# CIS7A Unit 10 Chapter 16 Notes: Graphs

We use **graphs** to ***model networks such as computer***, airline, phone, or social networks. They can also be used to model such diverse things as connections between data in a database or molecular structure.

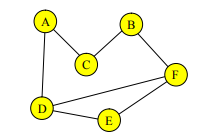
## Simple Graphs

A simple graph **G = (V, E)** is a ***set of vertices V******connected by edges E.***

An edge in a **simple graph** is just an ***unordered pair of vertices***, i.e. a ***set containing two vertices***. The ***two vertices of an edge*** are said to be **adjacent**. We talk about going along an edge to get to one vertex from another. In a simple graph, each edge is like a two-way street. When an edge e connects two vertices u and v, we write e = {u, v} or e = {v, u}2. **An edge cannot go from a vertex to itself**.

### Example 16.1:

This graph has six vertices and seven edges



V = {A, B, C, D, E, F}

E = {{A, C}, {A, D}{D, F}, {D, E}, {C, B}, {B, F}, {F, E}}

Vertices D and F are adjacent.

Vertices C and E are not adjacent.

### Example 16.2:

In the graph of example 16.1 vertices ***A, B, C, and E have degree 2***. ***Vertices D and F have degree 3***.

We say an ***edge e = {v, u} is incident to the vertices u and v***. The **degree**, **deg(v)** of a vertex is the ***number of edges that are incident to it***.

If u and v are vertices in a graph G = (V, E), a path of length n from u to v is a sequence of vertices hv0, v1, · · · , v ni in V such that u = v0, v = v n, and such that {v k, v k+1} ∈ E for **0 ≤ k ≤ n − 1**. That is, each successive pair of vertices is connected by an edge. We say that the path contains the vertices v0, v1, · · ·, v n and the edges {vk, v k+1}, 0 ≤ k ≤ n − 1. Note that ***the length of a path is the number of edges in the path***. The path (u) is a path of ***length 0 from u to itself***. A vertex v is reachable from a vertex u if there is a path from u to v. We also say that u and v are path connected if there is a path from u to v. The connected component of a vertex u is the set of all vertices v such that v is reachable from u.

### Example 16.3:

Here are some paths in the graph of example 16.1:

(A, C, B, F) is a path from vertex A to vertex F.

(F, E, D) is a path from vertex F to vertex D.

(D, E, Fi)is a path from vertex D to vertex F.

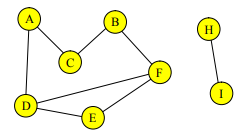
(D, F) is also a path from vertex D to vertex F.

(D, A, C, B, F) is another path from vertex D to vertex F.

(D, A, C, B, F, D, E, F, D, E, F) is a **long path** from vertex D to vertex F.

### Example 16.4

This graph has eight vertices and eight edges.



V = {A, B, C, D, E, F, H, I}

E = {{A, C}, {A, D}, {D, F}, {D, E}, {C, B}, {B, F}, {F, E}, {H, I}}

All the paths in example 16.3 are paths in this graph too.

(HI) is a path from H to I.

There is no path from vertex D to vertex I. Vertex I is not reachable from the vertex D.

The vertices A, B, C, D, E, and F are all path connected to each other.

The connected component of C is {A, B, C, D, E, F}.

The connected component of His {H, I}.

A **simple path** is a ***path that has no repeated vertices***. If you follow a simple path, you ***will never go through that same vertex twice***.

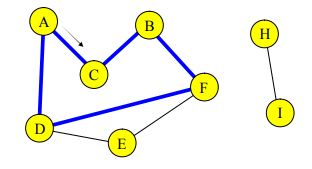
### Example 16.5

The paths (D, E, F), (D, F), and (D, A, C, B, F) are all **simple paths** ***from vertex D to vertex F*** in the graph of example 16.4 . The path (D, A, C, B, F, D, E, F, D, E, F) **is not a simple path** ***from vertex D to vertex F*** as it ***goes through each of the vertices D, E, and F*** multiple times.

