Lecture Seven

Graph, Depth and Breath First Search

MSc Karar Sadiq Mohsin

Email: karar.sadeq@uowa.edu.iq

General Objective

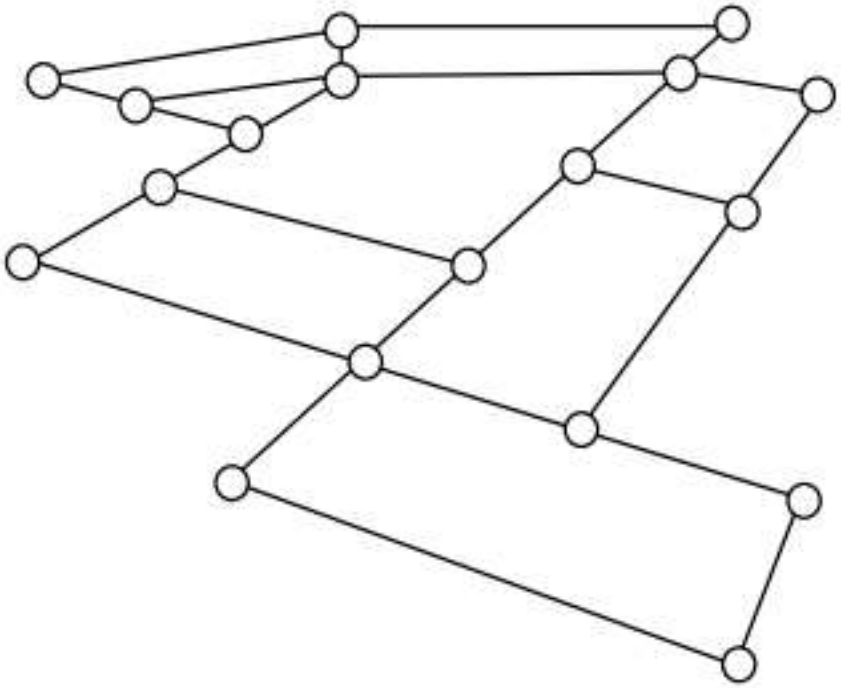
- To understand the concept of graphs, their representation in Python, and the implementation of key graph algorithms, emphasizing the importance of selecting the correct data representation for efficient graph algorithms.

Behavioral Objectives

- **Define Graphs**: Students will be able to define what a graph is and distinguish it from other types of graphs such as pie graphs or bar graphs.
- **Graph Representation in Python**: Students will demonstrate how to represent graphs in Python.
- **Implement Graph Algorithms**: Students will implement important graph algorithms using Python.
- **Data Structure Selection**: Students will analyze and select the appropriate data representation for efficient graph algorithms

GRAPH DEFINITIONS

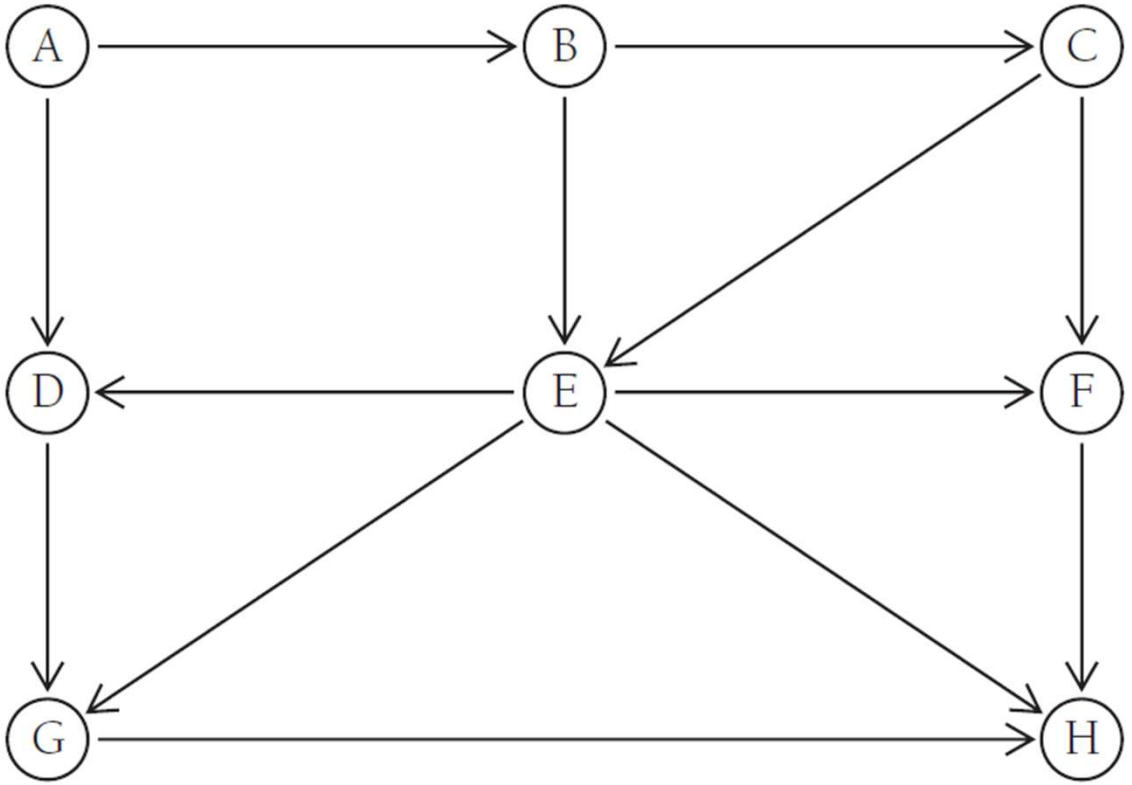
- **A graph consists of a set of vertices and a set of edges.**
- Think of a map of your state. Each town is connected with other towns via some type of road.
- A map is a type of graph. Each town is a vertex and a road that connects two towns is an edge.
- Edges are specified as a pair, (v1, v2), where v1 and v2 are two vertices in the graph.
- A vertex can also have a weight, sometimes also called a cost.



