មហាវិទ្យាល័យវិស្វកម្ម
FACULTY OF ENGINEERING

Data Structure & Algorithm II

Lecture 6
Graph

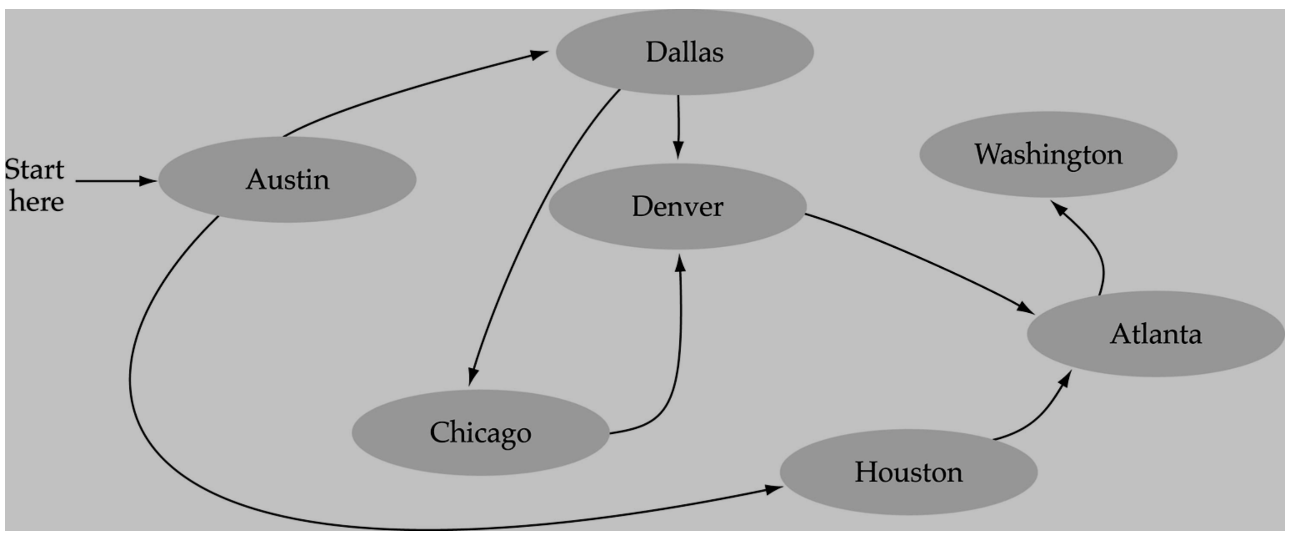
Chhoeum Vantha, Ph.D.
Telecom & Electronic Engineering

Content

- What is a graph?
- Directed vs. undirected graphs
- Trees vs graphs
- Graph terminology
- Graph implementation
- Adjacency matrix vs. adjacency list representation
- Graph searching

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:

