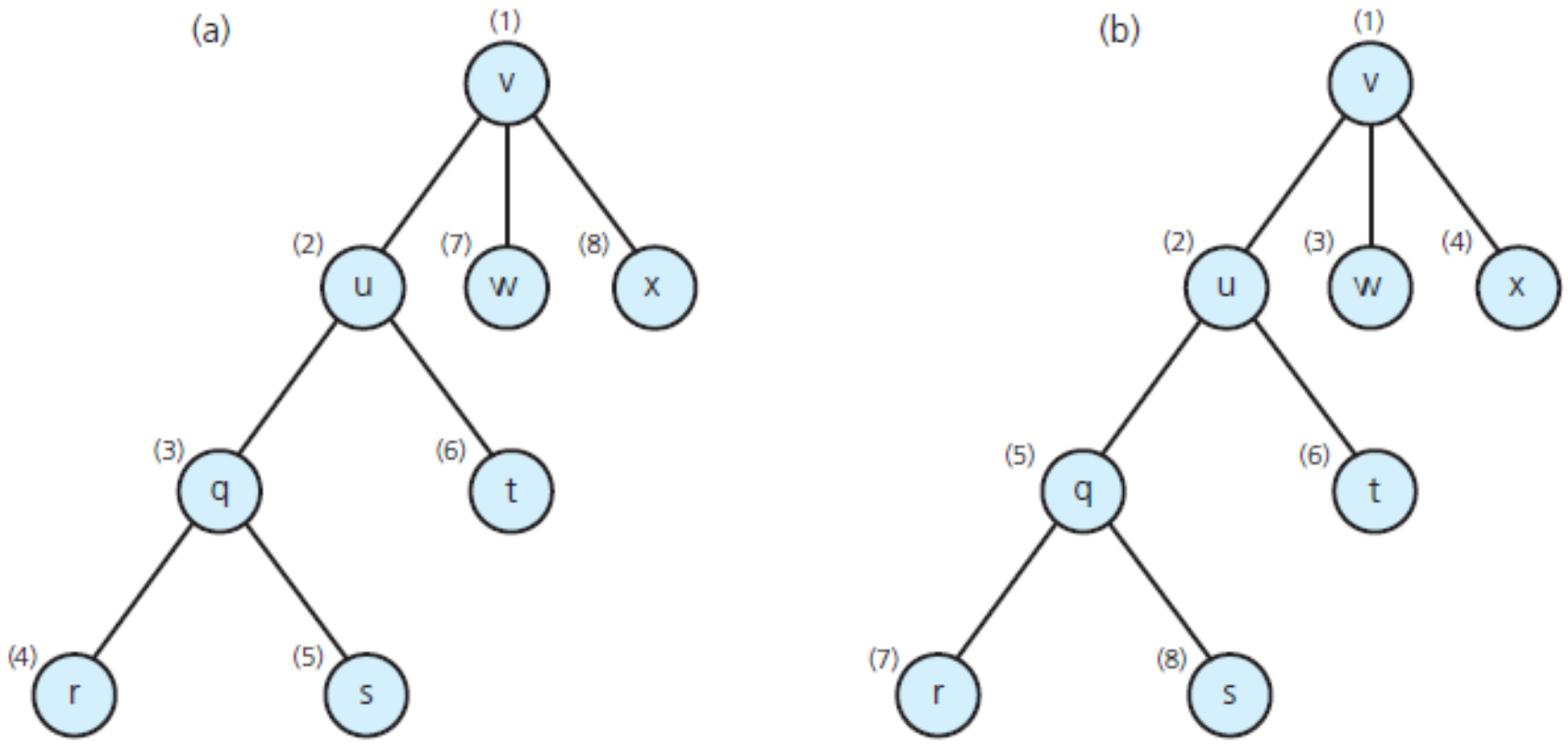


Depth-First Search

Visitation order for

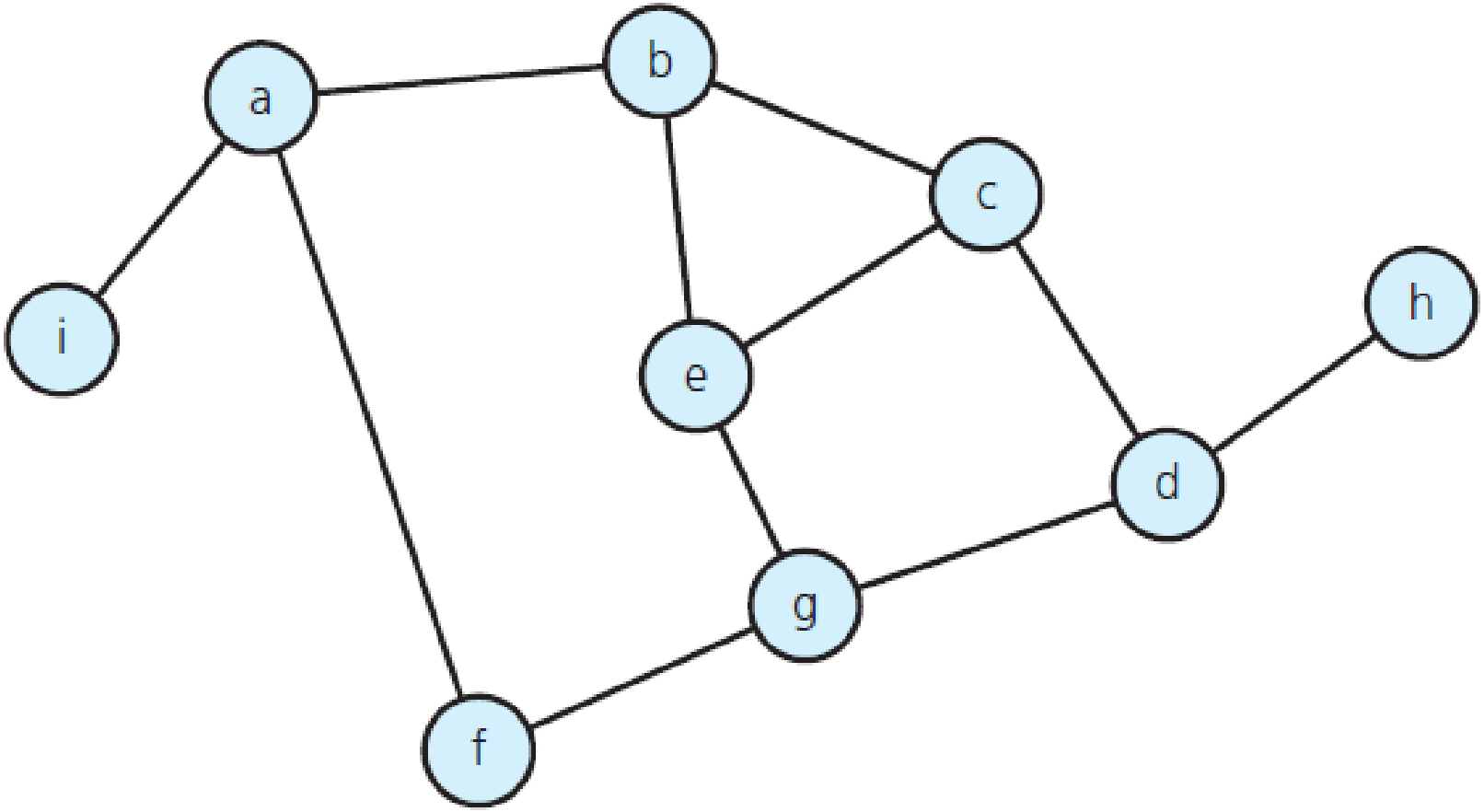
(a) a depth-first search;

