

Module 11: Additional Topics

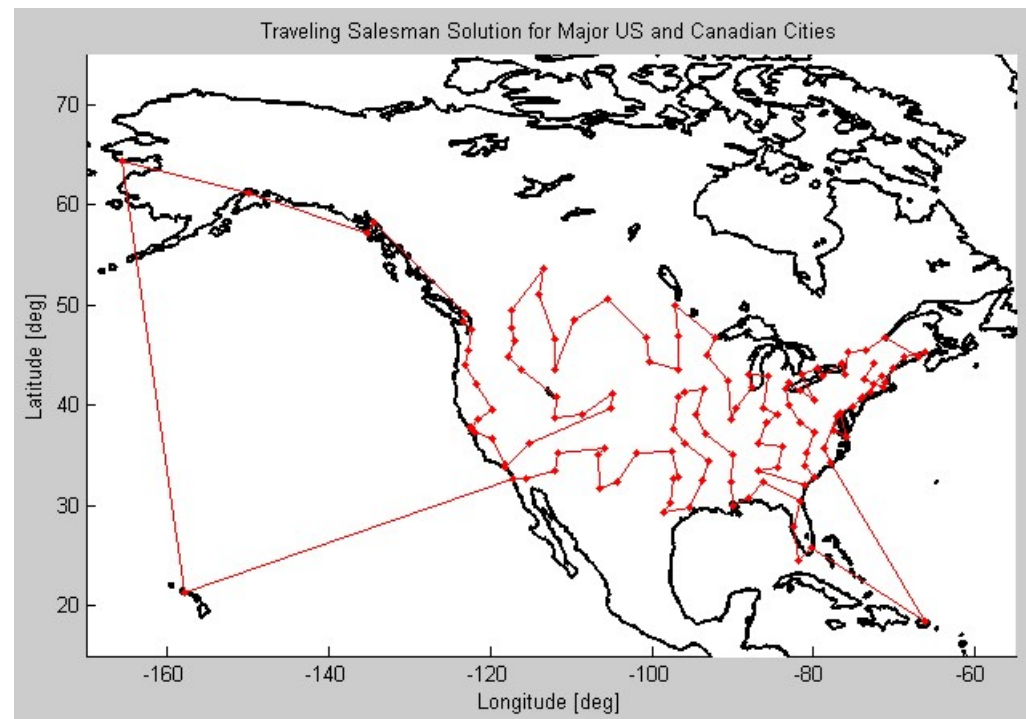
Graph Theory and Applications

Topics:

- Introduction to Graph Theory
- Representing (undirected) graphs
- Basic graph algorithms

Consider the following:

- Traveling Salesman Problem (TSP): Given N cities and the distances between them, find the shortest path to visit all cities and return to the start.