An Ordered Graph

- A graph whose pairs are ordered is called a directed graph, or just a digraph.
- An ordered graph is shown in Figure 1.

Figure 1
An Ordered Graph



An Unordered Graph

- If a graph is not ordered, it is called an unordered graph, or just a graph.
- An example of an unordered graph is shown in Figure 2.

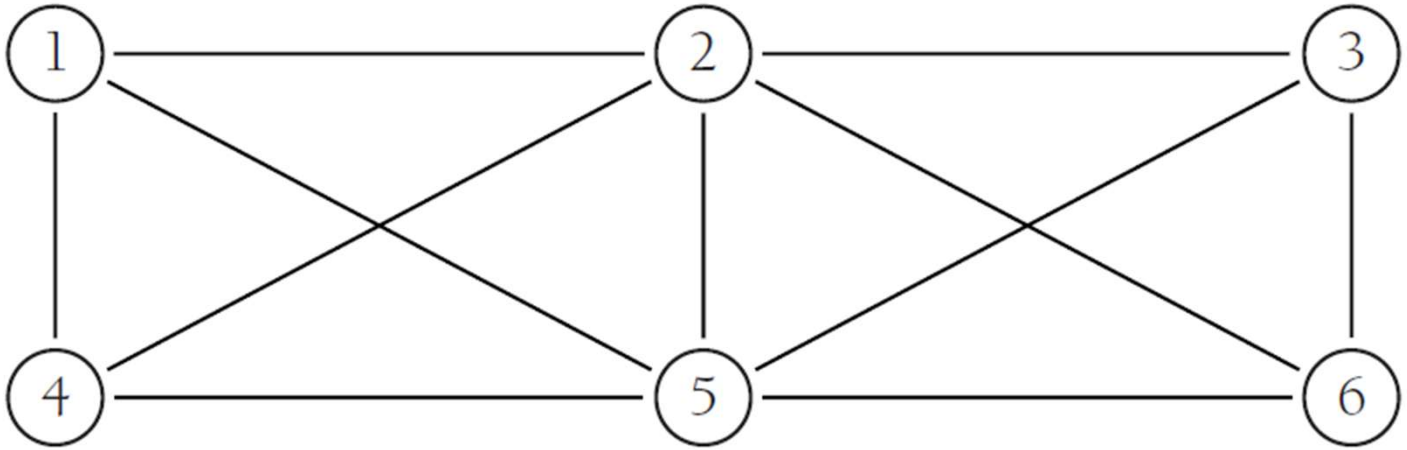


Figure 2
an unordered graph

A Path

- A path is a sequence of vertices in a graph such that all vertices are connected by edges.
- The length of a path is the number of edges from the first vertex in the path to the last vertex.
- A path can also consist of a vertex to itself, which is called a loop. Loops have a length of 0.

A Path

- A cycle is a path of at least 1 in a directed graph so that the beginning vertex is also the ending vertex.
- In a directed graph, the edges can be the same edge,
- but in an undirected graph, the edges must be distinct.