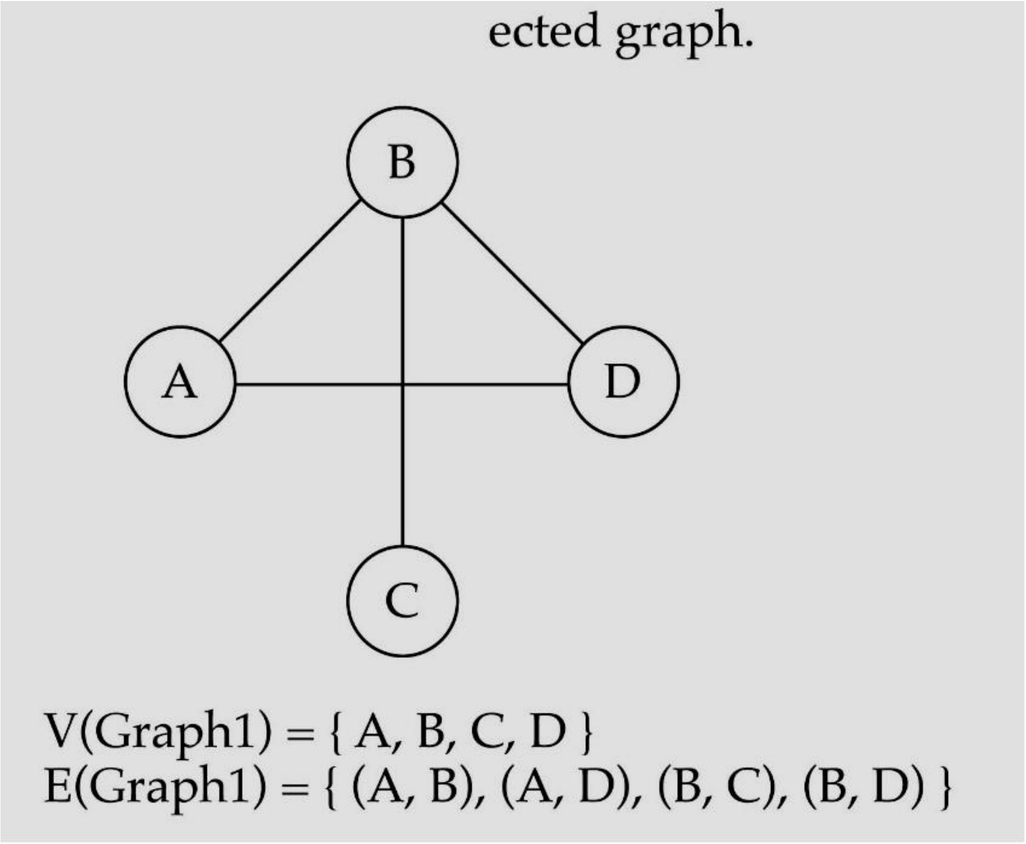
$$G=(V,E)$$

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

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

Directed vs. undirected graphs

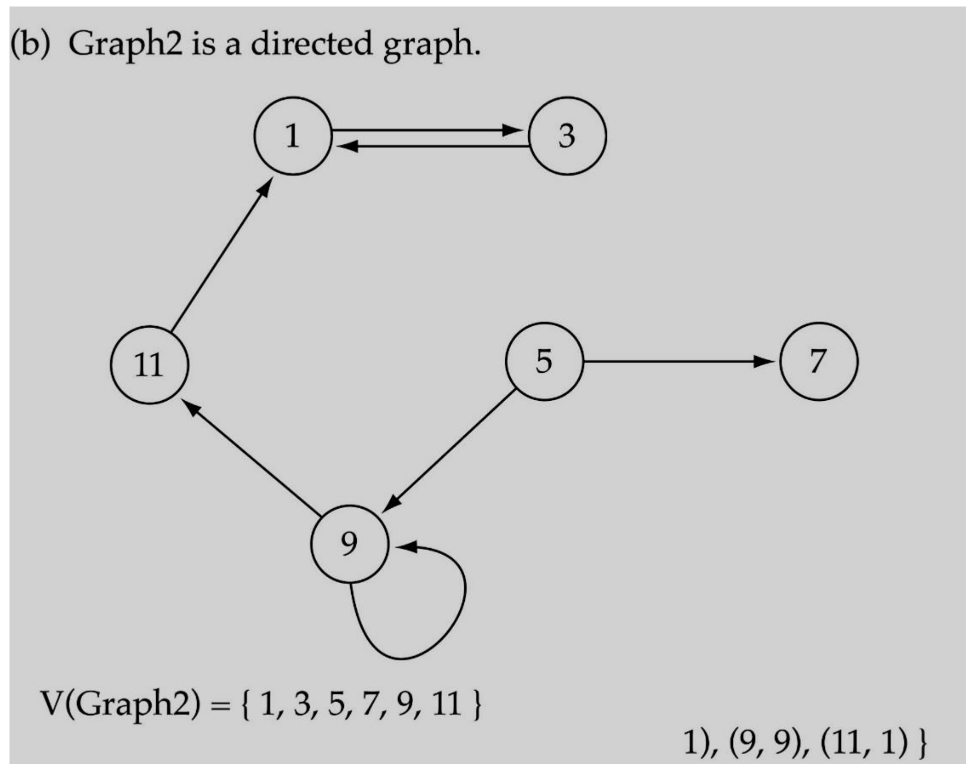
- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs

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

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

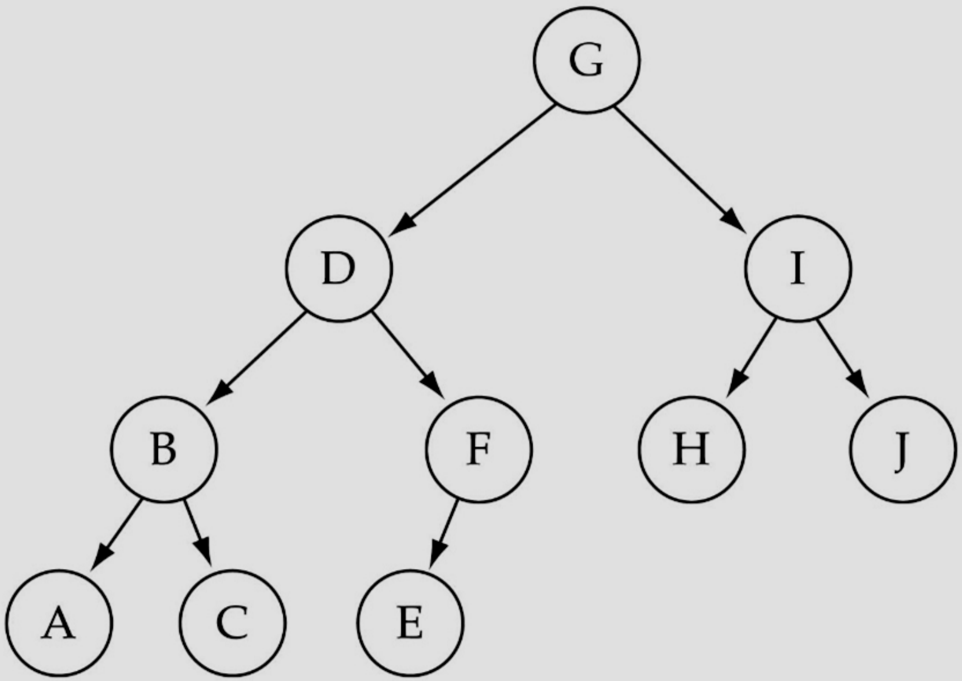


E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7)

Trees vs graphs

- Trees are special cases of graphs!!

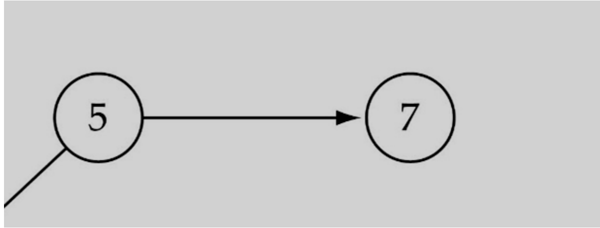
(c) Graph3 is a directed graph.



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

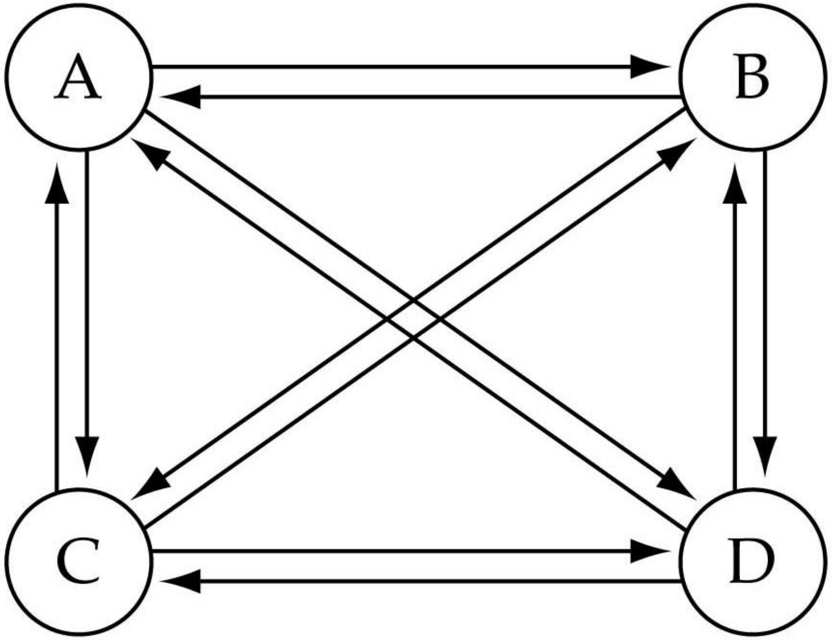
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology

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

N * (N-1)

$$O(N^2)$$



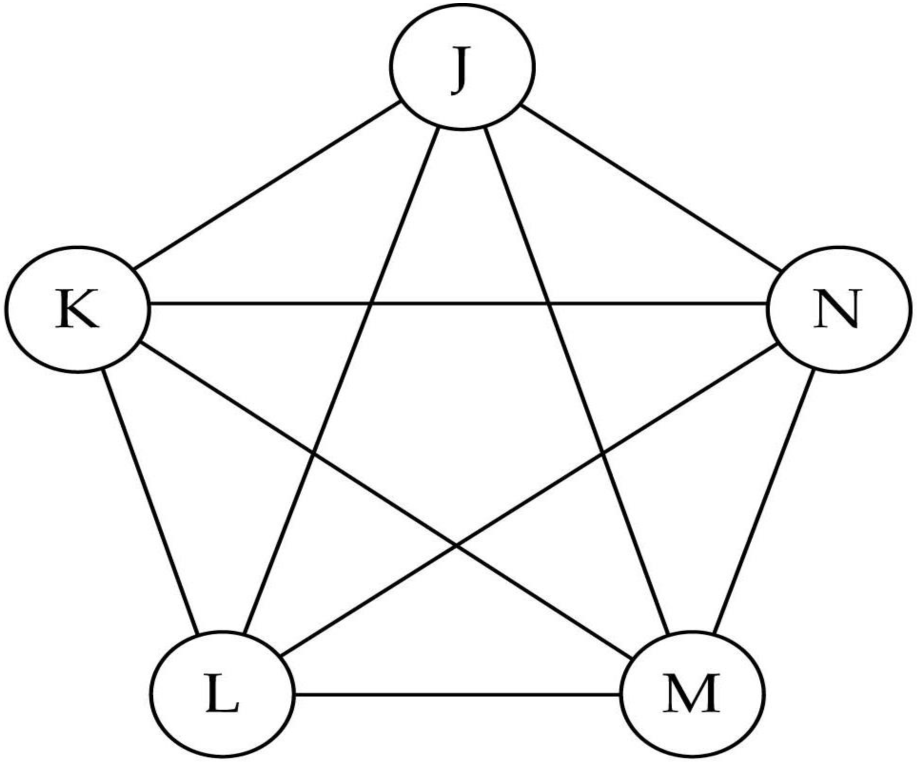
(a) Complete directed graph.

Graph terminology

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

*N * (N-1) / 2*

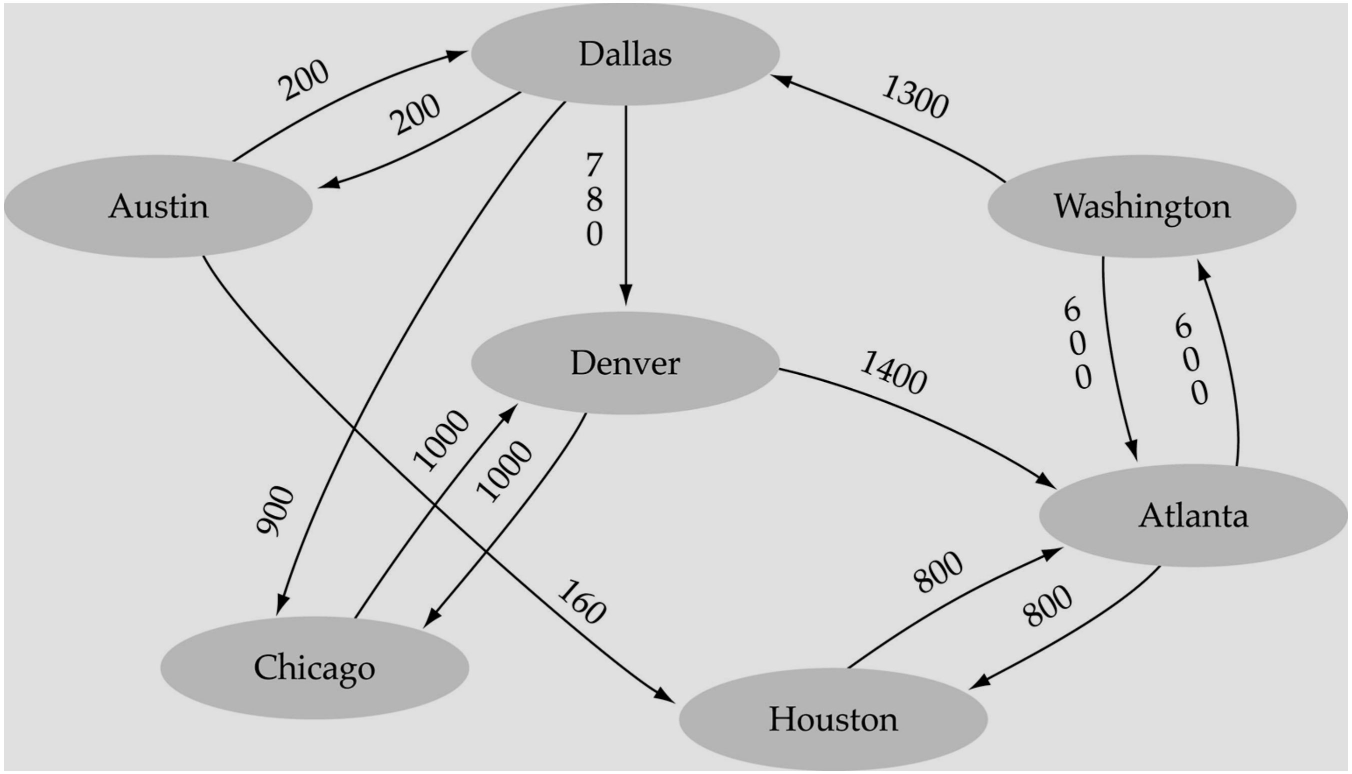
$O(N^2)$



(b) Complete undirected graph.