A **cycle** is a path (v0, v1, · · · , vn) such that the vertices (v1, · · · , vn) are distinct and v0 = vn. A ***cycle is just a closed path or loop that has no repeated vertices except for ending up where it started***.

### Example 16.6

This is the graph of example 16.4.



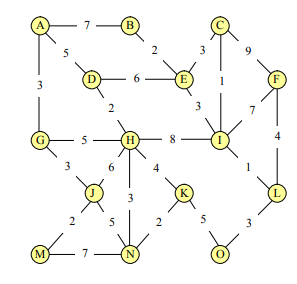
The ***path (A, C, B, F, D, A) is shown in blue***. The path starts at vertex A, and proceeds in the direction of the arrow. These ***paths (D, E, F, D), and (F, E, D, F) are also cycles*** in the graph.

The ***paths (D, E, F, D, E, F, D) and (D, E, F, D, A) are not cycles***.

## Weighted Graphs

In the graphs we have looked at so far, ***all edges were treated as equal***. If the graph is a model of a highway system with the edges representing roads and the vertices representing cities, then each edge will have a distance associated with it. If a graph represents a computer network, each edge might have a bandwidth. For an airline network, the edges would have prices of tickets. A weighted graph G = (V, E, w) is a simple graph with **a weight** ***associated with each edge that is given by a function*** **w : E → R.** When we draw a weighted graph, ***we put the weights on or next to the edges***. In the graph below, w({A, B}) = 7 and w({H, N}) = 3. The ***weight of a path is the sum of the weights of its edges***.

### Example 16.7



In this weighted graph, the edge {H, N} has weight 3.

The weight of the path (A, D, H, K, O) = 5 + 2 + 4 + 5 = 16.

The paths (E, D, H, K) and (E, C, F, L, O, K) are both paths from E to K. The weight of

(E, D, H, K) = 6 + 2 + 4 = 12 and the weight of (E, C, F, L, O, K) = 3 + 9 + 4 + 3 + 5 = 24.

## Graph Data Structures

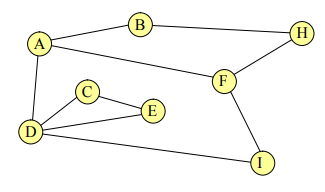
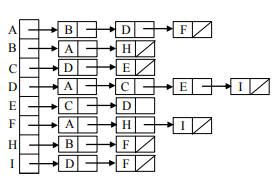
When you look for a flight on Travelocity, their computer must find a path between the cities you name. When it finds the cheapest flight, it is actually finding the path with the lowest weight. Before we can write programs to do these searches, we must have a way of representing graphs that our programs can understand. We could just keep a ***list of the vertices and a list of the edges***, but it is time consuming to search through such lists to find paths. There are ***two standard representations of graphs*** that are used in computer programs, **adjacency lists** and **adjacency matrices**.

## Adjacency Lists

The **adjacency list** of a graph G = (V, E) ***is an array of the vertices and a list, for each vertex, of the vertices adjacent to it***, i.e all the vertices u ∈ V such that {v, u} ∈ E. With an adjacency list, ***it is easy to find all the vertices that are adjacent to a given vertex*** which is the same as finding all the edges incident to that vertex.

### Example 16.8

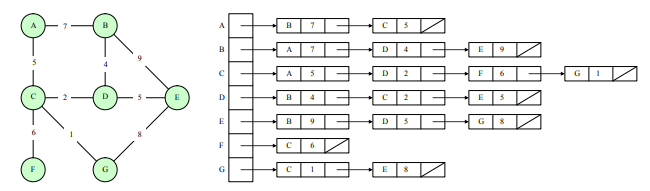
Here is a simple graph and its adjacency list.

For a weighted graph, we need a way to store the weights as well as the vertices and edges. We store these ***weights in the adjacency lists right next to the names*** ***of the adjacent vertices***.

### Example 16.9

Here is a weighted graph and its adjacency list:



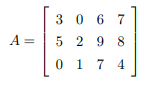
## Matrices

A **matrix** is just ***a two-dimensional array of numbers***. Matrices are very important in mathematics as they provide a concise and algebraically useful way of ***representing linear functions on n-dimensional spaces***. In computer science, you will see them used for such diverse things as solving systems of linear equations and representing rotations in computer graphics. Here, we use them to represent simple graphs.