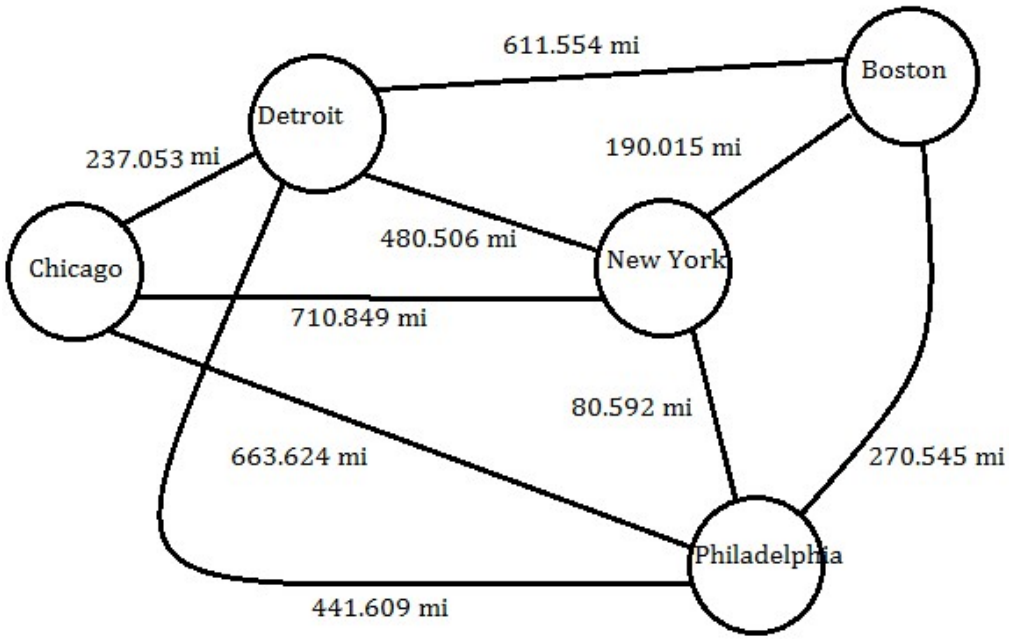
A Path

- An undirected graph is considered connected if there is a path from every vertex to every other vertex.
- In a directed graph, this condition is called strongly connected.
- A directed graph that is not strongly connected, but is considered connected, is called weakly connected.
- If a graph has a edge between every set of vertices, it is said to be a complete graph.

REAL WORLD SYSTEMS MODELED BY GRAPHS

- Graphs are used to model many different types of intersections, systems.
- One example is traffic flow.
- The vertices represent street intersections, and the edges represent the streets themselves.
- Weighted edges can be used to represent speed limits or the number of lanes.
- Modelers can use the system to determine best routes and streets that are likely to suffer from traffic jams.

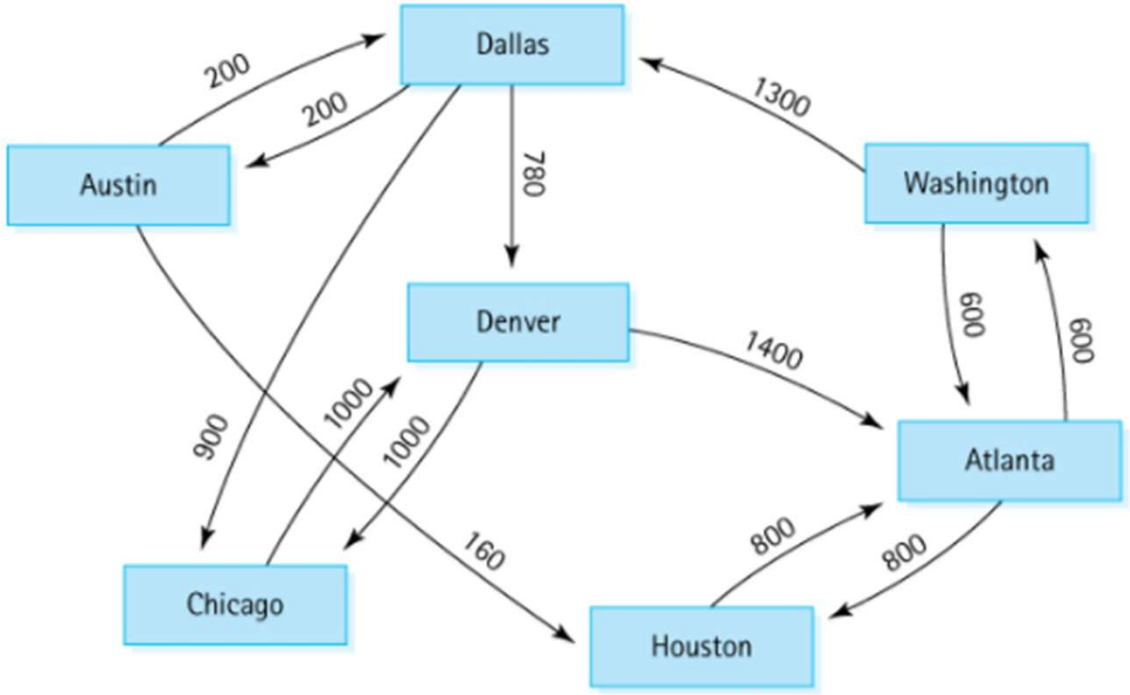
Figure 3
Traffic Flow Graph



REAL WORLD SYSTEMS MODELED BY GRAPHS

- Any type of transportation system can be modeled using a graph.
- For example, an airline can model their flight system using a graph.
- Each airport is a vertex and each flight from one vertex to another is an edge.
- A weighted edge can represent the cost of a flight from one airport to another, or perhaps the distance from one airport to another, depending on what is being modeled.

Figure 4
An Airline Graph



THE GRAPH CLASS

- At first glance, a graph looks much like a tree, and you might be tempted to try to build a graph class like a tree.
- There are problems with using a reference based implementation, however, so we will look at a different scheme for representing both vertices and edges.

Representing Vertices

- The first step we have to take to build a Graph class is to build a Vertex class to store the vertices of a graph.
- This class has the same duties the Node class had in the LinkedList and BinarySearchTree classes.
- The Vertex class needs two data members: one for the data that identifies the vertex, and the other a Boolean member we use to keep track of "visits" to the vertex. We call these data members label and wasVisited, respectively.

Representing Vertices

- The only method we need for the class is a constructor method that allows us to set the label and wasVisited data members.
- We won't use a default constructor in this implementation because every time we make a first reference to a vertex object, we will be performing instantiation.

