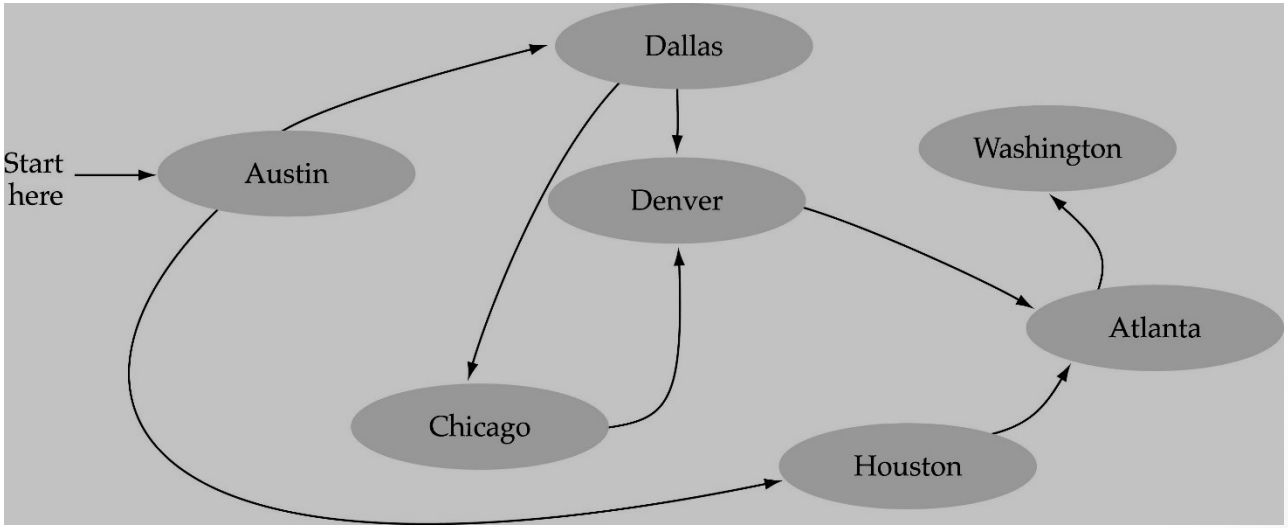
Graphs

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph *G* is defined as follows:

$$G=(V,E)$$

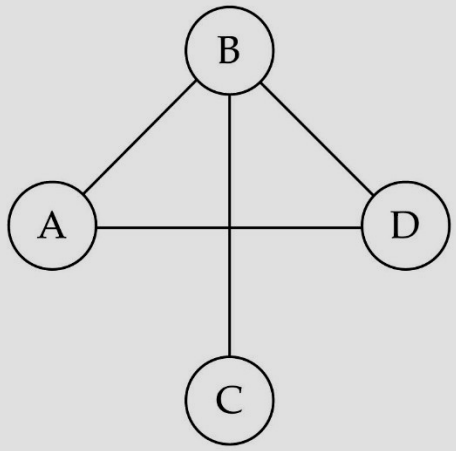
V(G): a finite, nonempty set of vertices

E(G): a set of edges (pairs of vertices)

Directed vs. undirected graphs

- When the edges in a graph have no direction, the graph is called *undirected*

a) Graph1 is undirected graph.