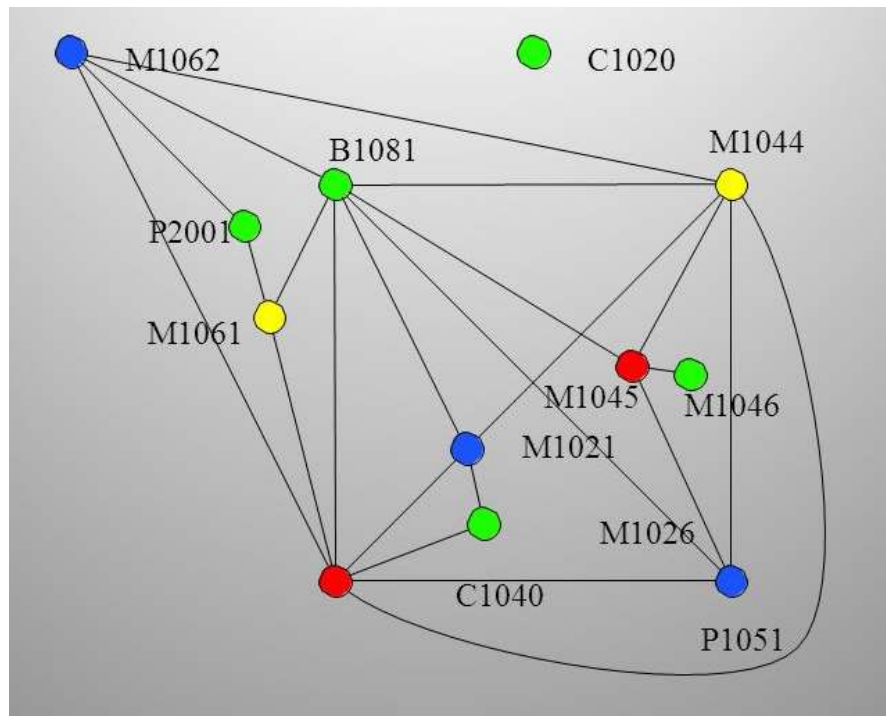


What does the TSP have in common with the following problems?

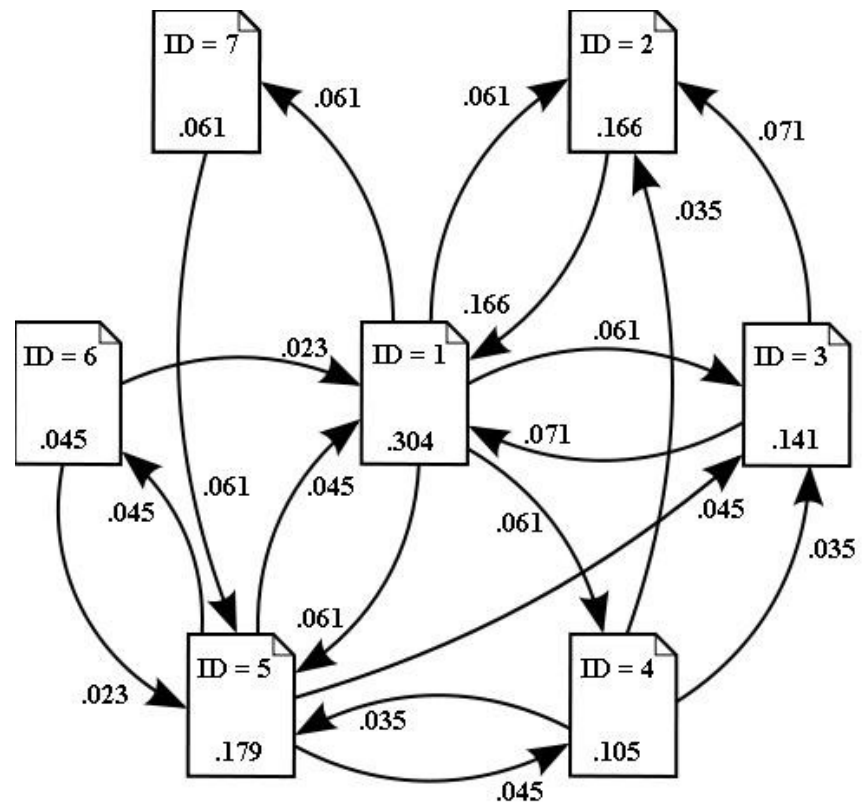
- Placement of new fire stations in a city to provide best coverage to all residents
- Ranking of "importance" of web pages by Google's PageRank algorithm
- Scheduling of final exams so they do not conflict
- Arranging components on a computer chip
- Analyzing strands of DNA
- Binary Search Trees