Code For The Vertex Class

```
public class Vertex
{
    public bool wasVisited;
    public string label;

    public Vertex(string label) {
            this.label = label;
            wasVisited = false;
            }
 }
```

We will store the list of vertices in an array and will reference them in the Graph class by their position in the array.

Representing Edges

- The real information about a graph is stored in the edges, since the edges detail the structure of the graph.
- As we mentioned earlier, it is tempting to represent a graph like a binary tree, but doing so would be a mistake.
- A binary tree has a fairly fixed representation, since a parent node can only have two child nodes, whereas the structure of a graph is much more flexible.
- There can be many edges linked to a single vertex or just one edge, for example.

Representing Edges

- The method we'll choose for representing the edges of a graph is called an adjacency matrix.
- This is a two-dimensional array where the elements indicate whether an edge exists between two vertices.
- Figure 5 illustrates how an adjacency matrix works for the graph in the figure.

Representing Edges

- The vertices are listed as the headings for the rows and columns.
- If an edge exists between two vertices, a 1 is placed in that position.
- If an edge doesn't exist, a 0 is used. Obviously, you can also use Boolean values here.

	V_0	V_1	V_2	V_3	V_4
V_0	0	0	1	0	0
V_1	0	0	1	0	0
V_2	1	1	0	1	1
V_3	0	0	1	0	0
V_4	0	0	1	0	0

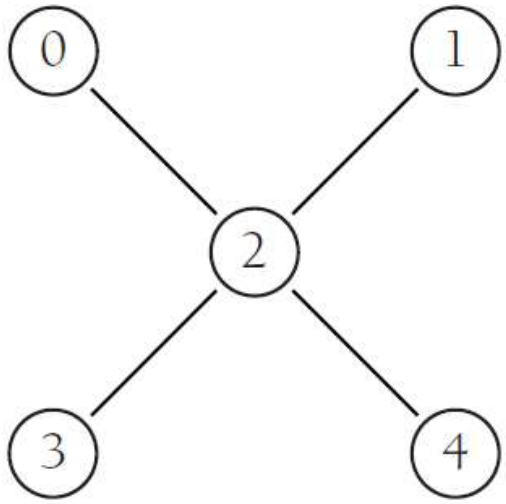


Figure 5 An Adjacency Matrix

Building a Graph

- Now that we have a way to represent vertices and edges, we're ready to build a graph.
- First, we need to build a list of the vertices in the graph.
- Here is some code for a small graph that consists of four vertices:

```
int nVertices = 0;
vertices[nVertices] = new Vertex("A");
nVertices++;
vertices[nVertices] = new Vertex("B");
nVertices++;
vertices[nVertices] = new Vertex("C");
nVertices++;
vertices[nVertices] = new Vertex("D");
```

Building a Graph

- Then we need to add the edges that connect the vertices.
- Here is the code for adding two edges:

```
adjMatrix[0,1] = 1;
adjMatrix[1,0] = 1;
adjMatrix[1,3] = 1;
adjMatrix[3,1] = 1;
```

- This code states that an edge exists between vertices A and B and that an edge exists between vertices B and D.

Building a Graph

- With these pieces in place, we're ready to look at a preliminary definition of the Graph class (along with the definition of the Vertex class):

```
public class Vertex {

  public bool wasVisited;
  public string label;

  public Vertex(string label) {
    this.label = label;
    wasVisited = false;
  }
}
```

```
public class Graph {

  private const int NUM_VERTICES = 20;
  private Vertex[] vertices;
  private int[,] adjMatrix;
  int numVerts;

  public Graph() {
    vertices = new Vertex[NUM_VERTICES];
    adjMatrix = new int[NUM_VERTICES, NUM_VERTICES];
    numVerts = 0;
    for(int j = 0; j <= NUM_VERTICES; j++)
      for(int k = 0; k <= NUMVERTICES-1; k++)
        adjMatrix[j,k] = 0;

  }
```

```
    public void AddVertex(string label) {
      vertices[numVerts] = new Vertex(label);
      numVerts++;
    }

    public void AddEdge(int start, int eend) {
      adjMatrix[start, eend] = 1;
      adjMatrix[eend, start] = 1;
    }

    public void ShowVertex(int v) {
      Console.Write(vertices[v].label + " ");
    }
}
```

- The constructor method redimensions the vertices array and the adjacency matrix to the number specified in the constant NUM VERTICES.
- The data member numVerts stores the current number in the vertex list so that it is initially set to zero, since arrays are zero-based.
- Finally, the adjacency matrix is initialized by setting all elements to zero.

- The AddVertex method takes a string argument for a vertex label, instantiates a new Vertex object and adds it to the vertices array.
- The AddEdge method takes two integer values as arguments.
- These integers represent to vertices and indicate that an edge exists between them.
- Finally, the showVertex method displays the label of a specified vertex.

A FIRST GRAPH APPLICATION: TOPOLOGICAL SORTING

- Topological sorting involves displaying the specific order in which a sequence of vertices must be followed in a directed graph.
- The sequence of courses a college student must take on their way to a degree can be modeled with a directed graph.
- A student can’t take the Data Structures course until they’ve had the first two introductory Computer Science courses, as an example.
- Figure 6 depicts a directed graph modeling part of the typical Computer Science curriculum.