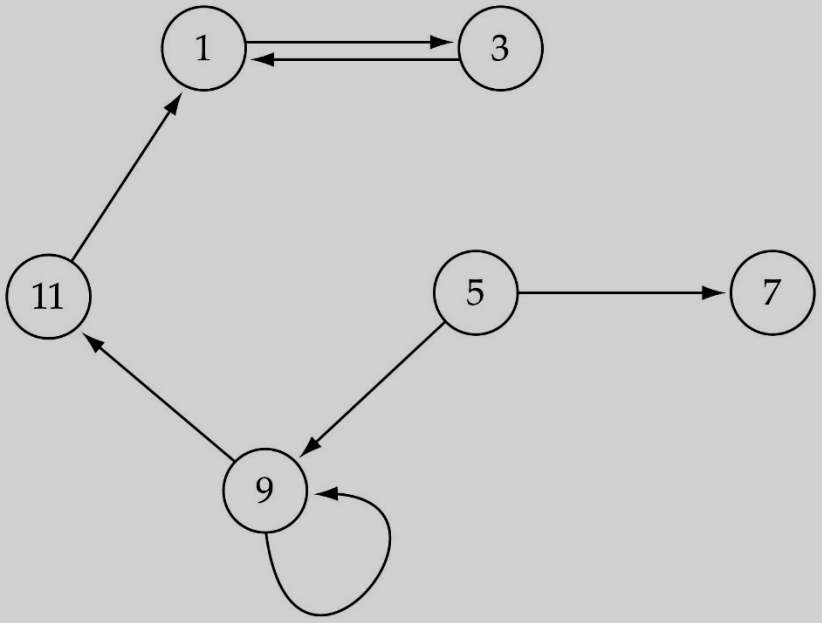


V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



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

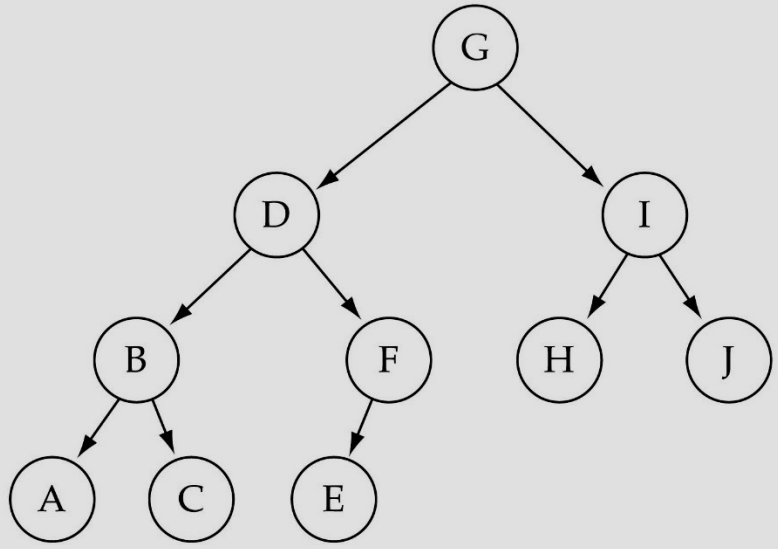
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7),(9.9),(11,1)}

Warning: if the graph is directed, the order of the vertices in each edge is important !!

Trees vs graphs

- Trees are special cases of graphs!!

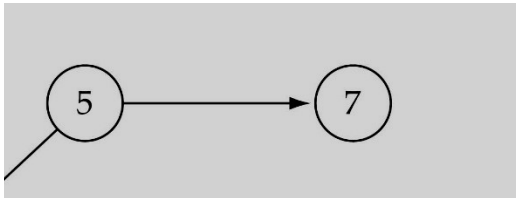
(c) Graph3 is a directed graph.



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

Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7
7 is adjacent from 5

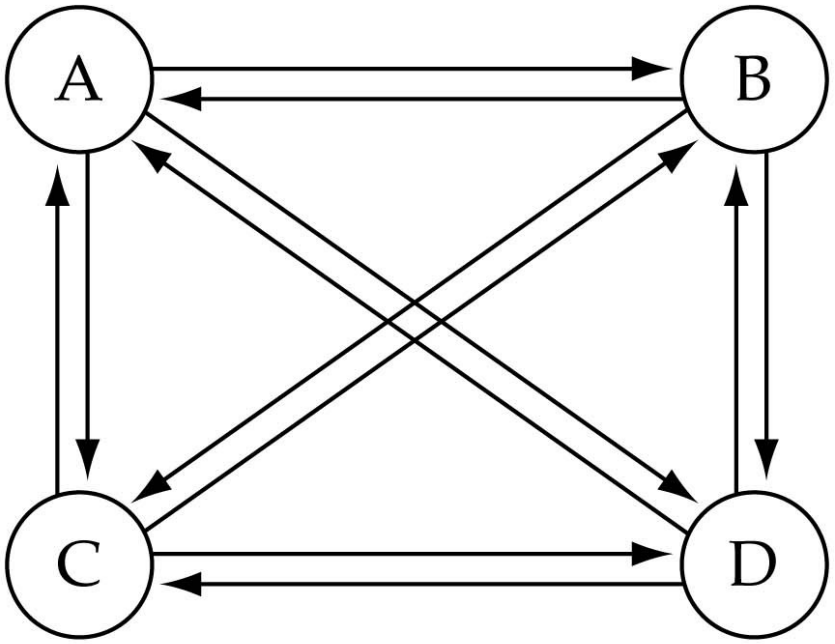
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