They all fall within the field of *GRAPH THEORY*

Non-conflicting exams

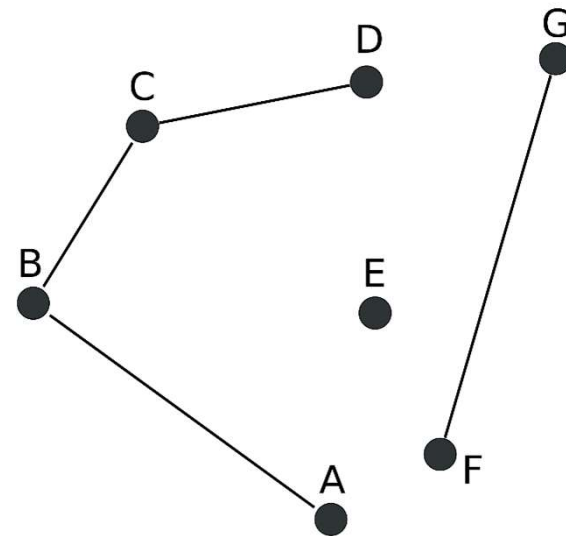
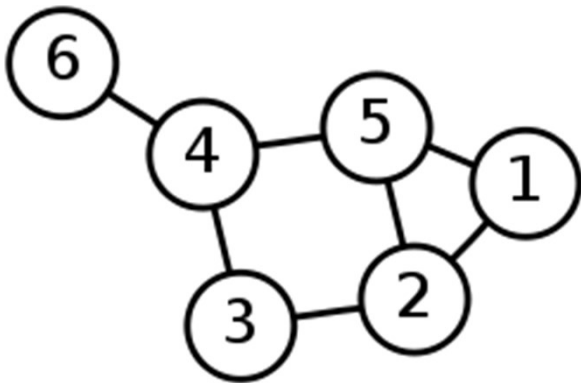


PageRank Algorithm



Undirected Simple Graphs

An undirected simple **graph** G is a set V , of **vertices**, and a set E , of unordered distinct pairs from V , called **edges**. We write $G=(V,E)$.



Graph Terminology

- If (v_k, v_p) is an edge, we say that v_k and v_p are **neighbours**, and are **adjacent**. Note that k and p must be different.
- The number of neighbours of a vertex is also called its **degree**
- A sequence of nodes v_1, v_2, \dots, v_k is a **path** of length $k-1$ if $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ are all edges
 - If $v_1 = v_k$, this is called a **cycle**
- A graph G is **connected** if there exists a path through all vertices in G

Interesting Results on Graphs

Let n = number of vertices,
and m = number of edges:

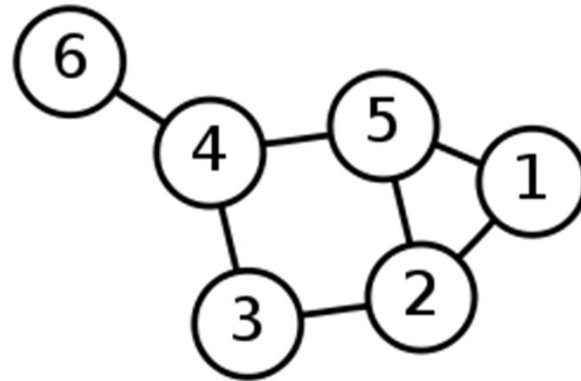
1. $m \leq n(n - 1)/2$
2. The number of graphs on n vertices is $2^{n(n-1)/2}$
3. The sum of the degrees over all vertices is $2m$.

How can we store information about graphs in Python?

- We need to store labels for the vertices
 - These could be strings or integers
- We need to store both endpoints using the labels on the vertices.
- We will consider three different implementations for undirected, unweighted graphs

Implementation 1: Vertex and Edge Lists

- $V = [v_1, v_2, v_3, \dots, v_m]$,
- $E = [e_1, e_2, e_3, \dots, e_m]$, where
edge $e_j = [a, b]$ when vertices a and b are
connected by an edge



$V = [6, 4, 5, 3, 2, 1]$

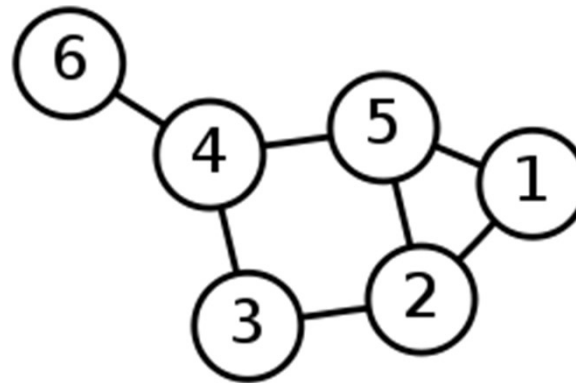
**$E = [[6, 4], [4, 5], [4, 3], [3, 2],$
 $[5, 2], [1, 2], [5, 1]]$**