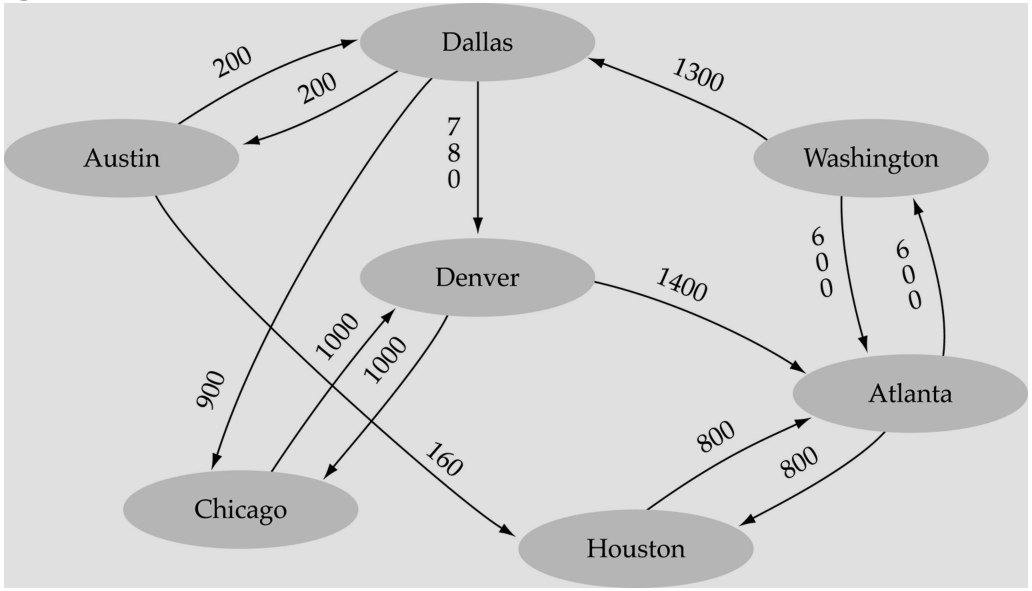
Graph terminology

- **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]	