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

*N * (N-1)*

$O(N^2)$



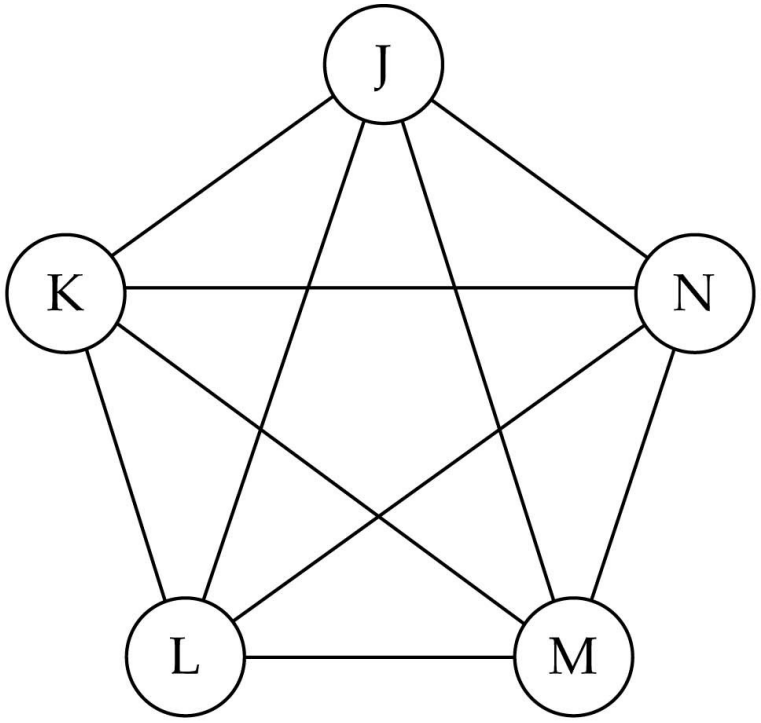
(a) Complete directed graph.

Graph terminology (cont.)