(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:

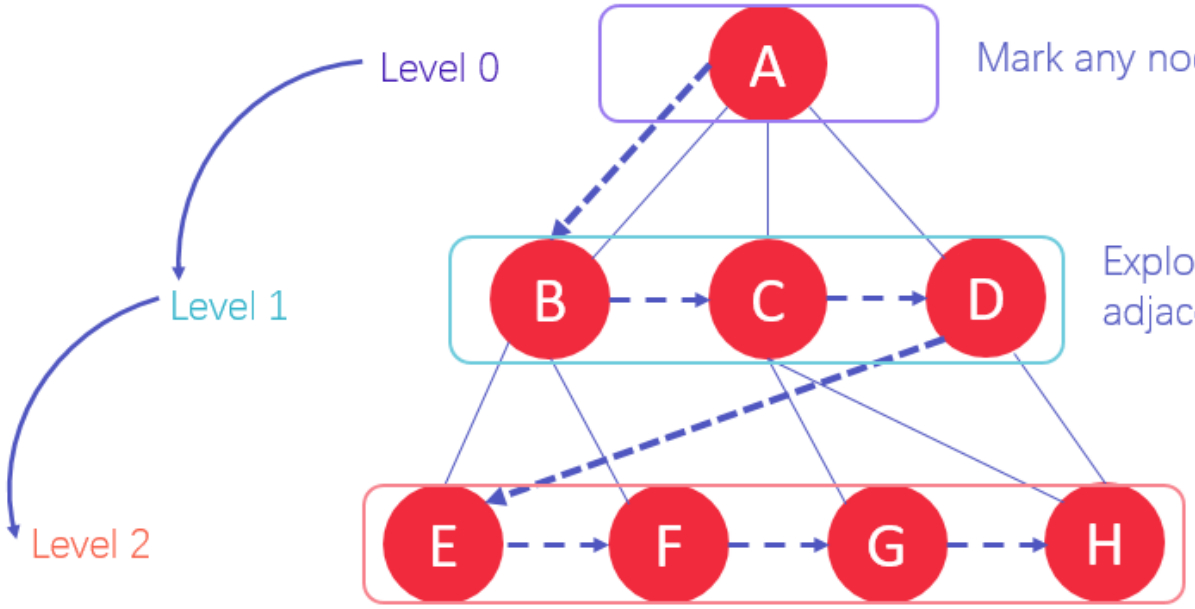
- Visited
- Not Visited

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.

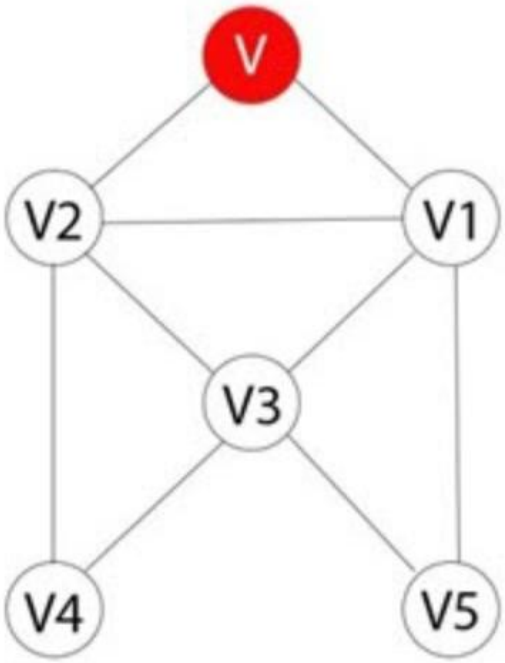
How does BFS Algorithm Work?

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.

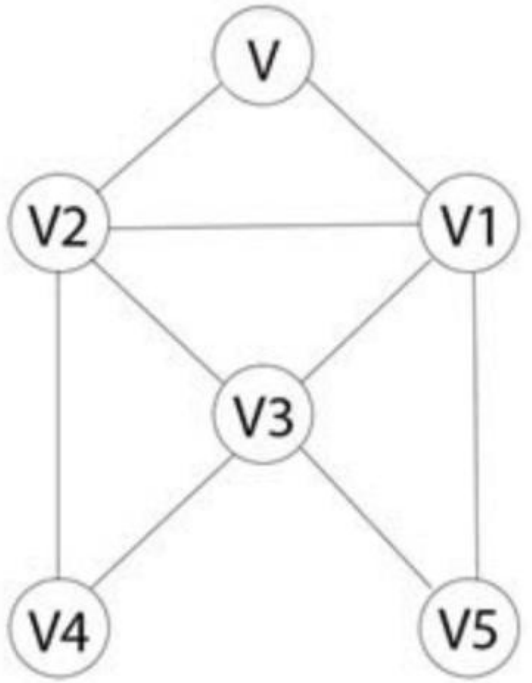
Hence, you can say that all the nodes adjacent to the current vertex are visited and traversed in the first iteration. A simple queue methodology is utilized to implement the working of a BFS algorithm, and it consists of the following steps:

Step 1)



Each vertex or node in the graph is known. For instance, you can mark the node as V.

Step 2)