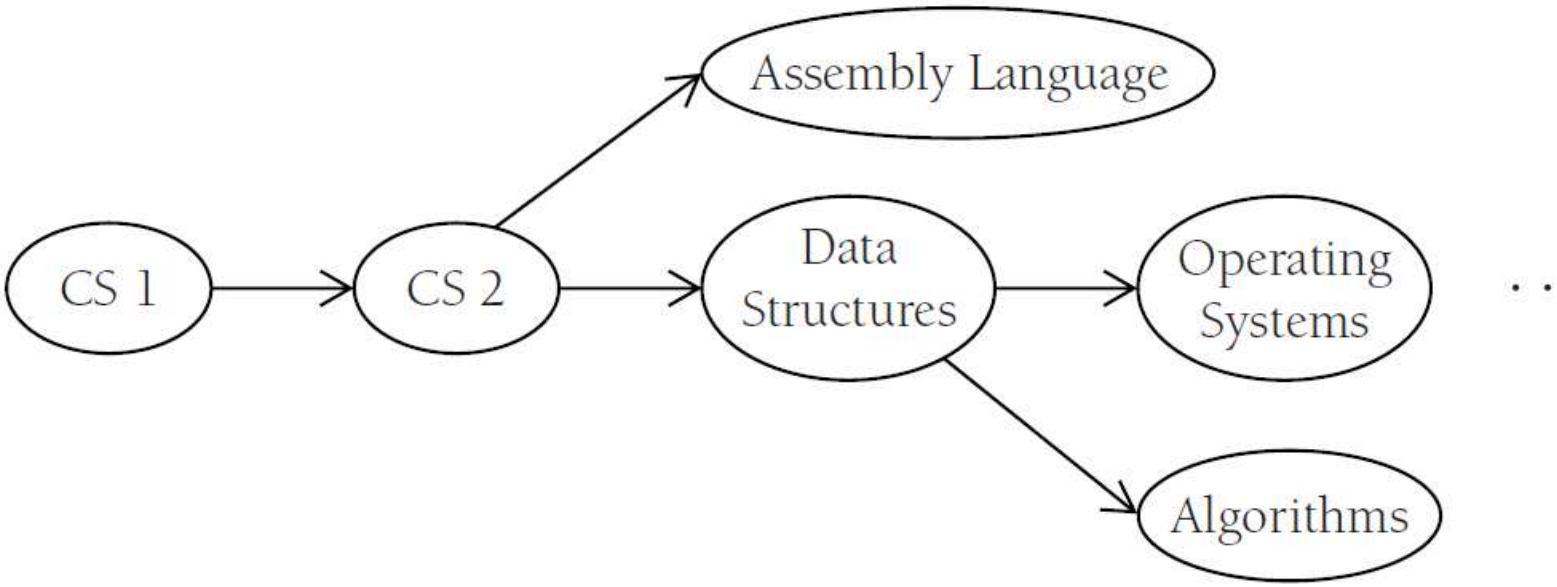


Figure 6 A Directed Graph Model of Computer Science Curriculum Sequence

A FIRST GRAPH APPLICATION: TOPOLOGICAL SORTING

- A topological sort of this graph would result in the following sequence:
  1. CS1
  2. CS2
  3. Assembly Language
  4. Data Structures
  5. Operating Systems
  6. Algorithms
- Courses 3 and 4 can be taken at the same time, as can 5 and 6.

An Algorithm for Topological Sorting

- The basic algorithm for topological sorting is very simple:
  1. Find a vertex that has no successors.
  2. Add the vertex to a list of vertices.
  3. Remove the vertex from the graph.
  4. Repeat Step 1 until all vertices are removed.

An Algorithm for Topological Sorting

- The algorithm will actually work from the end of the directed graph to the beginning.
- Look again at Figure 6. Assuming that Operating Systems and Algorithms are the last vertices in the graph (ignoring the ellipsis), neither of them have successors and so they are added to the list and removed from the graph.
- Next come Assembly Language and Data Structures. These vertices now have no successors and so they are removed from the list.
- Next will be CS2. Its successors have been removed so it is added to the list. Finally, we're left with CS1.

Implementing the Algorithm

- We need two methods for topological sorting—a method to determine if a vertex has no successors and a method for removing a vertex from a graph.
- Let's look at the method for determining no successors first.
- A vertex with no successors will be found in the adjacency matrix on a row where all the columns are zeroes.
- Our method will use nested for loops to check each set of columns row by row. If a 1 is found in a column, then the inner loop is exited, and the next row is tried.
- If a row is found with all zeroes in the columns, then that row number is returned. If both loops complete and no row number is returned, then a −1 is returned, indicating there is no vertex with no successors. Here's the code:

```
public int NoSuccessors() {
  bool isEdge;
  for(int row = 0; row <= numVertices-1; row++) {
    isEdge = false;
    for(int col = 0 col <= numVertices-1; col++)
      if (adjMatrix[row, col] > 0) {
        isEdge = true;
        break;
      }
    }
    if (!(isEdge))
      return row;
  }
  return -1;
}
```