Formally, an **n × m** matrix A has **n rows and m columns**. We write A = (aij ) where (aij ) is the number in row i and column j. The numbers (aij ) are called the elements of A.

### Example 16.10:

The matrix A below is a 3 × 4 matrix. a13 = 6, a31 = 0, a24 = 8, and there is no a42.

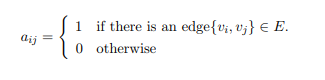


## Adjacency Matrix

Given the simple graph G = (V, E), assume V = {v1, v2, · · · , vn} where n = |V |. Recall that

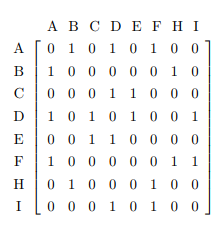
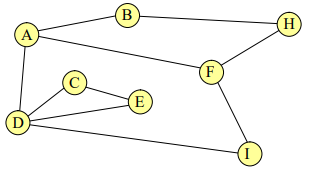
|V | is the cardinality of V which is just the number of vertices in the graph. The adjacency

matrix of G = (V, E) is a |V | × |V | matrix, adj = (aij ) where

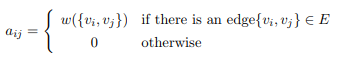


### Example 16.11:

Here is the same simple graph as in example 16.8 , this time with its adjacency matrix.

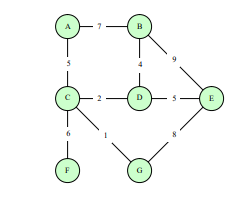
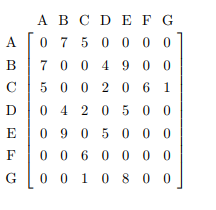


Notice that the elements of adj along the main diagonal, aii, i = 1 . . . n, are all 0. That’s because there are no edges from a vertex to itself. Also notice that the matrix is symmetric about its main diagonal, aij = aji for i, j = 1 . . . n. That is because any edge {u, v} is and edge from u to v and an edge from v to u. For a weighted graph, we put the weight of the edge in the matrix instead of a 1. The adjacency matrix of a weighted graph G = (V, E, w) is a |V | × |V | matrix, adj = (aij ) where



### Example 16.12:

Here is the weighted graph of example 16.9 with its adjacency matrix.

## Graph Traversal:

To traverse a graph, you must visit all the vertices of the graph by following edges. We wantnto be able to do this in a systematic and efficient way. A solution to this problem would allow an airline inspector to fly to every airport a company flies to, a subway aficionado to take the train to every stop in a city, or a network administrator to send a message to every computer on a network.

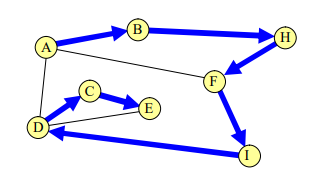
Can you start at any vertex and visit all the other vertices by following edges? Not in general; in the graph of example 16.4 you cannot start at vertex A and visit vertex H. ***You can only visit the vertices in the connected component of the starting vertex***. To visit all the vertices, every vertex must be reachable from every other vertex. In this case, we say the graph is connected. We will look at two methods for traversing a connected graph. Both of these search methods can be used ***for traversals or for finding paths between two vertices***.

## Depth First Search

The idea of **depth first search** is to ***move forward from the starting vertex as far as you can go without repeating a vertex, then backup one edge and look for another vertex to visit***, again using a depth first search. As ***you can see the process is recursive***.

### Example 16.13:

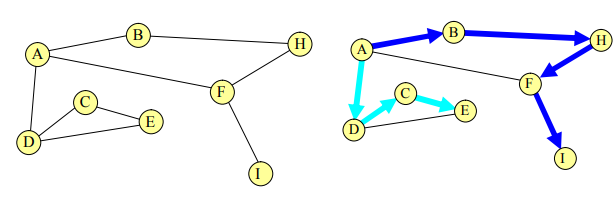
We’ll show a **depth first traversal** of the graph of example 16.8 and ***use its adjacency list to guide the search***. Note that the vertices in the adjacency lists are in alphabetical order so it is easy to remember which comes next.