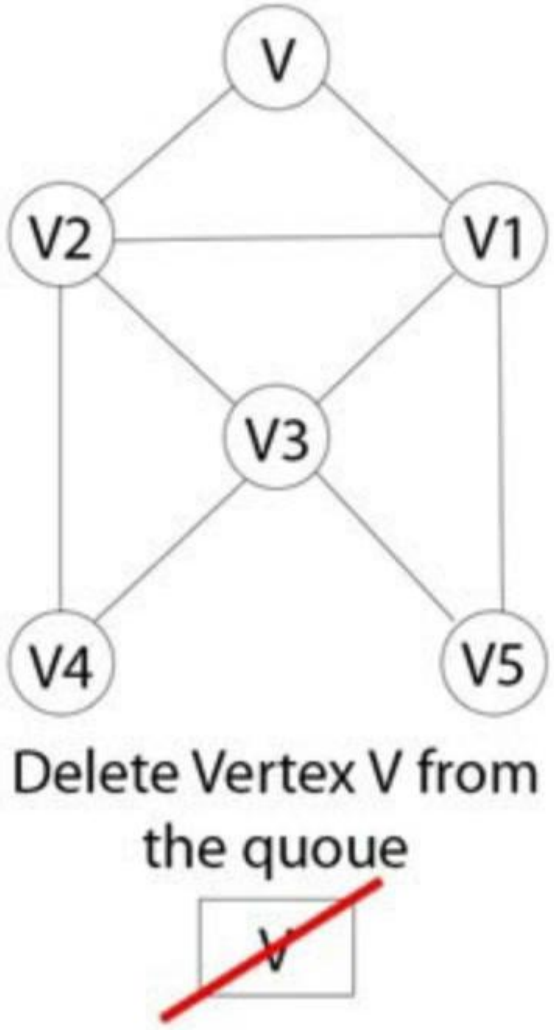


In case V is not visited
add it to BFS queue

Queue = | V | |

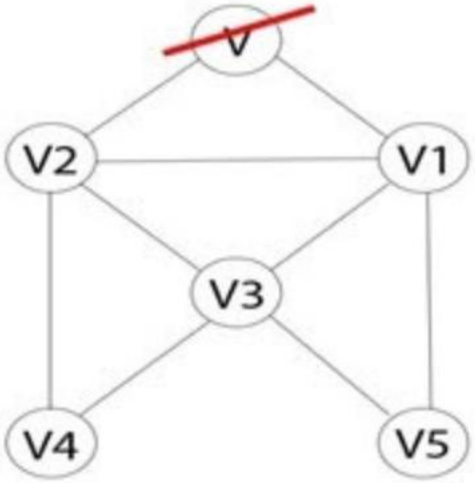
In case the vertex V is not accessed then add the vertex V into the BFS Queue

Step 3)



Start the BFS search, and after completion, Mark vertex V as visited.

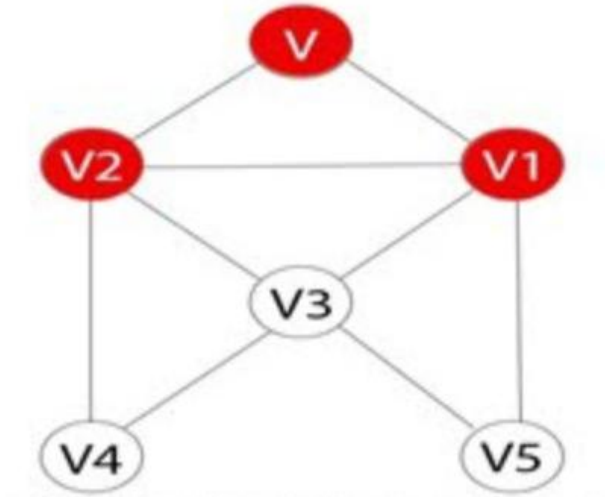
Step 4)



1. Start the BFS search
2. After completion, mark V as completed

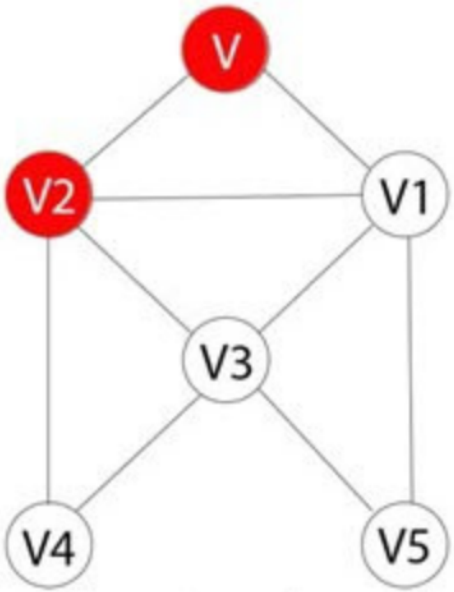
The BFS queue is still not empty, hence remove the vertex V of the graph from the queue.

Step 5)



Retrieve all the remaining vertices on the graph that are adjacent to the vertex V

Step 6)

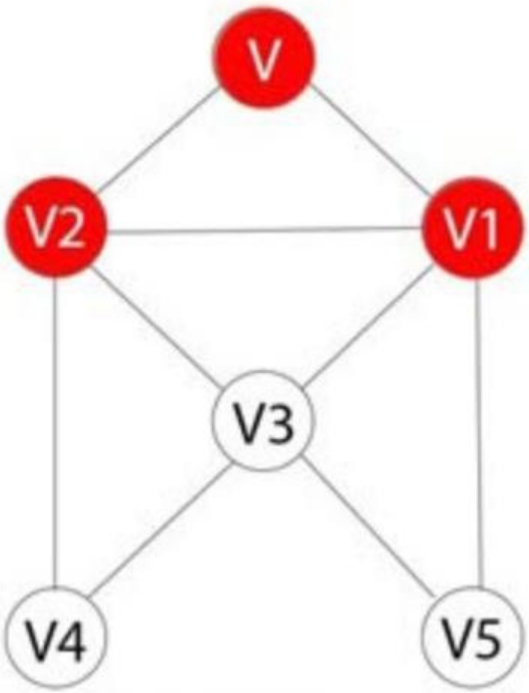


For adjacent and un-visited vertex say V1, add it to BFS queue

Queue = ~~V~~ V1

For each adjacent vertex let's say V1, in case it is not visited yet then add V1 to the BFS queue

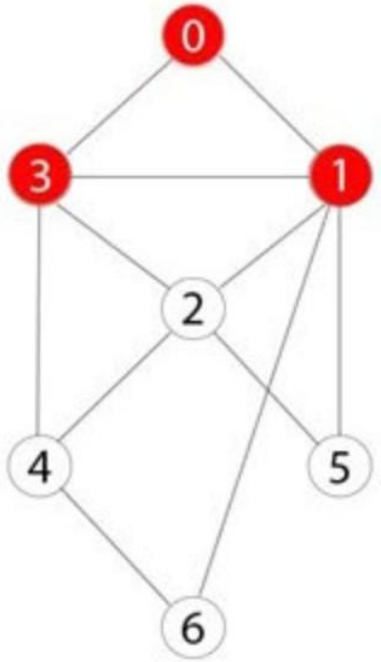
Step 7)



BFS will visit V1 and mark it as visited and delete it from the queue.

Example BFS Algorithm

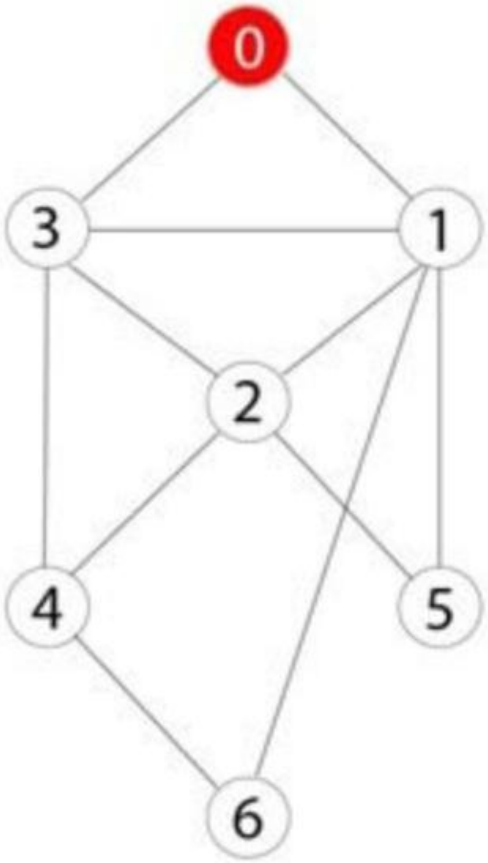
Step 1)



1. Mark 0 as visited
2. Insert 0 to the queue
3. Traverse the un-visited adjacent nodes which are 3 and 1

You have a graph of seven numbers ranging from 0 – 6.