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

*N * (N-1) / 2*

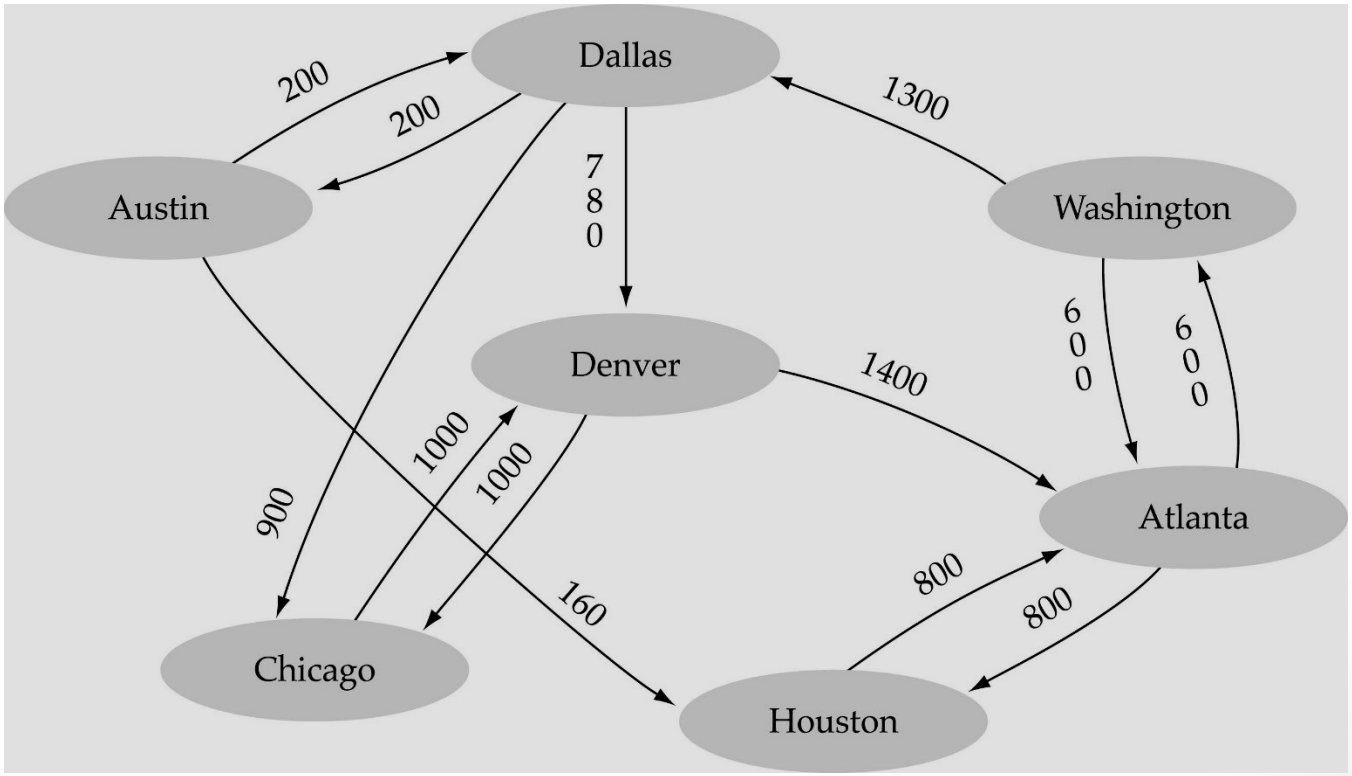
$O(N^2)$



(b) Complete undirected graph.

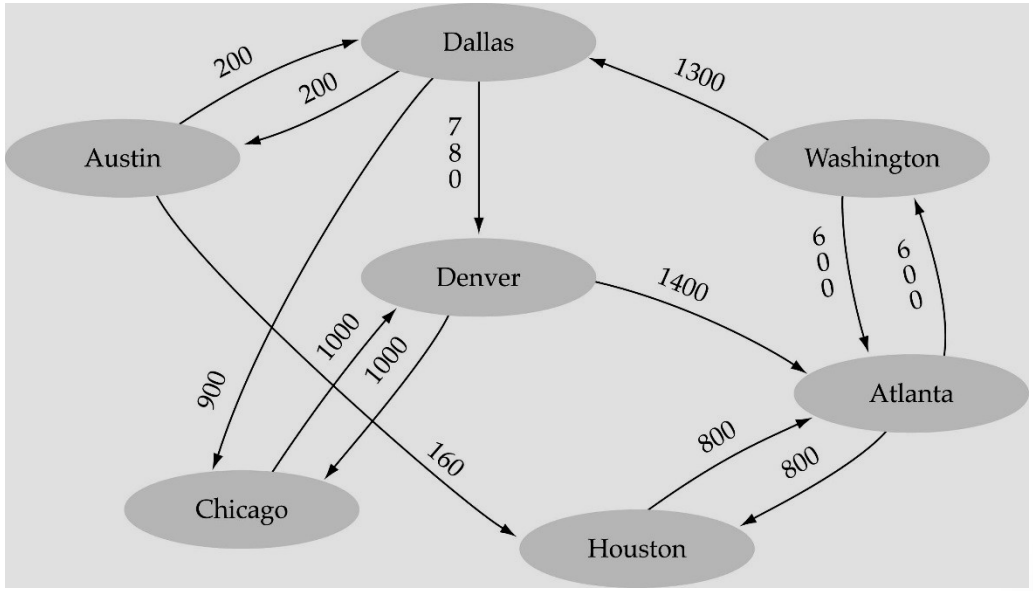
Graph terminology (cont.)

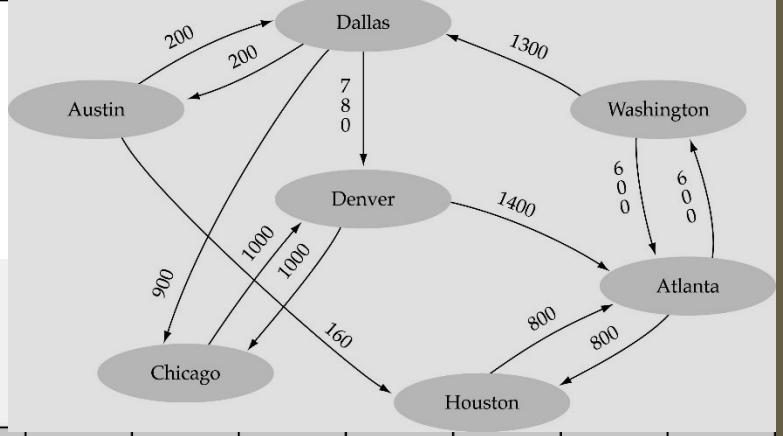
- Weighted graph: a graph in which each edge carries a value