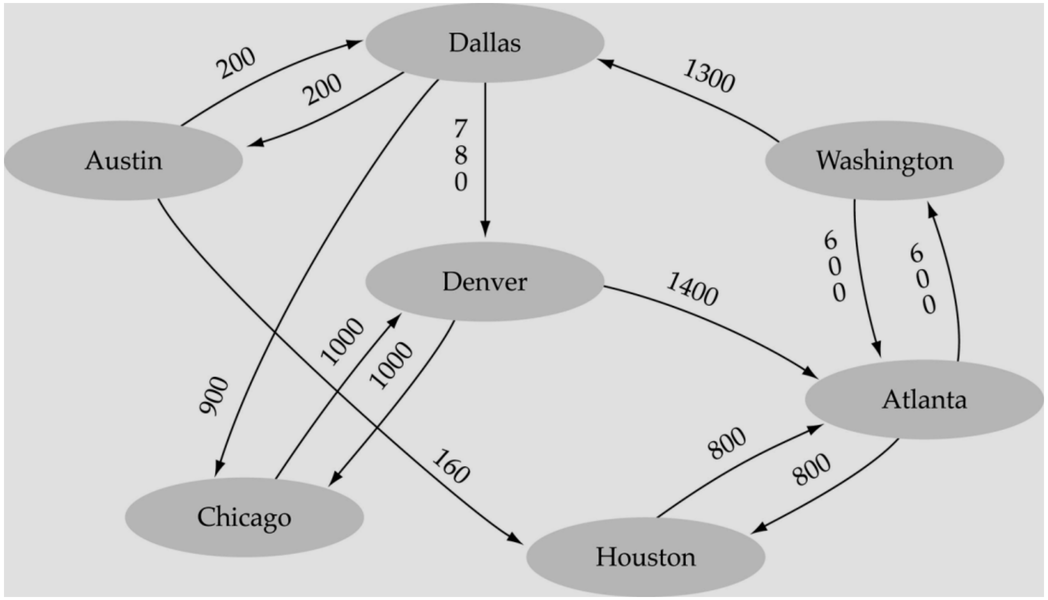
.edges

	[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]	•	•	•	•	•	•	•	•	•	•

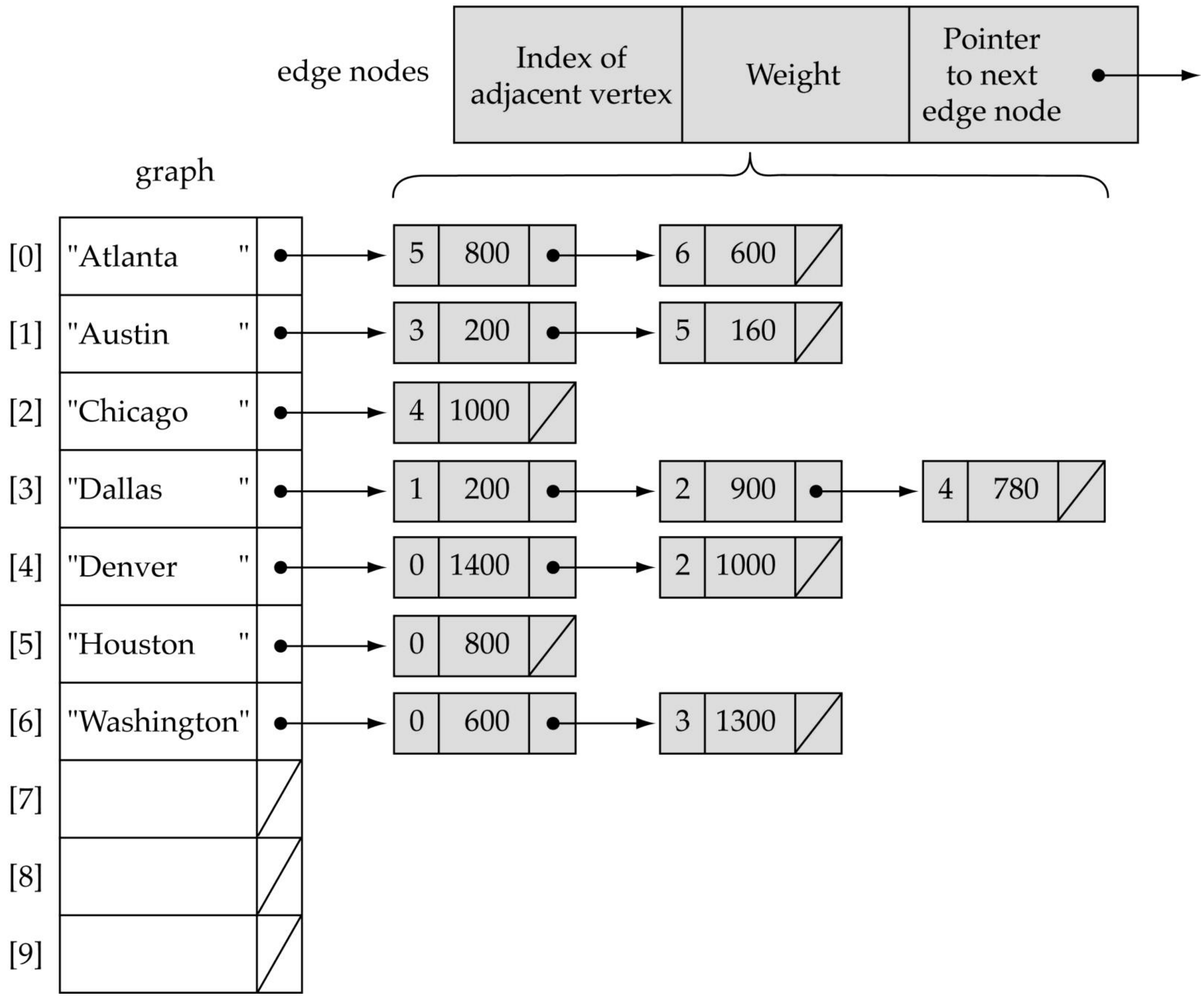
(Array positions marked '•' are undefined)

Graph implementation

- 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)



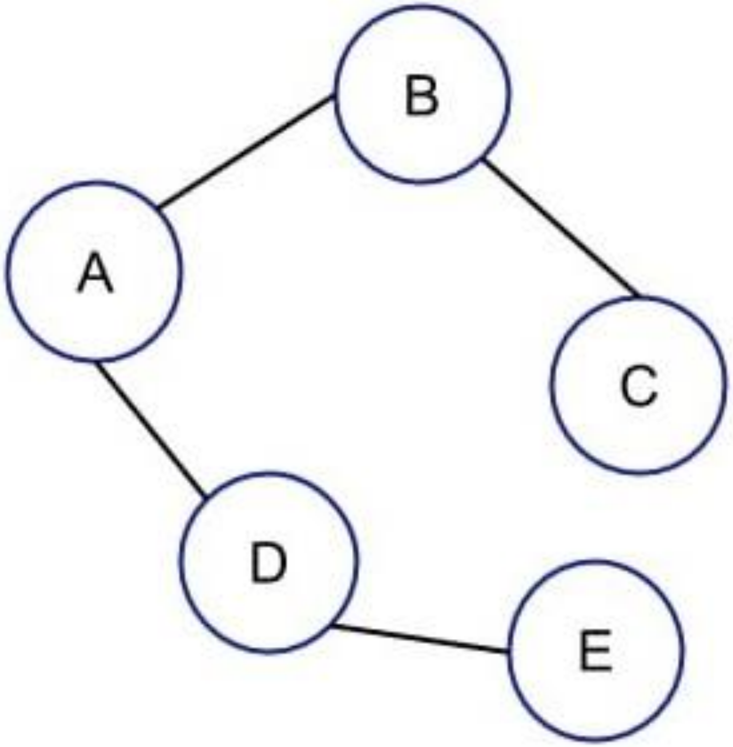
(a)


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

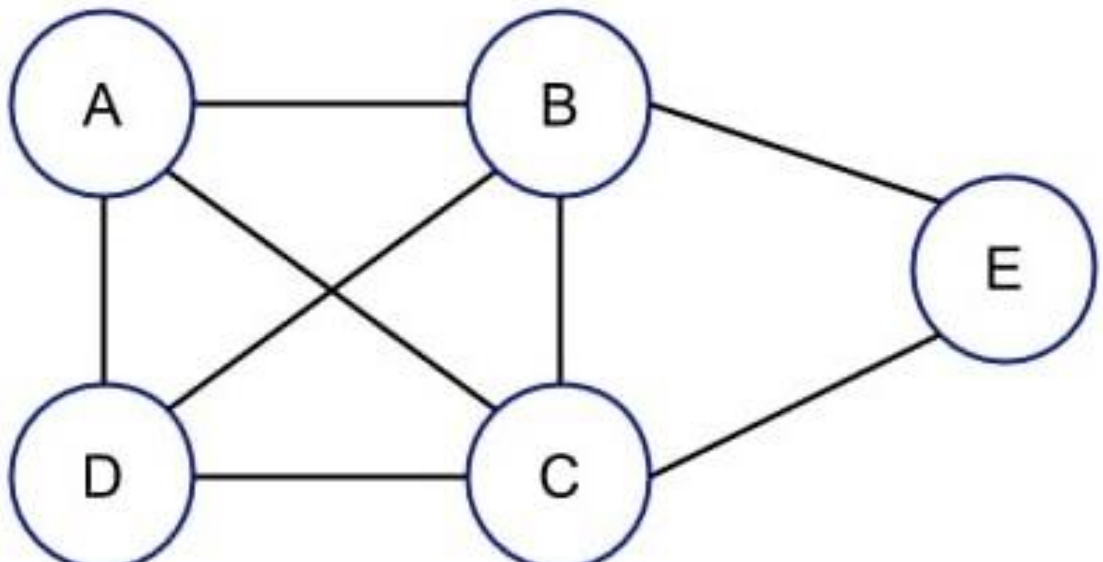
Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
  - Good for dense graphs --$|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly
- **Adjacency list**
  - Good for sparse graphs -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|)=O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