Implementation 2: Adjacency list

- For each vertex:
 - Store the labels on its neighbours in a list
- We will use a dictionary
 - Keys: labels of vertices
 - Recall: integers or strings can be keys
 - Associated values: List of neighbours (adjacent vertices)

Example:

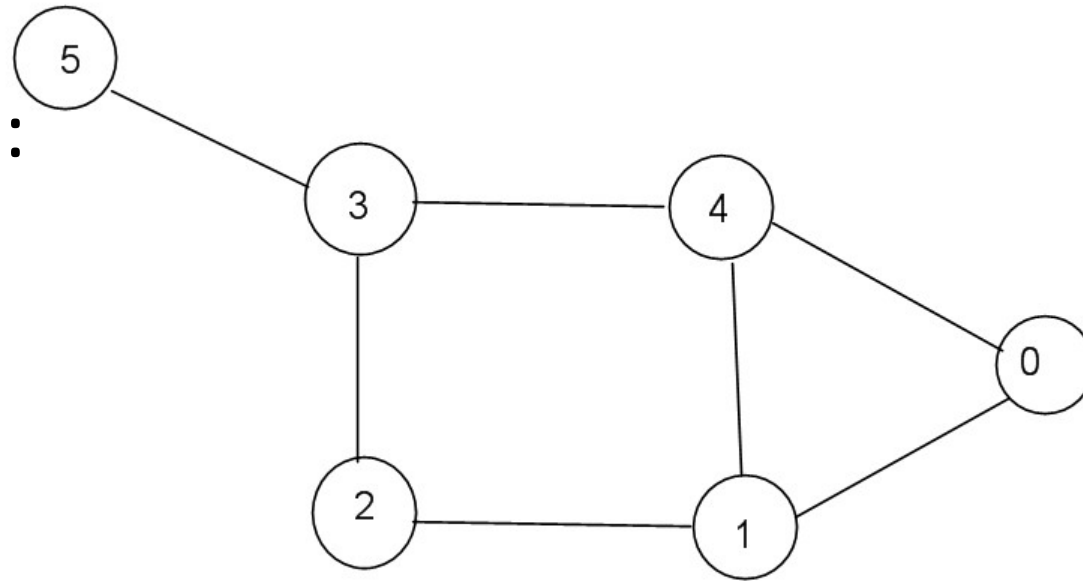
$\{1 : [2, 5],$
 $2 : [1, 3, 5],$
 $3 : [2, 4],$
 $4 : [3, 5, 6],$
 $5 : [1, 2, 4],$
 $6 : [4] \}$



Implementation 3: Adjacency Matrix

- For simplicity, assume vertices are labelled $0, \dots, n - 1$
- Create an $n \times n$ matrix for G
- If there is an edge connecting i and j :
 - Set $G[i][j] = 1$,
 - Set $G[j][i] = 1$
- Otherwise, set these values to 0

Example:



G:

vertex	0	1	2	3	4	5
0	[[0,	1,	0,	0,	1,	0],
1	[1,	0,	1,	0,	1,	0],
2	[0,	1,	0,	1,	0,	0],
3	[0,	0,	1,	0,	1,	1],
4	[1,	1,	0,	1,	0,	0],
5	[0,	0,	0,	1,	0,	0]]

Comparing the implementations on simple tasks

- Determine if two vertices are neighbours.
- Find all the neighbours of a vertex.

Which implementation to use?

- We'll use the adjacency list (a good case could also be made for the adjacency matrix).