Graph implementation

- Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges





graph

.numVertices 7

.vertices

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

.edges

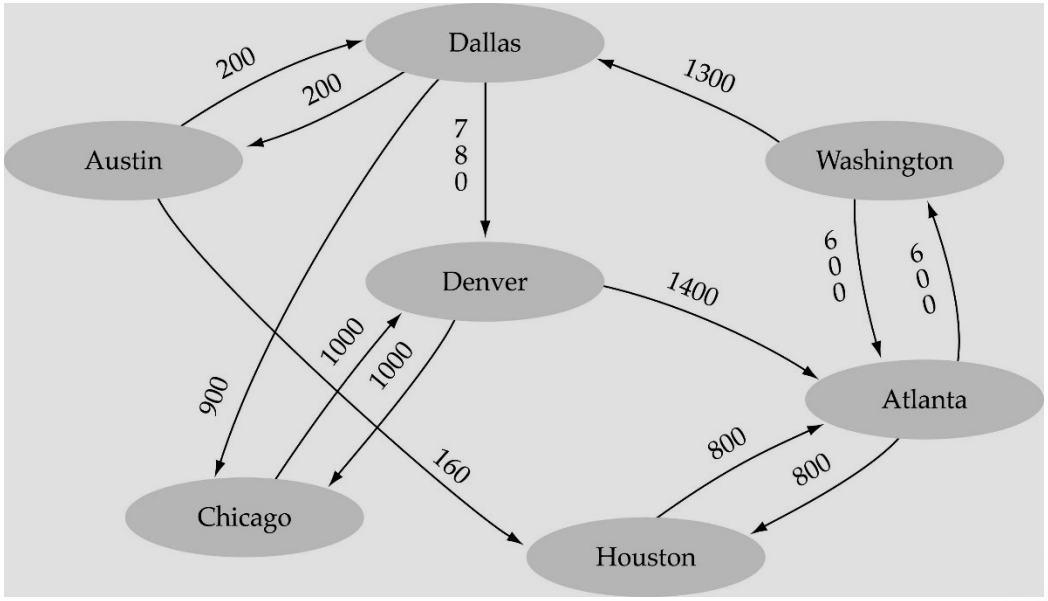
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•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

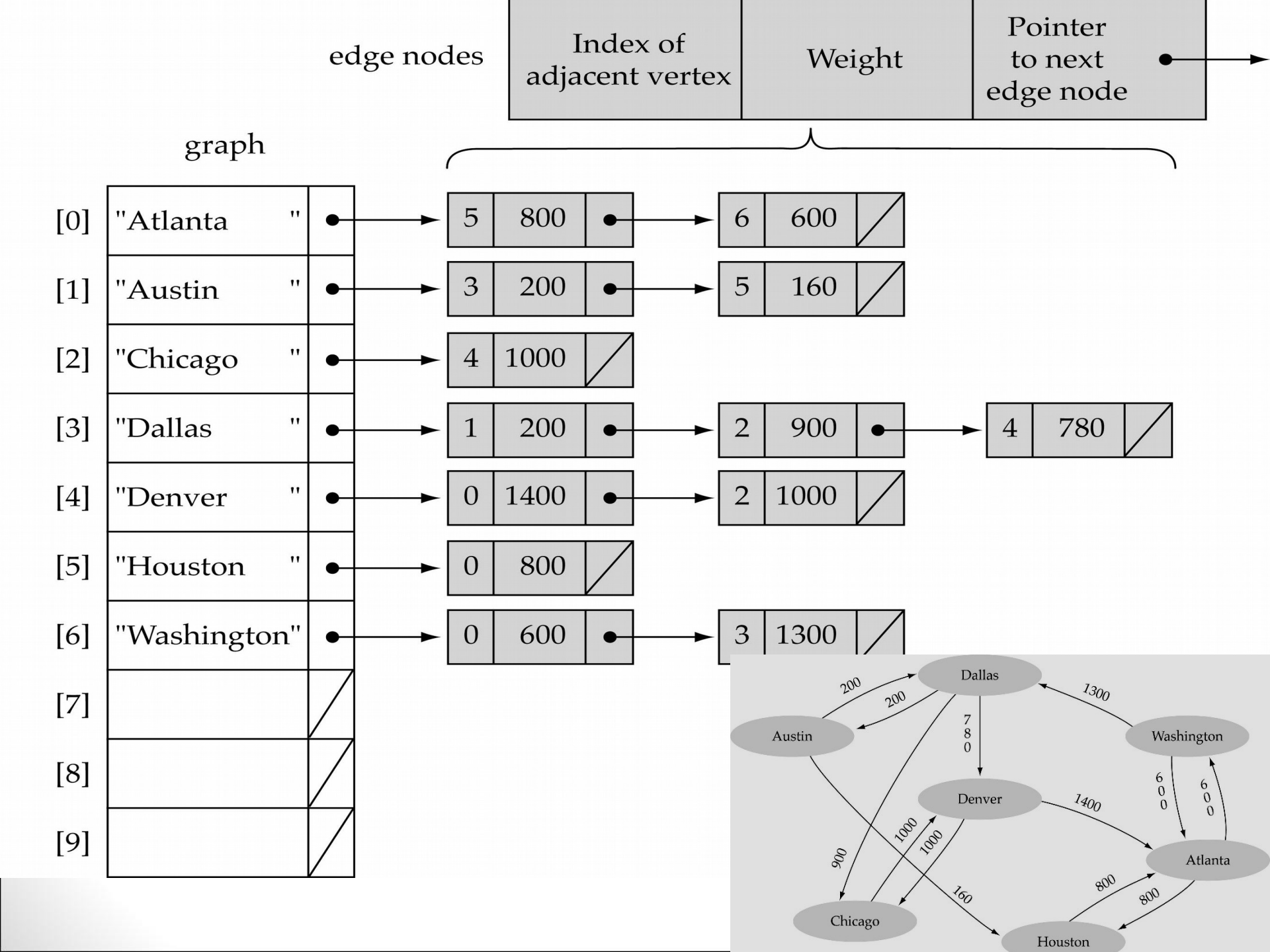
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