Adjacency matrix vs. adjacency list representation



sparse graphs



dense graphs

Graph searching

- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)
- *Methods*:
  - Depth-First-Search (DFS)
  - Breadth-First-Search (BFS)

Depth-First-Search (DFS)

- What is the idea behind DFS?
  - 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)
- DFS can be implemented efficiently using a *stack*

Breadth-First-Searching (BFS)

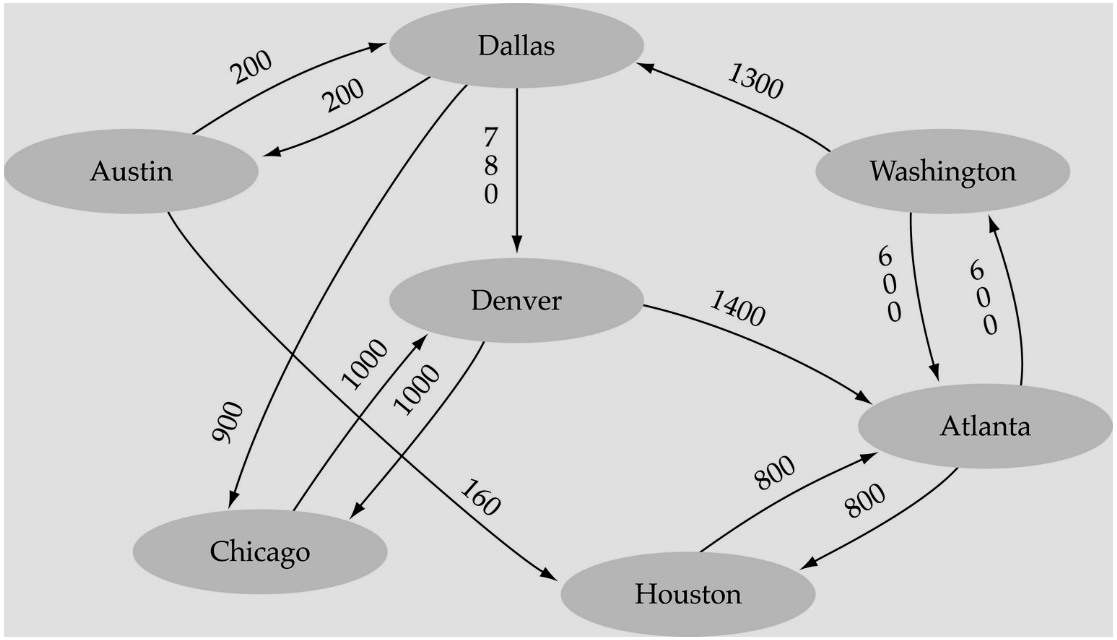
- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- ***Shortest path***: the path whose total weight (i.e., sum of edge weights) is minimum

Single-source shortest-path problem

- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles



Single-source shortest-path problem

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is **weightless** or all the weights are the same

(mark vertices before Enqueue)

W7 – Lab

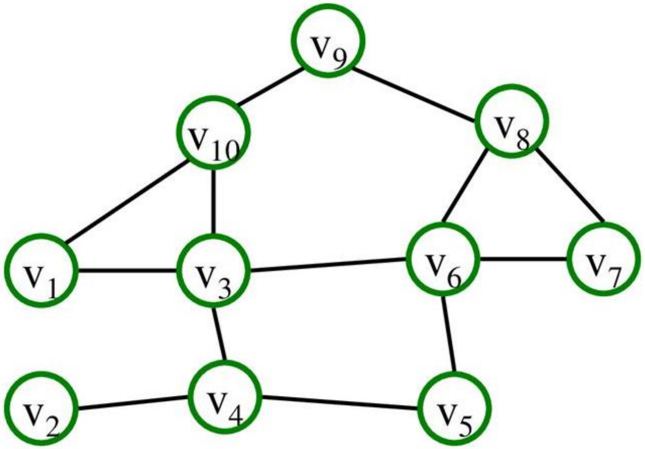
Graph

- G = (V, E)

V are the vertices; E are the edges.

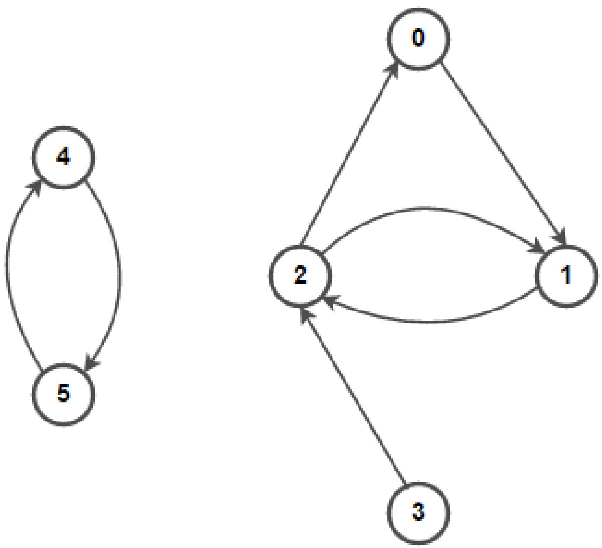
Edges are of the form (v, w), where v, w $\in$ V.