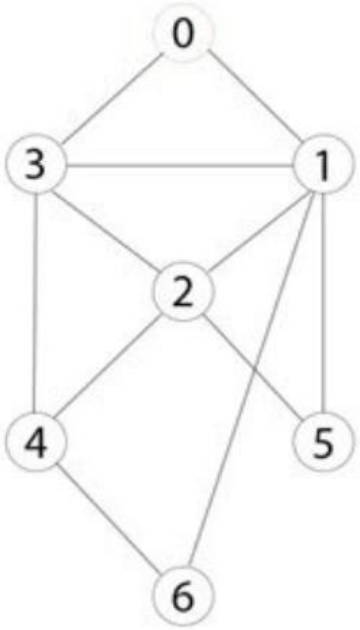
Step 2)



Root node = 0

0 or zero has been marked as a root node.

Step 3)



Queue:

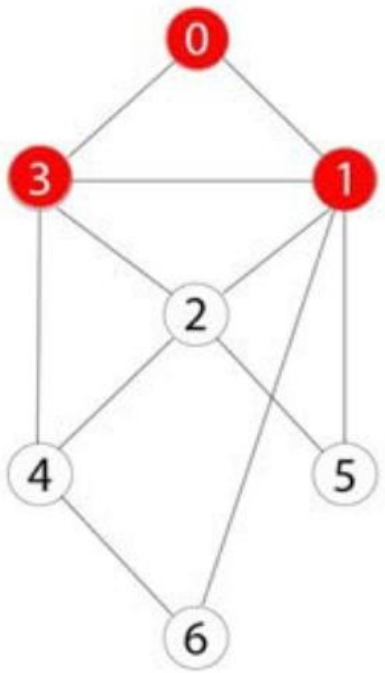
0	1	2	3	4	5	6
~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~6~~

Delete values from queue and Print as result

Result = 0, 1, 2, 3, 4, 5, 6

0 is visited, marked, and inserted into the queue data structure.

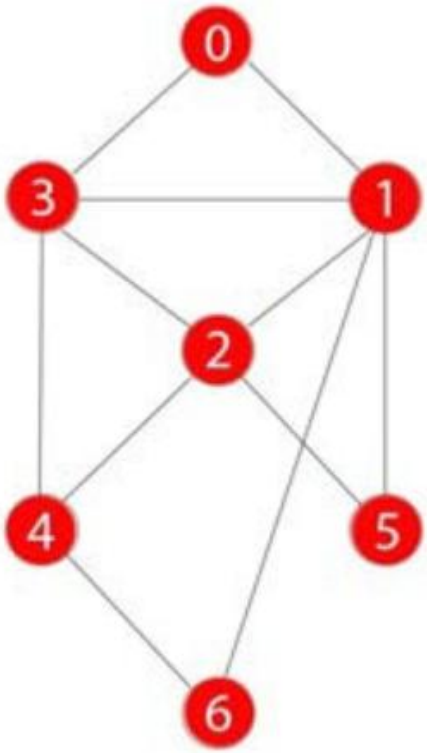
Step 4)



Visit 3 and 1 in any sequence and mark them as visited and add them to the queue

Remaining 0 adjacent and unvisited nodes are visited, marked, and inserted into the queue.

Step 5)



Visit all adjacent and un-visited nodes of the previous node and iterate until all visited

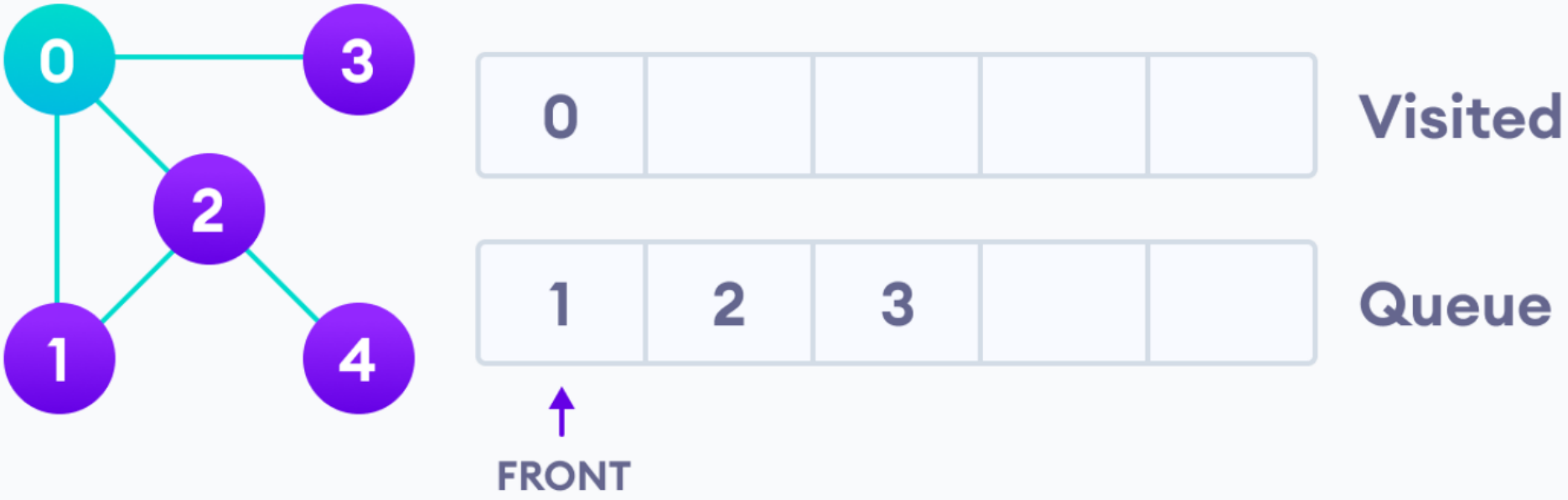
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