(Array positions marked '•' are undefined)

Graph implementation (cont.)

- Linked-list implementation
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)




edge nodes
Index of adjacent vertex
Weight
Pointer 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
1300
780
600
600
1400
1000
1000
900
160
800
800

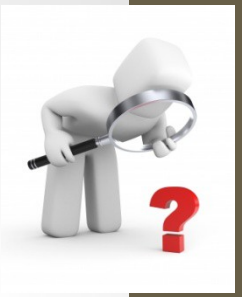
Search Techniques

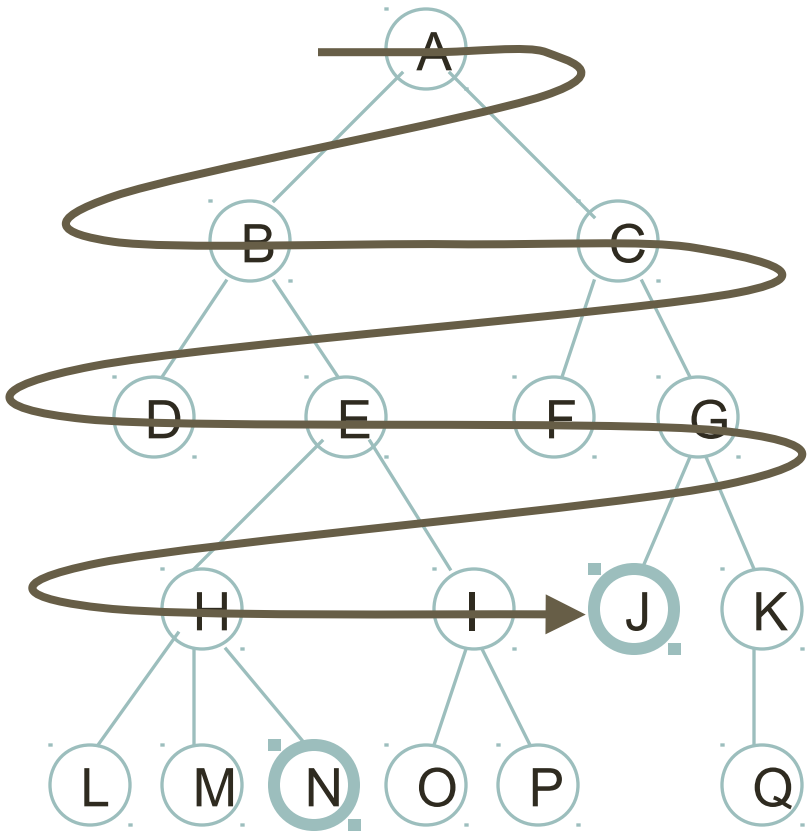
- There are two types of Search techniques:

  Breadth first search,

  Depth first search

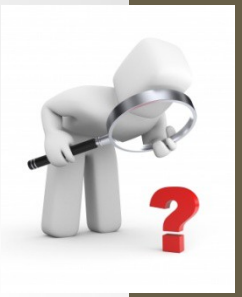
Blind Search: Breadth-first searching





- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away.
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