This ***traversal starts at vertex A and proceeds to the first vertex on A’s adjacency list B*** and ***then to the only vertex on B’s list H***. The ***first vertex on H’s list is B which has already been visited so the traversal proceeds to the next vertex in H’s list, F, then I, then D***. The first vertex on D’s list is A which has already been visited so the traversal goes on to C and then to E. There is no place left to go from E so the traversal is complete. The vertices are visited in this order: ***A, B, H, F, I, D, C, E***.

### Example 16.15:

A depth first traversal starting from A is shown on the right. The vertices are visited in this order: A, B, H, F, I, C, D

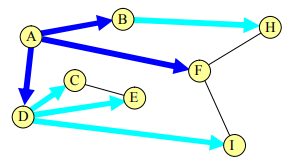


## Breadth First Search

In **breadth first search** you ***visit all the vertices adjacent to the starting vertex*** and ***then do a breadth first search from each of those vertices***.

### Example 16.16

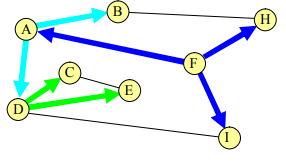
Here is a breadth first traversal of the graph of example 16.8 using its adjacency list to guide the search.



Starting from A, we first visit all of A’s neighbors in the order they appear on A’s adjacency list, i.e. B, D, F. We then visit the neighbors of B that have not yet been visited; that’s just vertex H. Then the unvisited neighbors of D, in the order they appear on D’s list, C, E, I. At this point, all the vertices have been visited and the traversal is complete. The vertices are visited in this order: A, B, D, F, H, C, E, I.

### Example 16.17

This figure shows a breadth first traversal starting from vertex F.



After F, we visit F’s neighbors, A, H, I, then the unvisited neighbors of A, i.e. B and D.

At that point all the neighbors of H and I are visited so we visit the remaining neighbors of

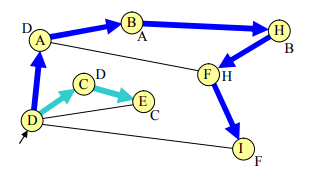
D, the vertices C and E. The vertices are visited in this order: F, A, H, I, B, D, C, E.

## Any Path

We start a depth first or breadth first search at one of our two vertices and follow the search until we reach the second vertex. Then we know a path exists but we won’t know what the path is. A slight modification of either search will give us the path too. Instead of just ***marking a vertex as having been visited, mark it with the name of the vertex you just came from***. This is like leaving a trail of bread crumbs. We can follow the trail back to reconstruct the path. If the graph is not connected, the search might never get to the second vertex. This is a case of, “You can’t get there from here.”

### Example 16.18

This is the graph and search of example 16.14 with labels added to the vertices to show how we got there. The arrow points to vertex D where the search started.



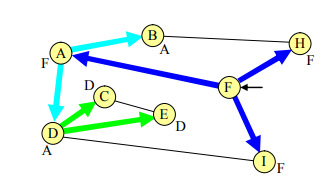
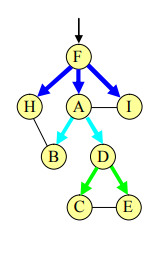
If we want a path from D to F, we start the search at D and when we arrive at F, we can read the labels to create a path,(F, H, B, A, D) from F to D, or writing in reverse, a path (D, A, B, H, F) from D to F.

## Shortest Path

The **shortest path** problem is to ***find the path with the smallest number of edges that connects two given vertices in a graph***. When you ask Orbitz for a flight between Madison, Wisconsin and Manhattan, Kansas with as few stops as possible, you are actually asking the Orbitz program to solve a shortest path problem. To find the shortest path in a graph from vertex u to vertex v, just do what we suggested for “any path” but use a breath first search. The path you find will be a shortest path.