Graph Traversals

- Determine all vertices of G that can be reached from a starting vertex
- There can be different types of traversals
- If you find all vertices starting from v , the graph is ***connected***
- If not all vertices can be reached, a ***connected component*** containing v has been found
- Must determine a way to ensure we do not cycle indefinitely

Applications of traversals

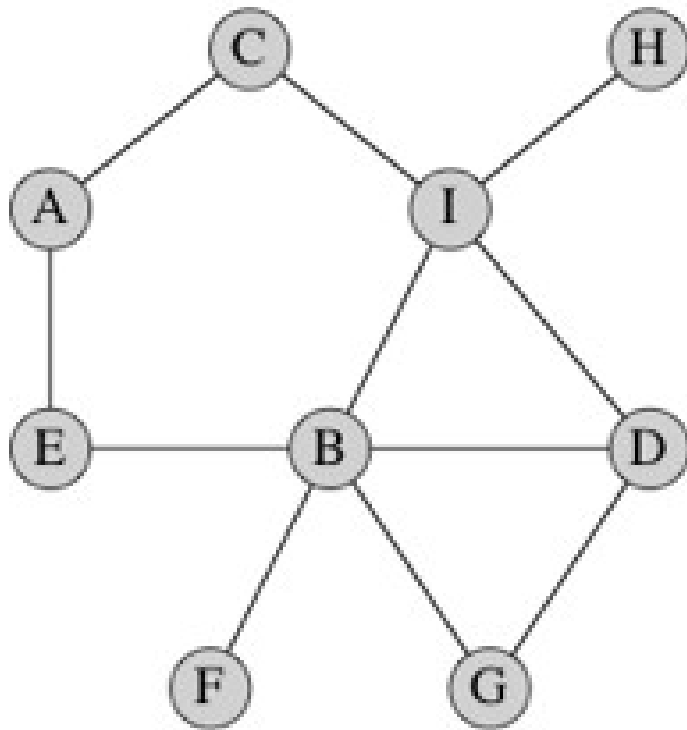
- Finding path between two vertices
- Finding connected components
- Tracing garbage collection in programs (managing memory)
- Shortest path between two points
- Planarity testing
- Solving puzzles like mazes
- Graph colouring

One approach:

Breadth-first search Traversal (bfs)

- Choose a starting point v
- Visit all the neighbours of v
- Then, visit all of the neighbours of the neighbours of v , etc.
- Repeat until all reachable vertices are visited
- Need some way to avoid visiting edges more than once
- Note: there may be more than one bfs ordering of a graph, starting from v .

Sample bfs orders



- A, C, E, B, I, F, G, D, H
- A, E, C, I, B, H, D, G, F
- B, E, F, G, D, I, A, H, C
- B, D, E, F, G, I, A, C, H
- H, I, C, B, D, A, E, F, G

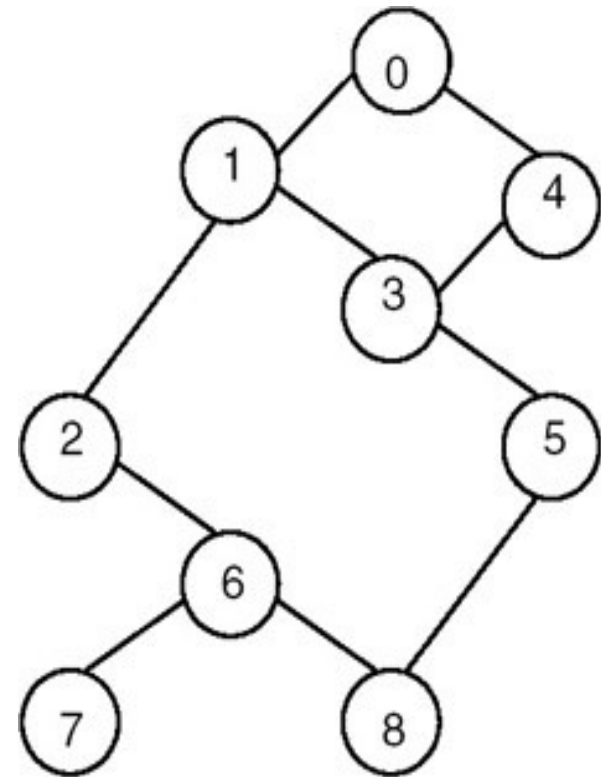
plus more ...