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

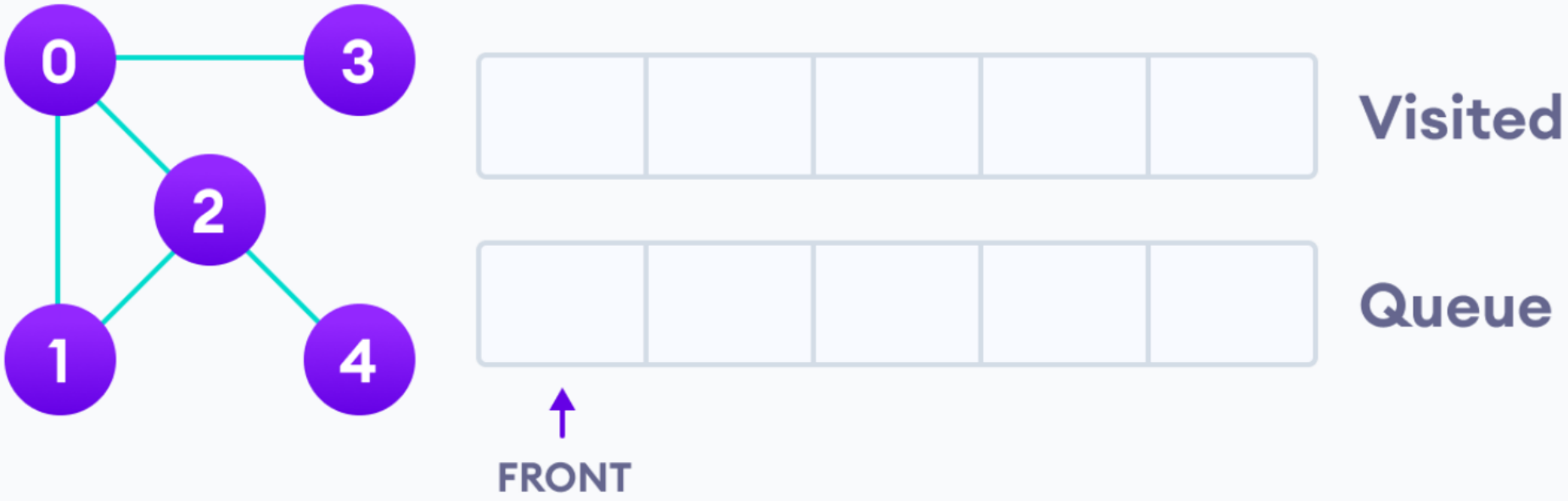
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Visit start vertex and add its adjacent vertices to queue

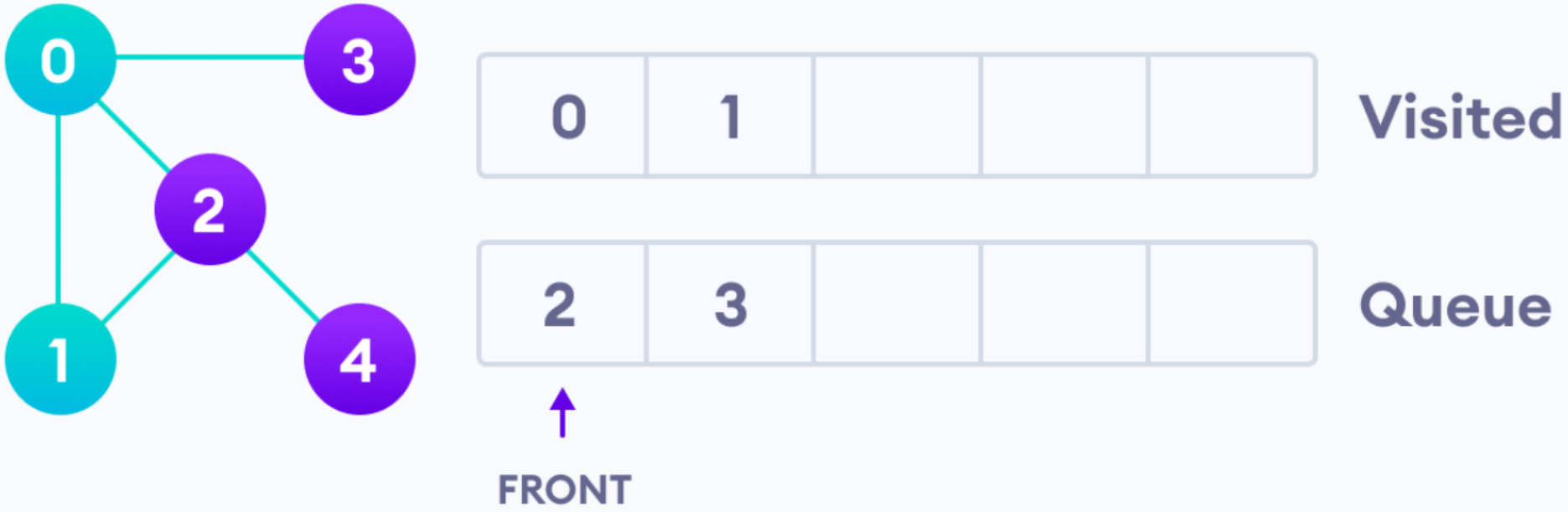
BFS example

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



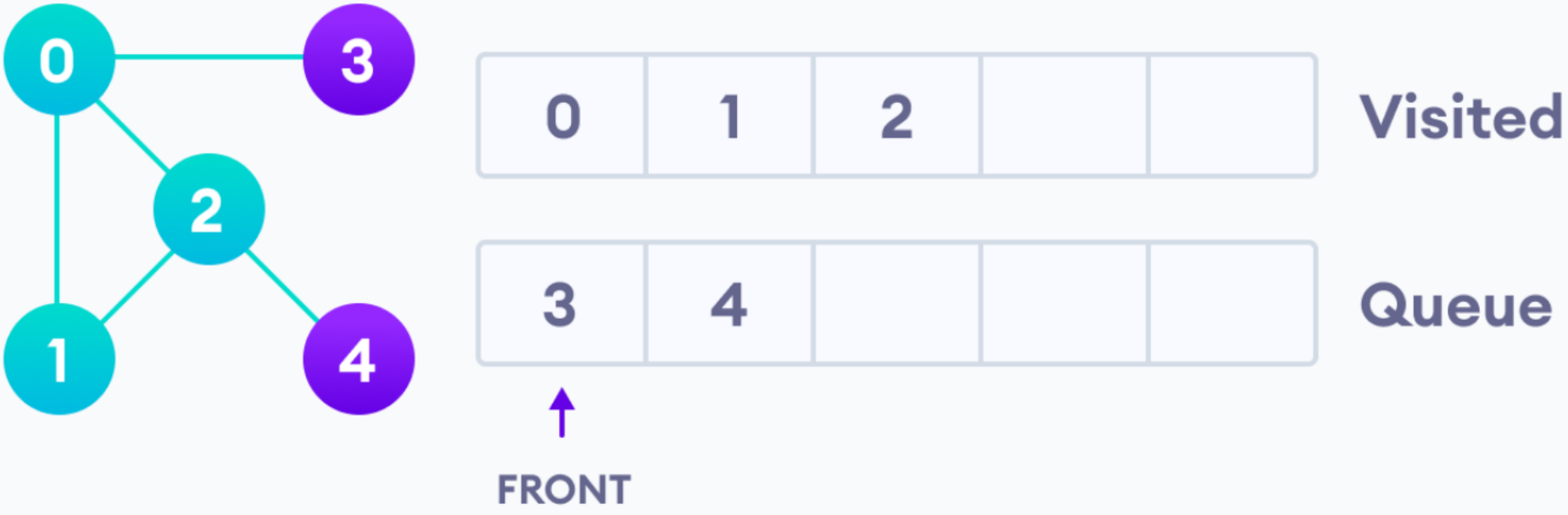
Undirected graph with 5 vertices

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.