Implementing the Algorithm

- Next we need to see how to remove a vertex from the graph.
- The first thing we have to do is remove the vertex from the vertex list. This is easy.
- Then we need to remove the row and column from the adjacency matrix, followed by moving the rows and columns above and to the right of the vertex are moved down and to the left to fill the void left by the removed vertex.
- To perform this operation, we write a method named delVertex, which includes two helper methods, moveRow and moveCol. Here is the code:

```
public void DelVertex(int vert)
  if (vert ! = numVertices-1) {
    for(int j = vert; j <= numVertices-1; j++)
      vertices[j] = vertices[j+1];
    for(int row = vert; row <= numVertices-1; row++)
      moveRow[row, numVertices];
    for(int col = vert; col <= numVertices-1; col++)
      moveCol[row, numVertices-1];
  }
}
```

```
private void MoveRow(int row, int length) {
  for(int col = 0; col <= length-1; col++)
    adjMatrix[row, col] = adjMatrix[row+1, col];
}

private void MoveCol(int col, int length) {
  for(int row = 0; row <= length-1; row++)
    adjMatrix[row, col] = adjMatrix[row, col+1];
}
```

- Now we need a method to control the sorting process.We'll show the code first and then explain what it does:

```
public void TopSort() {
  int origVerts = numVertices;
  while(numVertices > 0) {
    int currVertex = noSuccessors();
    if (currVertex == -1) {
      Console.WriteLine("Error: graph has cycles.");
      return;
    gStack.Push(vertices[currVertex].label);
    DelVertex(currVertex);
  }
  Console.Write("Topological sorting order: ");
  while (gStack.Count > 0)
    Console.Write(gStack.Pop() + " ");
}
```

- The TopSort method loops through the vertices of the graph, finding a vertex with no successors, deleting it, and then moving on to the next vertex.
- Each time a vertex is deleted, its label is pushed onto a stack.
- A stack is a convenient data structure to use because the first vertex found is actually the last (or one of the last) vertices in the graph. When the TopSort method is complete, the contents of the stack will have the last vertex pushed down to the bottom of the stack and the first vertex of the graph at the top of the stack.
- We merely have to loop through the stack popping each element to display the correct topological order of the graph.

- These are all the methods we need to perform topological sorting on a directed graph.
- Here's a program that tests our implementation:
- The output from this program shows that the order of the graph is A B C D.

```
static void Main() {
  Graph theGraph = new Graph();
  theGraph.AddVertex("A");
  theGraph.AddVertex("B");
  theGraph.AddVertex("C");
  theGraph.AddVertex("D");
  theGraph.AddEdge(0, 1);
  theGraph.AddEdge(1, 2);
  theGraph.AddEdge(2, 3);
  theGraph.AddEdge(3, 4);
  theGraph.TopSort();
  Console.WriteLine();
  Console.WriteLine("Finished.");
}
```