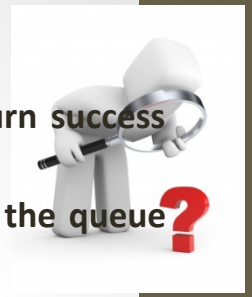
Breadth First Search: Algorithm



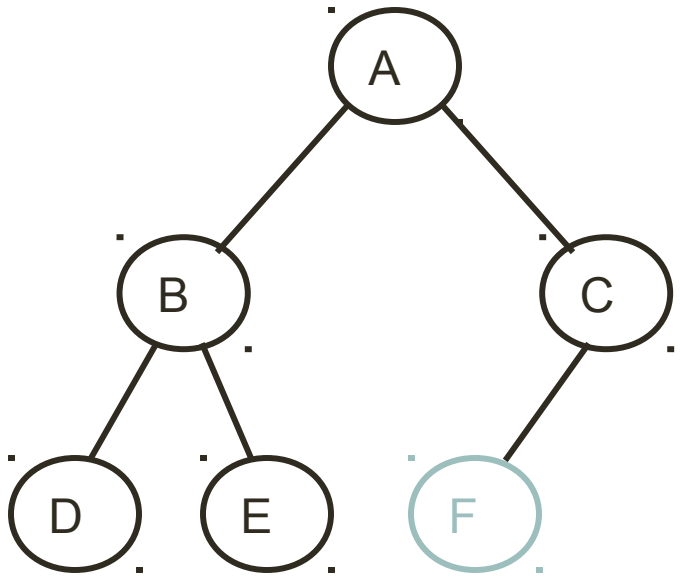
1. Place the starting node s in the queue.
2. If the queue is empty, return failure and stop.
3. If the first element in the queue is a goal state g, return success and stop.
4. Otherwise, remove and expand the first element from the queue and place all children at the tail of the queue.
5. Return to step 2.

Breadth First Search: Dry run

1. Place the starting node s in the queue.
2. If the queue is empty, return failure and stop.
3. If the first element in the queue is a goal state g, return success and stop.
4. Otherwise, remove and expand the first element from the queue and place all children at the tail of the queue.
5. Return to step 2.