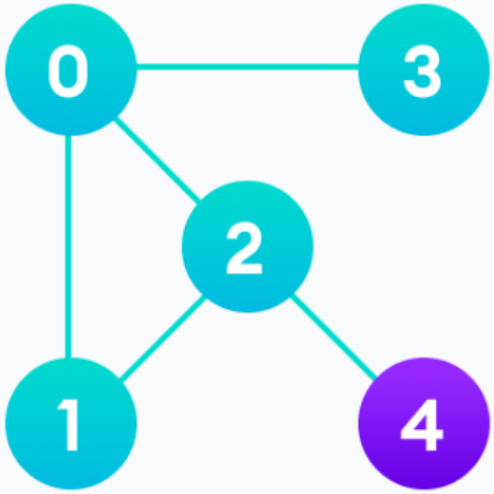


Visit the first neighbour of start node 0, which is 1

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.

BFS Pseudocode

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

    remove the head u of Q

    mark and enqueue all (unvisited) neighbours of u

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

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

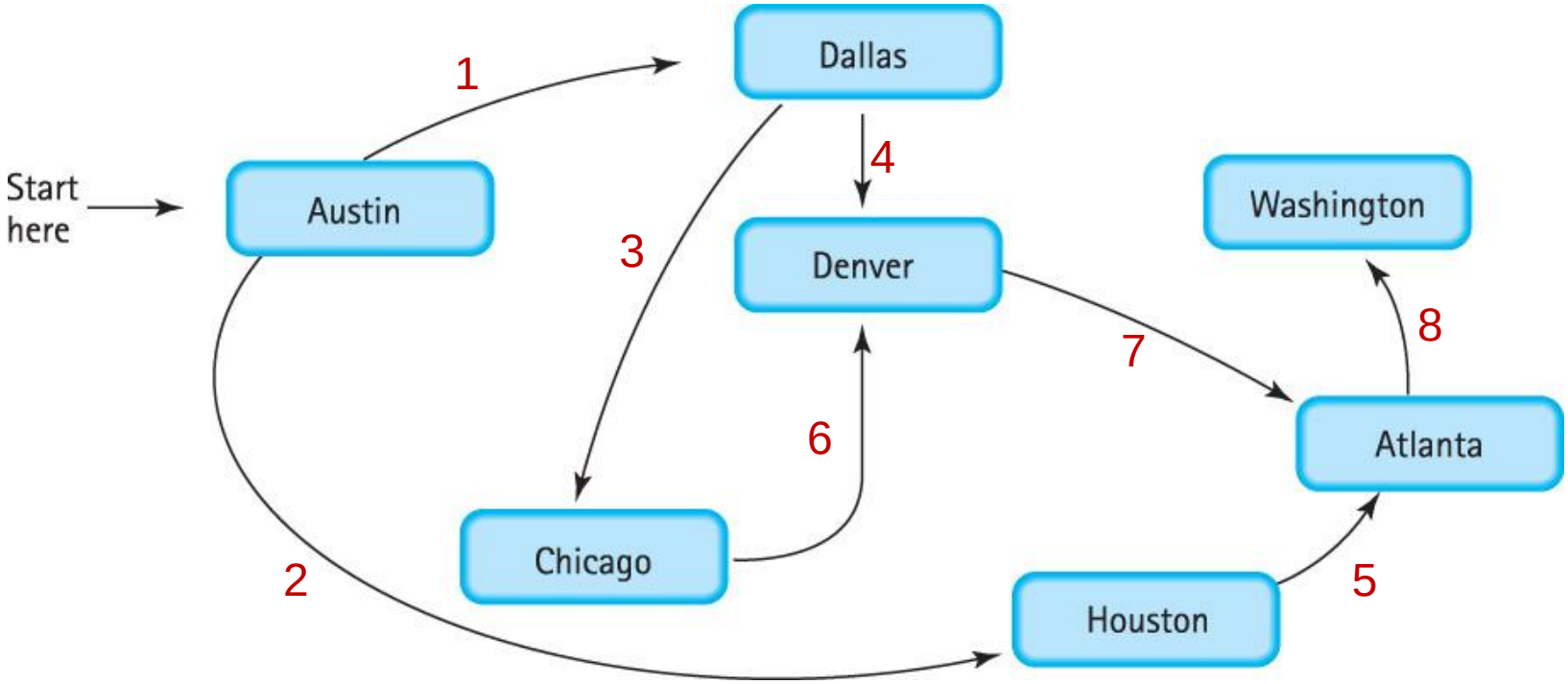
Let's take a look at some of the real-life applications where a BFS algorithm implementation can be highly effective.

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.

Breadth-First-Searching (BFS)