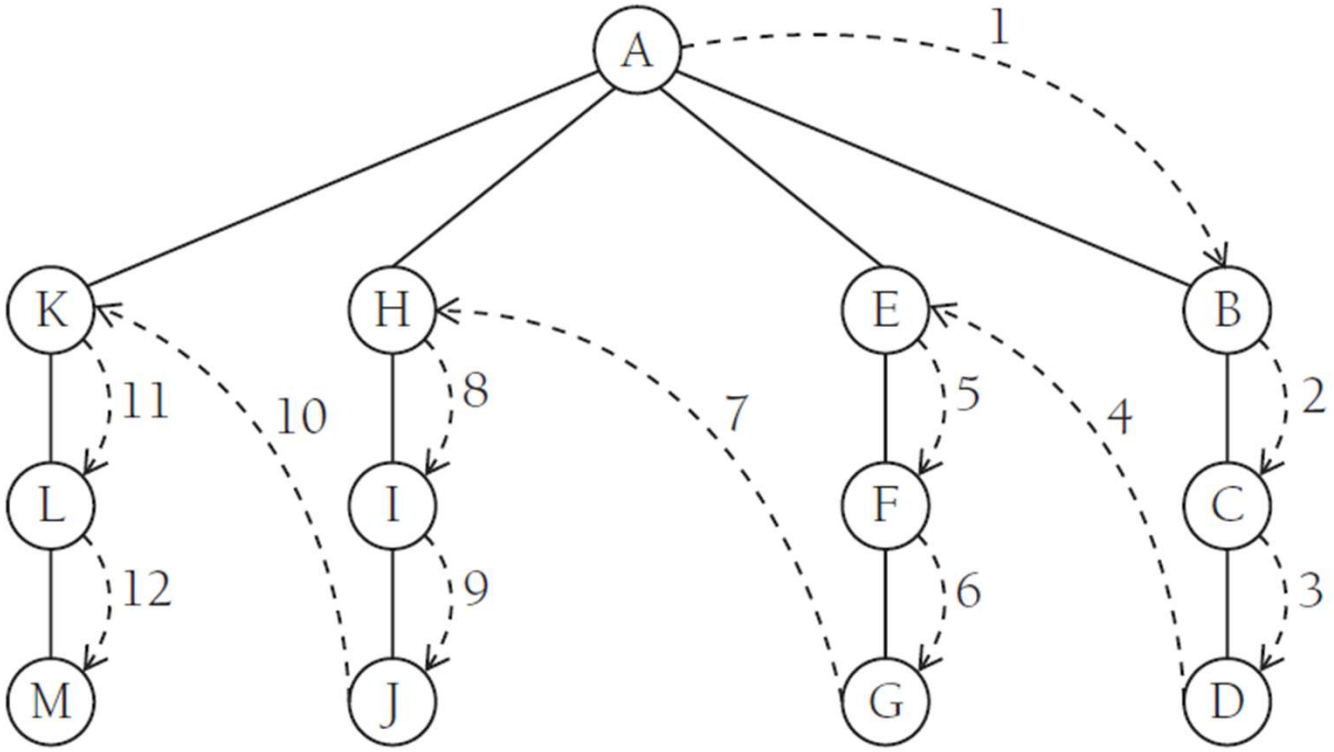
SEARCHING A GRAPH

- Determining which vertices can be reached from a specified vertex is a common activity performed on graphs.
- We might want to know which roads lead from one town to other towns on the map, or which flights can take us from one airport to other airports.
- These operations are performed on a graph using a search algorithm.
- There are two fundamental searches we can perform on a graph: a depth-first search and a breadth-first search.

Depth-First Search

- Depth-first search involves following a path from the beginning vertex until it reaches the last vertex, then backtracking and following the next path until it reaches the last vertex, and so on until there are no more paths left.
- A diagram of a depth-first search is shown in Figure 7.

Figure 7
Depth-First Search



Depth-First Search

- At a high level, the depth-first search algorithm works like this:
- First, pick a starting point, which can be any vertex. Visit the vertex, push it onto a stack, and mark it as visited.
- Then you go to the next vertex that is unvisited, push it on the stack and mark it. This continues until you reach the last vertex.

Depth-First Search

- Then you check to see if the top vertex has any unvisited adjacent vertices.
- If it doesn't, then you pop it off the stack and check the next vertex. If you find one, you start visiting adjacent vertices until there are no more,
- check for more unvisited adjacent vertices and continue the process.
- When you finally reach the last vertex on the stack and there are no more adjacent, unvisited vertices, you've performed a depth-first search.

Depth-First Search Implementation

- The first piece of code we have to develop is a method for getting an unvisited, adjacent matrix.
- Our code must first go to the row for the specified vertex and determine if the value 1 is stored in one of the columns. If so, then an adjacent vertex exists.
- We can then easily check to see if the vertex has been visited or not. Here's the code for this method:

Depth-First Search Implementation

```
private int GetAdjUnvisitedVertex(int v) {
  for(int j = 0; j <= numVertices-1; j++)
    if ((adjMatrix(v,j) = 1) && (vertices[j].WasVisited_
                                 == false))
      return j;
  return -1;
}
```

Now we're ready to look at the method that performs the depth-first search:

```
public void DepthFirstSearch() {
  vertices[0].WasVisited = true;
  ShowVertex(0);
  gStack.Push(0);
  int v;
  while (gStack.Count > 0) {
    v = GetAdjUnvisitedVertex(gStack.Peek());
    if (v == -1)
      gStack.Pop();
    else {
      vertices[v].WasVisited = true;
      ShowVertex(v);
      gStack.Push(v);
    }
  }
  for(int j = 0; j <= numVertices-1; j++)
    vertices[j].WasVisited = false;
}
```

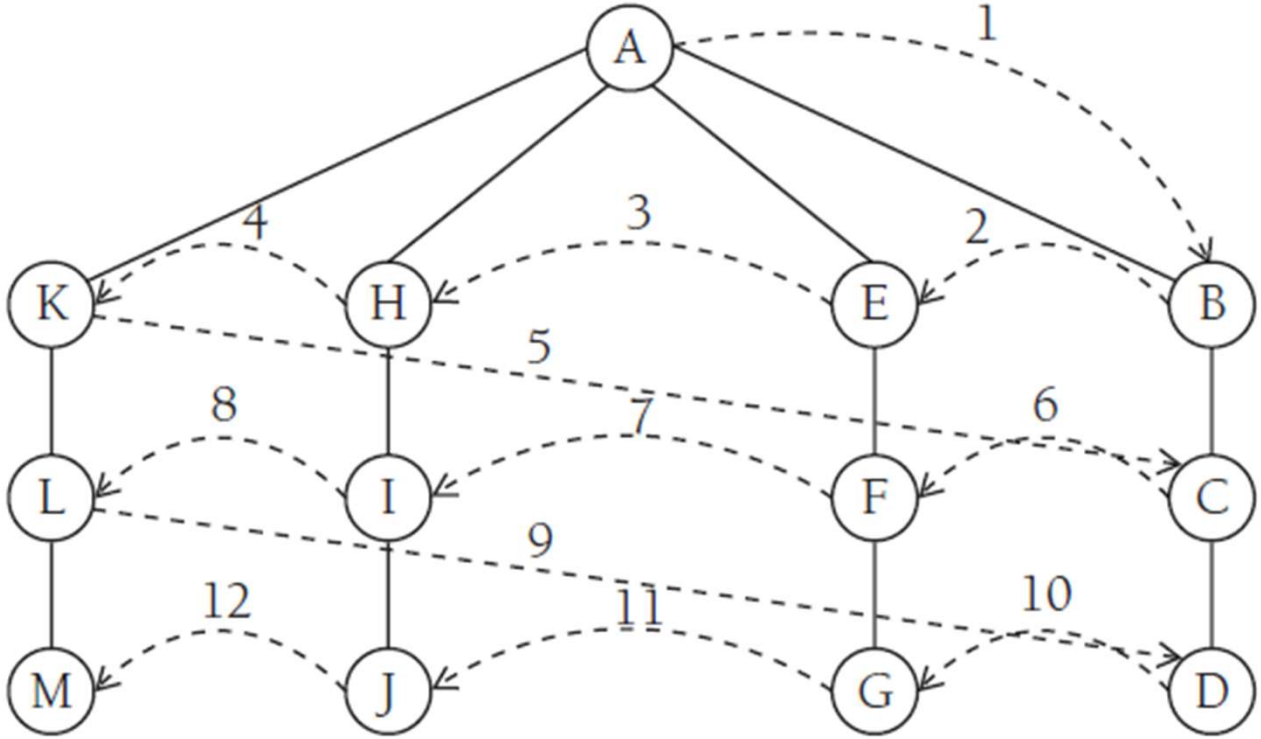
Here is a program that performs a depth-first search on the graph shown
in Figure 7.

```
static void Main() {
  Graph aGraph = new Graph();
  aGraph.AddVertex("A");
  aGraph.AddVertex("B");
  aGraph.AddVertex("C");
  aGraph.AddVertex("D");
  aGraph.AddVertex("E");
  aGraph.AddVertex("F");
  aGraph.AddVertex("G");
  aGraph.AddVertex("H");
  aGraph.AddVertex("I");
  aGraph.AddVertex("J");
  aGraph.AddVertex("K");
  aGraph.AddVertex("L");
  aGraph.AddVertex("M");
    aGraph.AddEdge(0, 1);
    aGraph.AddEdge(1, 2);
    aGraph.AddEdge(2, 3);
    aGraph.AddEdge(0, 4);
    aGraph.AddEdge(4, 5);
    aGraph.AddEdge(5, 6);
    aGraph.AddEdge(0, 7);
    aGraph.AddEdge(7, 8);
    aGraph.AddEdge(8, 9);
    aGraph.AddEdge(0, 10);
    aGraph.AddEdge(10, 11);
    aGraph.AddEdge(11, 12);
    aGraph.DepthFirstSearch();
    Console.WriteLine();
  }
```