Tree



Queue

Front Initial State: A Tail

A					

Expand A

B	C				

Expand B (front) and place at tail

C	D	E			

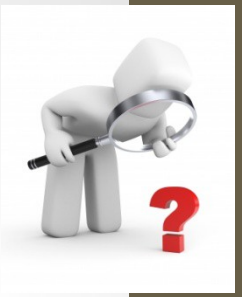
Expand C (front) and place at tail

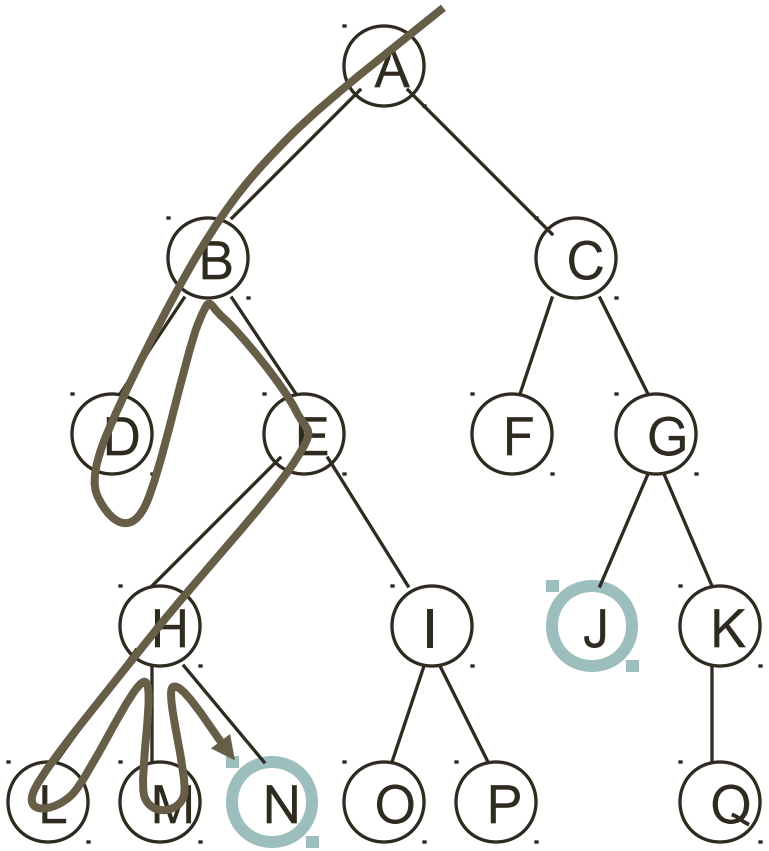
D	E	F			

F					

Simply Remove D, then E, as they have no children and the goal F is found

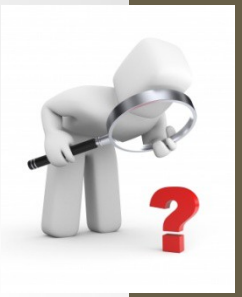
Blind Search: Depth-first searching





- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

Depth First Search: Algorithm

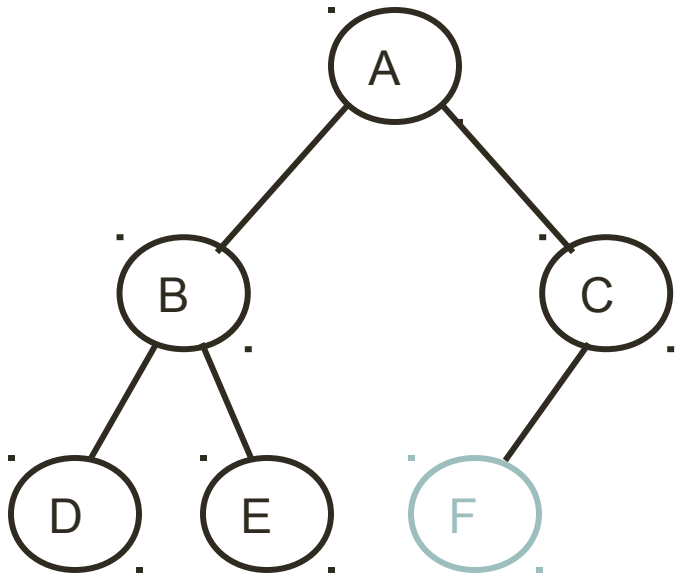


1. Place the starting node s in the queue.
2. If the queue is empty, return failure and stop.
3. If the first element in the queue is a goal state g, return success and stop.
4. Otherwise, remove and expand the first element from the queue and place all children at the front of the queue.
5. Return to step 2.

Depth First Search: Dry run

1. **Place the starting node s in the queue.**
2. **If the queue is empty, return failure and stop.**
3. **If the first element in the queue is a goal state g, return success and stop.**
4. **Otherwise, remove and expand the first element from the queue and place all children at the front of the queue.**
5. **Return to step 2.**

Tree



Queue

Front Initial State: A Tail

A					

Expand A

B	C				

Expand B (front) and place at front

D	E	C			

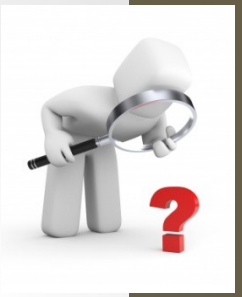
Simply remove D and then E, as they have no children.

Expand C (front) and place at tail

F					

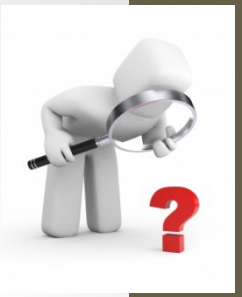
The goal F is found

DFS: Backtracking



- In this search, a single branch of the tree has to pursue until it yields a solution or until a decision to terminate the path is made.
- It makes sense to terminate a path if it reaches dead-end, produces a previous state. In such a state backtracking occurs.
- Chronological Backtracking: Order in which steps are undone depends only on the temporal sequence in which steps were initially made.
- Specifically most recent step is always the first to be undone.
- This is also simple backtracking.

BFS Vs DFS



S.No	BFS	DFS
1	For a large tree, memory requirements may be excessive.	It requires less memory since only the nodes on the current path are stored.
2	If there is a solution, BFS is guaranteed to find it. ☐ if the goal is found in shortest paths, then longer paths would not be required to be searched, hence saving time and efforts. ☐ if number of paths are large, then BFS takes lots of time and becomes unmanageable.	Does not guarantee to find an optimal solution because as soon as a solution is found, it will stop the search & solution may not be optimal.
3	If there are multiple solutions, then a minimal solution will be found (i.e. one that requires minimum number of steps).	If there is a single solution, it will be found efficiently.