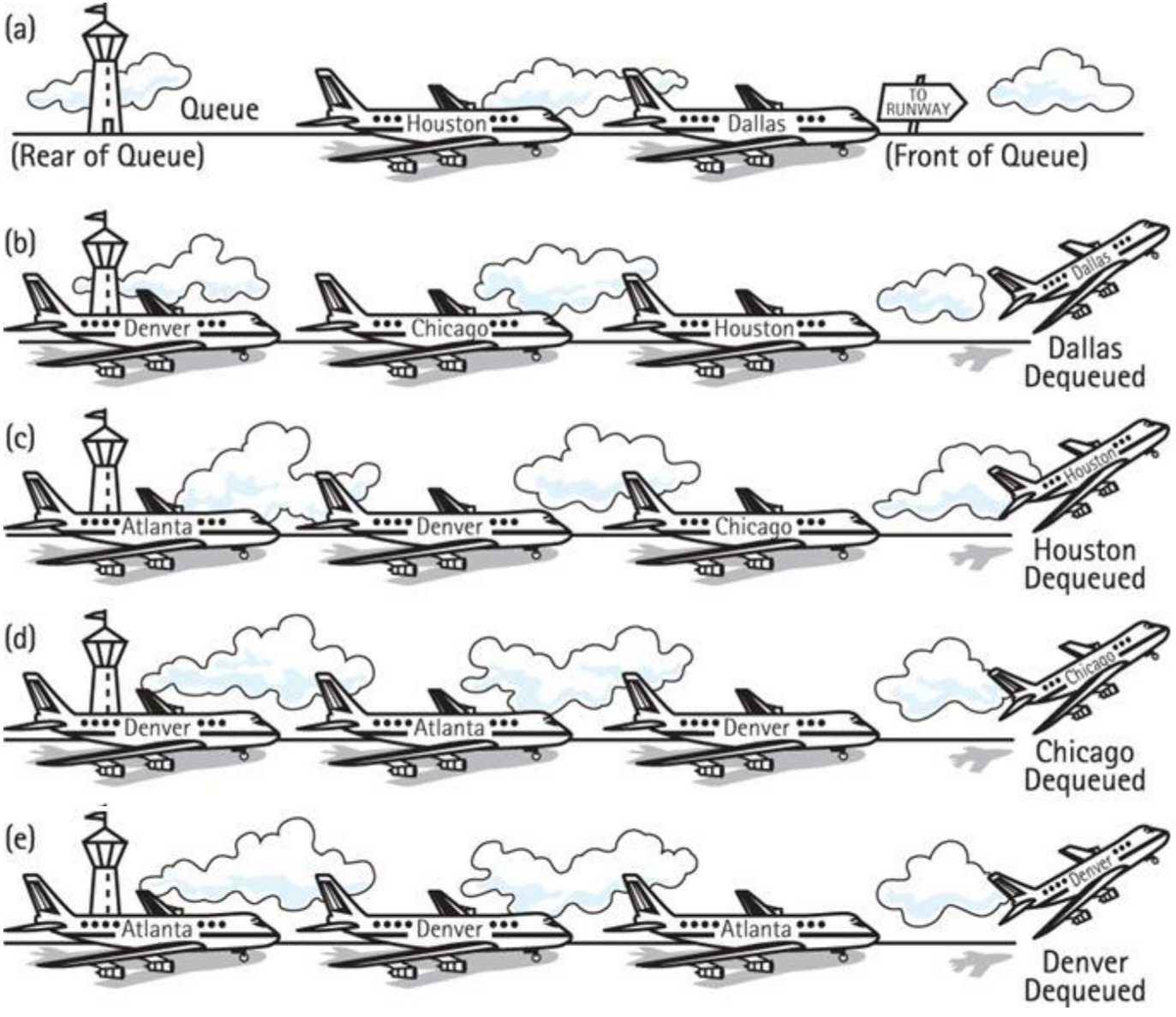
- Main idea:
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up **as far as possible** when you reach a "dead end“
    - i.e., next vertex has been "marked" or there is no next vertex

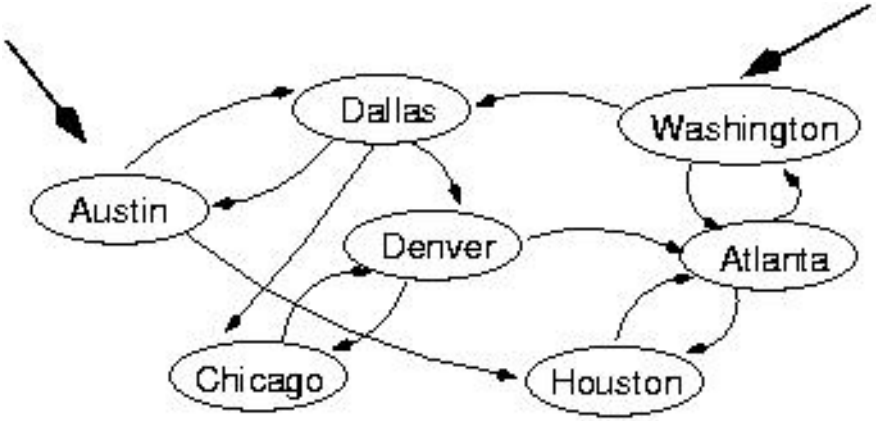
Breadth First: Follow Across



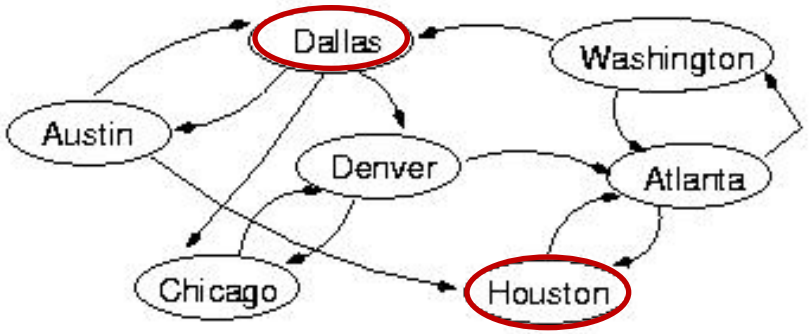
BFS uses Queue !

Breadth First Uses Queue

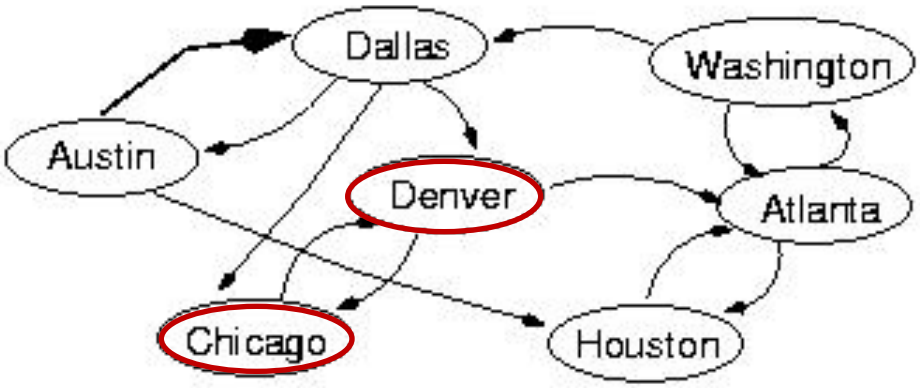




(initialization)