- ordered pair: directed graph or digraph
- unordered pair: undirected gra

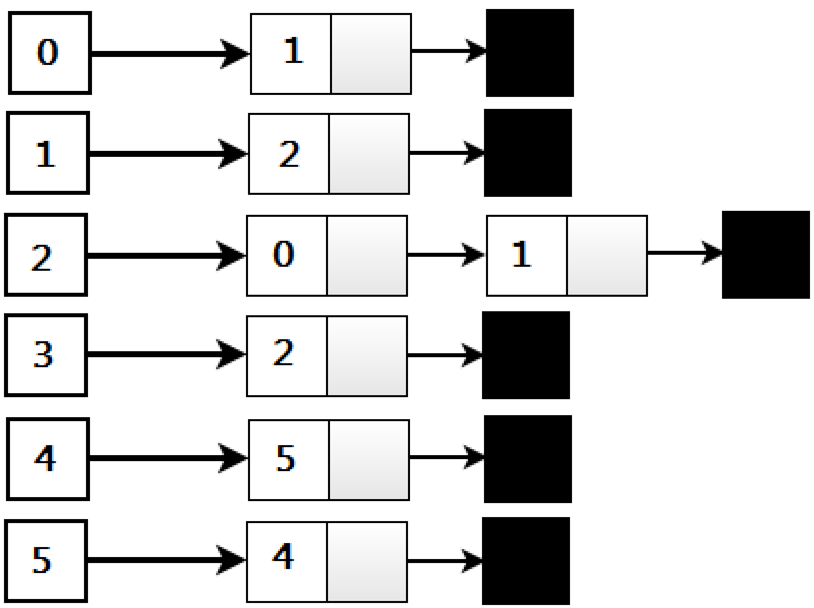


Directed Graph

- Graph



- adjacency list representation of the graph



Directed Graph

```
#include <iostream>
#include <vector>
using namespace std;
// Data structure to store a graph edge
struct Edge {
    int v, w;
};
```

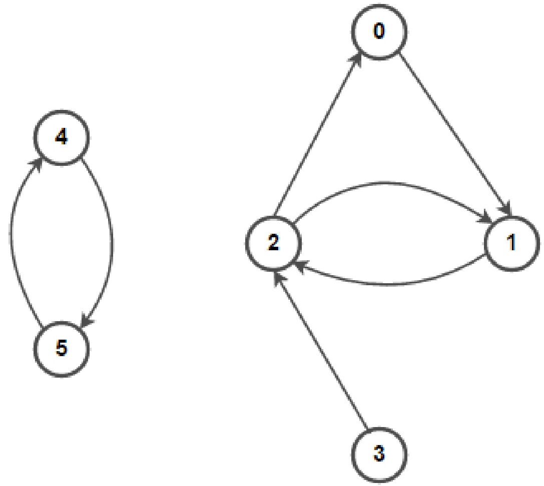
Directed Graph

```
class Graph
{
public:
    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;
    // Graph Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements of type `vector<int>`
        adjList.resize(n);
        // add edges to the directed graph
        for (auto &edge: edges)
        {
            // insert at the end
            adjList[edge.v].push_back(edge.w);
            // uncomment the following code for undirected graph
            // adjList[edge.w].push_back(edge.v);
        }
    }
};
```

Directed Graph

```
// Function to print adjacency list representation of a graph
void printGraph(Graph const &graph, int n)
{
    for (int i = 0; i < n; i++)
    {
        // print the current vertex number
        cout << i << " -> ";
        // print all neighboring vertices of a vertex `i`
        for (int v: graph.adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
}
```

Directed Graph



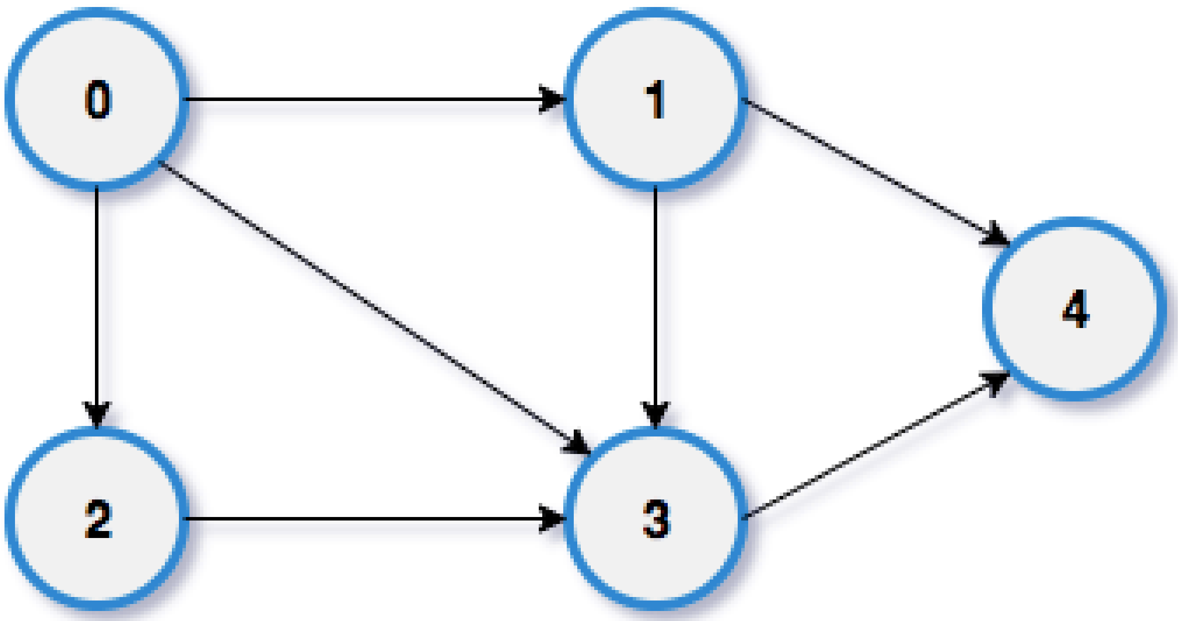

```cpp
// Graph Implementation using STL
int main()
{
    // vector of graph edges as per the above diagram.
    // Please note that the initialization vector in the below format will
    // work fine in C++11, C++14, C++17 but will fail in C++98.
    vector<Edge> edges =
    {
        {0, 1}, {1, 2}, {2, 0}, {2, 1}, {3, 2}, {4, 5}, {5, 4}
    };
    // total number of nodes in the graph (labelled from 0 to 5)
    int n = 6;
    // construct graph
    Graph graph(edges, n);
    // print adjacency list representation of a graph
    printGraph(graph, n);

    return 0;
}
```