Breadth-First Search

- A breadth-first search starts at a first vertex and tries to visit vertices as close To the first vertex as possible. In essence, this search moves through a graph layer by layer, examining the layers closer to the first vertex first and moving down to the layers farthest away from the starting vertex.
- Figure 8 demonstrates how breadth-first search works.

Figure 8
Breadth-First Search



Breadth-First Search Algorithm

- The algorithm for breadth-first search uses a queue instead of a stack, though a stack could be used. The algorithm is as follows:

1. Find an unvisited vertex that is adjacent to the current vertex, mark it as visited, and add to a queue.
2. If an unvisited, adjacent vertex can't be found, remove a vertex from the queue (as long as there is a vertex to remove), make it the current vertex, and start over.
3. If the second step can't be performed because the queue is empty, the algorithm is finished.

Now let's look at the code for the algorithm:

```
public void BreadthFirstSearch() {
  Queue gQueue = new Queue();
  vertices[0].WasVisited = true;
  ShowVertex(0);
  gQueue.EnQueue(0);
  int vert1, vert2;
  while (gQueue.Count > 0) {
    vert1 = gQueue.Dequeue();
    vert2 = GetAdjUnvisitedVertex(vert1);
    while (vert2 ! = -1) {
      vertices[vert2].WasVisited = true;
      ShowVertex(vert2);
      gQueue.Enqueue(vert2);
      vert2 = GetAdjUnvisitedVertex(vert1);
    }
  }
  for(int i = 0; i <= numVertices-1; i++)
    vertices[index].WasVisited = false;
}
```

- Notice that there are two loops in this method .
- The outer loop runs while the queue has data in it, and the inner loop checks adjacent vertices to see if they've been visited. The for loop simply cleans up the vertices array for other methods.
- A program that tests this code, using the graph from Figure 8, is shown as follows:

```
static void Main() {
  Graph aGraph = new Graph();
  aGraph.AddVertex("A");
  aGraph.AddVertex("B");
  aGraph.AddVertex("C");
  aGraph.AddVertex("D");
  aGraph.AddVertex("E");
  aGraph.AddVertex("F");
  aGraph.AddVertex("G");
  aGraph.AddVertex("H");
  aGraph.AddVertex("I");
  aGraph.AddVertex("J");
  aGraph.AddVertex("K");
  aGraph.AddVertex("L");
  aGraph.AddVertex("M");
  aGraph.AddEdge(0, 1);
  aGraph.AddEdge(1, 2);
  aGraph.AddEdge(2, 3);
  aGraph.AddEdge(0, 4);
  aGraph.AddEdge(4, 5);
  aGraph.AddEdge(5, 6);
  aGraph.AddEdge(0, 7);
  aGraph.AddEdge(7, 8);
  aGraph.AddEdge(8, 9);
  aGraph.AddEdge(0, 10);
  aGraph.AddEdge(10, 11);
  aGraph.AddEdge(11, 12);
  Console.WriteLine();
  aGraph.BreadthFirstSearch();
}
```

Answer the following questions

- Compare directed graphs and undirected graphs
- Compare DFS (Depth-First Search) and BFS (Breadth-First Search)

See You Next Lecture