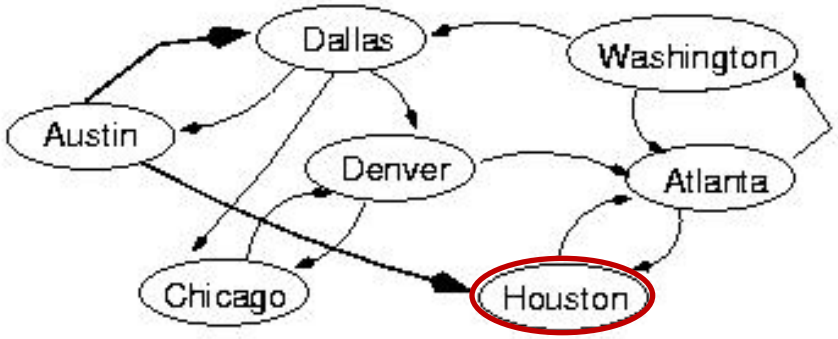


dequeue Austin



dequeue Dallas

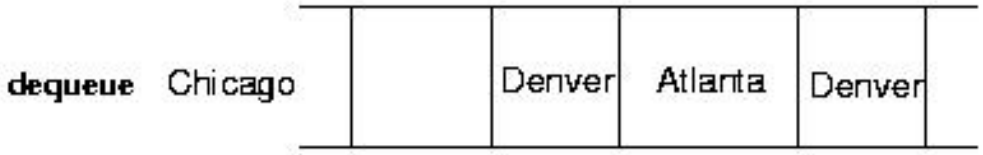
	Houston	Chicago	Denver

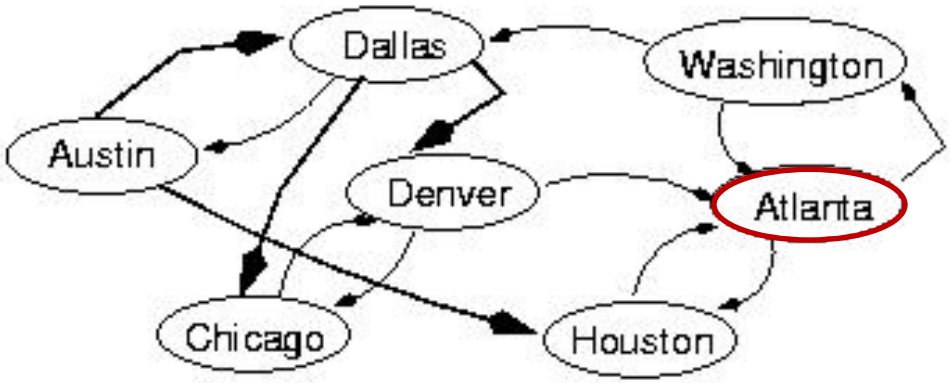


dequeue Houston

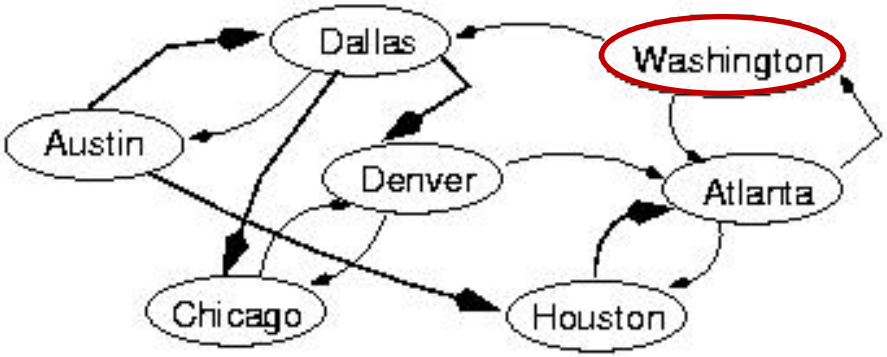
	Chicago	Denver	Atlanta


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston

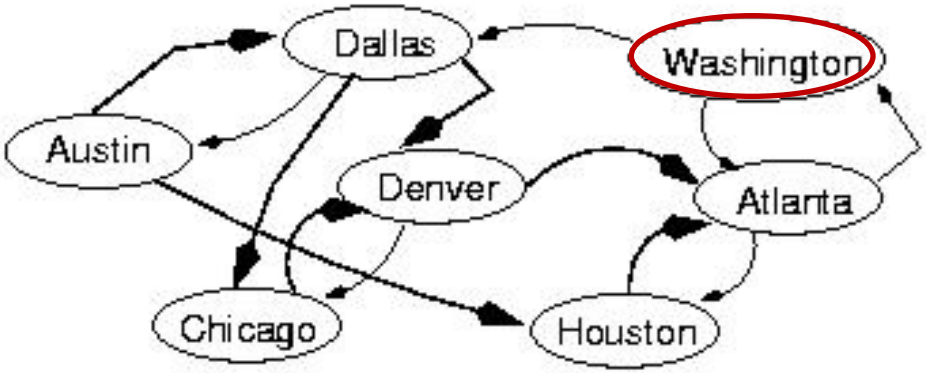

dequeue
Chicago
Denver
Atlanta
Denver


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston

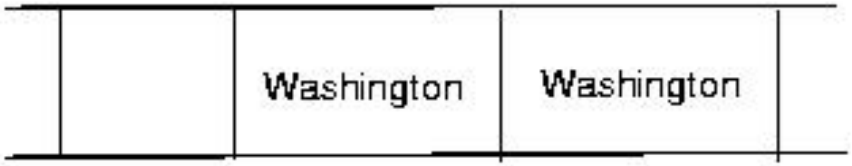

dequeue
Denver
Atlanta
Denver
Atlanta

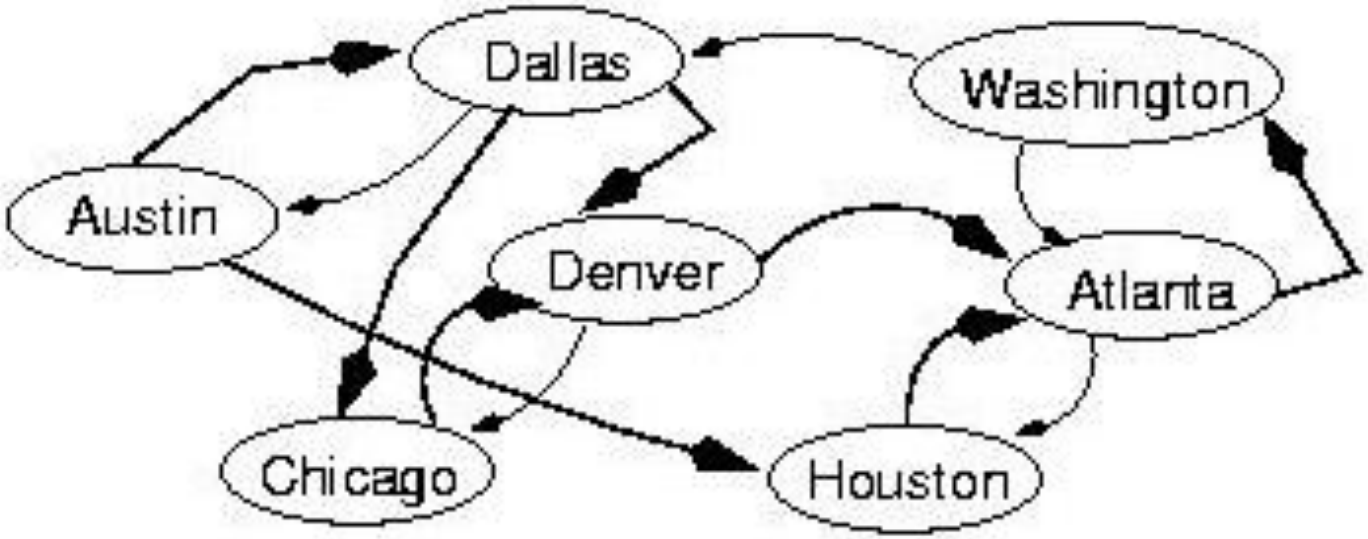


dequeue Atlanta



dequeue Denver, Atlanta




Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston


dequeue
Washington
Washington

Breadth-First Search

- Visits all vertices adjacent to vertex before going forward
- Breadth-first search uses a queue