### Example 16.19

On the left is the graph of example 16.8 showing a **breadth first search** from vertex F with labels added showing where you came from to get to each vertex. On the right is the same graph (without the labels) ***rearranged to show how far away each vertex is from the starting vertex***. Breadth first search first visits all the vertices you can get to by one edge, then all those you can get to by two edges but not by one edge, and so on. ***You always get to a vertex by a path with as few edges as possible***.

## Cheapest Path

Given a weighted graph, G = (V, E, w) and two vertices, u, v ∈ V , the cheapest path problem is to find a path connecting u to v that has the lowest weight among all paths connecting u and v. A cheapest path may also be a shortest path but this is not always true. When I buy an airline ticket, I generally look for the cheapest one I can find even if it means an extra stop, i.e. a longer route. The flight finder program must solve a cheapest path problem to offer me a route. The cheapest path problem is the same as the shortest path problem if all the weights are 1 so the two problems are often grouped together and just called the “shortest path problem.”

We have treated them separately because the “all the weights are 1” problem is pretty easy to solve just using breadth first search. ***Dijkstra’s Shortest Path Algorithm*** solves the more difficult problem with ***arbitrary weights***.

### Example 16.20

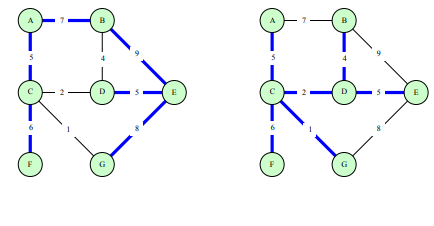
In the graph of example 16.9, the shortest path from A to E is (A, B, E) which has length 2 and weight 16. The cheapest path from A to E is (A, C, D, E) which has length 3 and weight 12.

## Spanning Tree

We look for a spanning tree of minimal weight to find a set of highways with minimal total distance that connect a group of major cities. Kruskal’s and Prim’s algorithms provide minimal spanning trees.

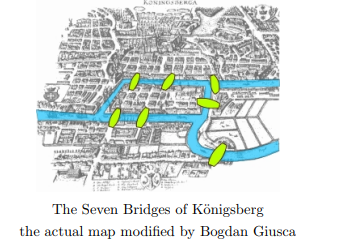
### Example 16.21

Each of the images below shows, in thickened edges, a spanning tree for the weighted graph of example 16.9 . The one on the left has weight 40 and the one on the right has weight 23.



## Graph Theory

***Graph Theory is a field of pure mathematics devoted to the study of graphs*** for their own sake. The theorems that come out of this theoretical study often have applications to computer science. ***Euler’s solution of the K¨oonigsberg bridge problem is considered to be the first theorem of graph theory***. It all started with seven real bridges in the city of K¨onigsberg.



Is it possible to take a walk that crosses each bridge exactly once and return to your starting point? In 1736, Leonhard Euler proved that is impossible. He used a graph with four vertices to represent the four land masses and seven edges to represent the seven bridges. He proved:

**Theorem 9:**  You can traverse a connected simple graph, following each edge exactly once and returning to the starting point, if and only if there are no vertices of odd degree. Recall that the degree of a vertex deg(v) is the number of edges incident to that vertex. The graph representing the Bridges of K¨onigsberg has one vertex of degree 5 and three of degree 3.

There are many theorems about graphs that involve the vertex degree. This theorem relates the vertex degrees to the number of edges in a graph.

**Theorem 10**: Given a simple graph G = (V, E), the sum of the degrees of the vertices is twice the number of edges.



***Proof***: Each edge contributes to the degree of two vertices that is it adds 2 to the sum on the left.

## Directed Graphs

**Directed graphs** are just the structure we need to model these problems. Much of our discussion of simple graphs applies to these graphs too with just minor changes.

A directed graph G = (V, E) is a set of vertices V connected by edges E. ***What distinguishes a directed graph from a simple graph is that the edges are one-way.*** An edge e ∈ E is an ordered pair of vertices, e = (u, v). The edge e = (u, v) goes from u to v.