Ex. 1 – Graph 1

a) Execute in Main() graph in the figure below

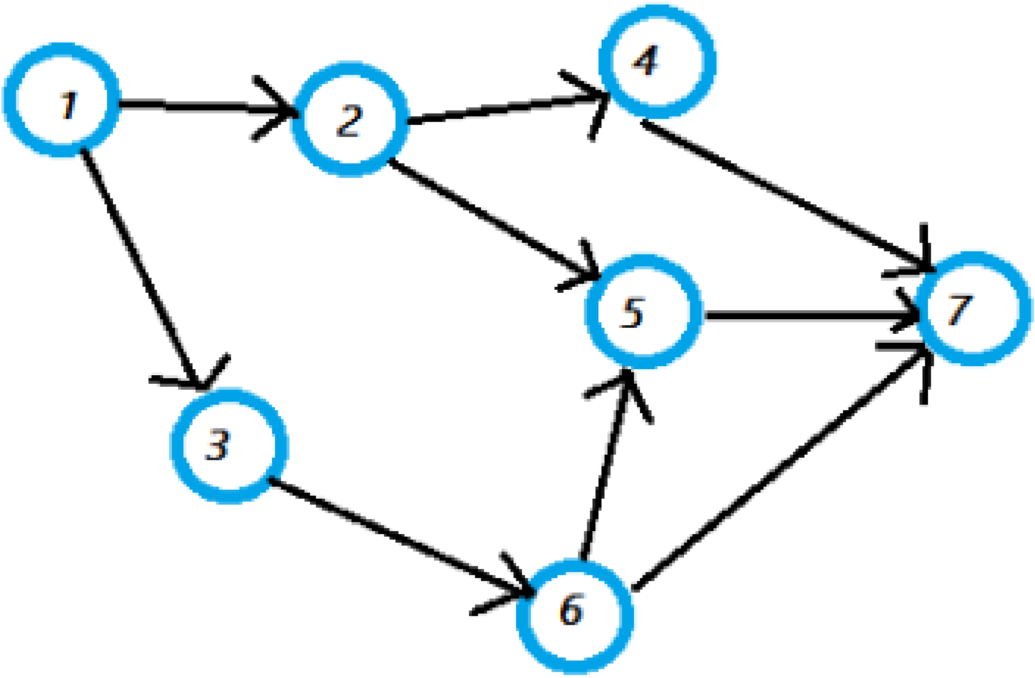
b) Draws an adjacency list representation of the graph



Ex. 2 – Graph 2

a) Execute in Main() graph in the figure below

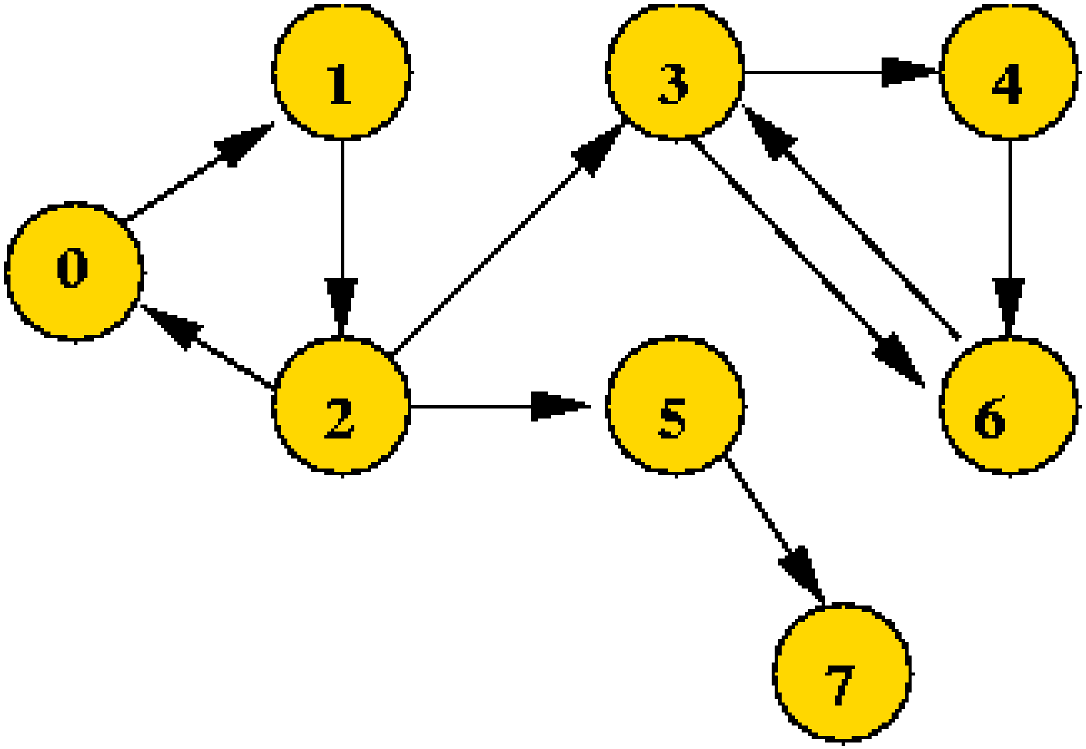
b) Draws an adjacency list representation of the graph



Ex. 3 – Graph 3

a) Execute in Main() graph in the figure below

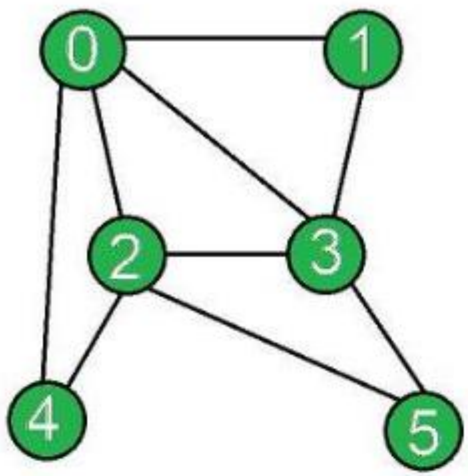
b) Draws an adjacency list representation of the graph



Ex. 4 – Graph 4

a) Base on the code above, change from a directed graph to an undirected graph then

b) Execute in Main() graph in the figure below



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

Ex. 5 – Teamwork

a) Explain code line by line

b) Show briefly about three step of each function in the graph

Thanks!