Implementing bfs

- We will look at one implementation.
- Assumes an adjacency list representation.
- We use several lists:
 - **a11** includes all "visited" vertices
 - Vertices are appended to the end
 - **Q** includes vertices waiting to be "visited" (it will grow and shrink as the algorithm progresses)
 - Vertices are appended to the end and removed from the front of **Q**

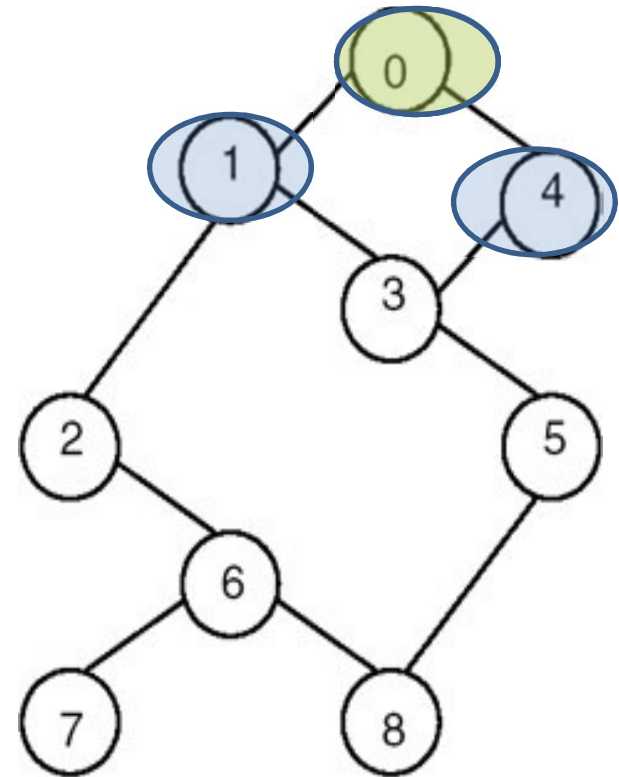
Implementation of bfs traversal

```
def bfs(graph, v):  
    all = []  
    Q = []  
    Q.append(v)  
    while Q != []:  
        v = Q.pop(0)  
        all.append(v)  
        for n in graph[v]:  
            if n not in Q and\  
                n not in all:  
                Q.append(n)  
    return all
```



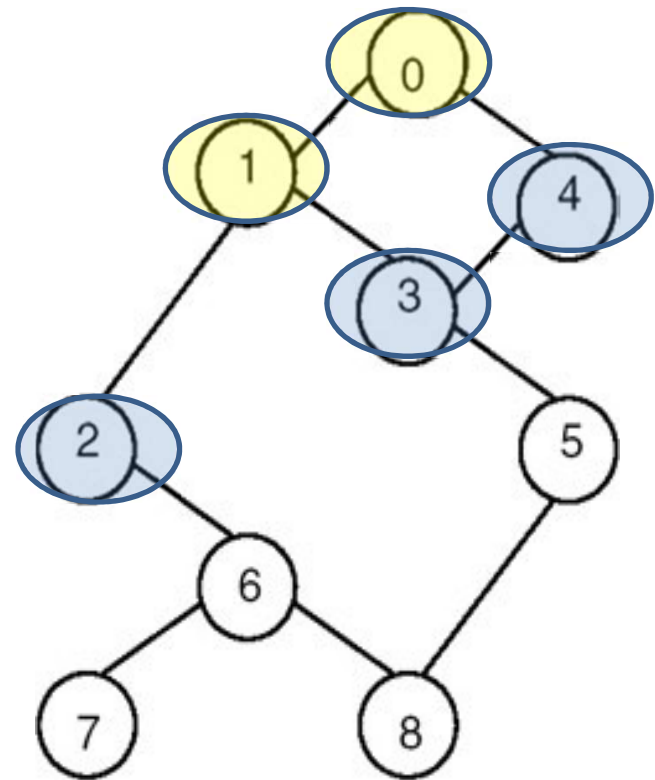
Starting bfs from 0 (1)

- Start from $v=0$
- $all = []$
- $Q = [0]$
 - $v = 0$
 - $all = [0]$
 - Neighbours of 0: 1,4
 - $Q = [1,4]$



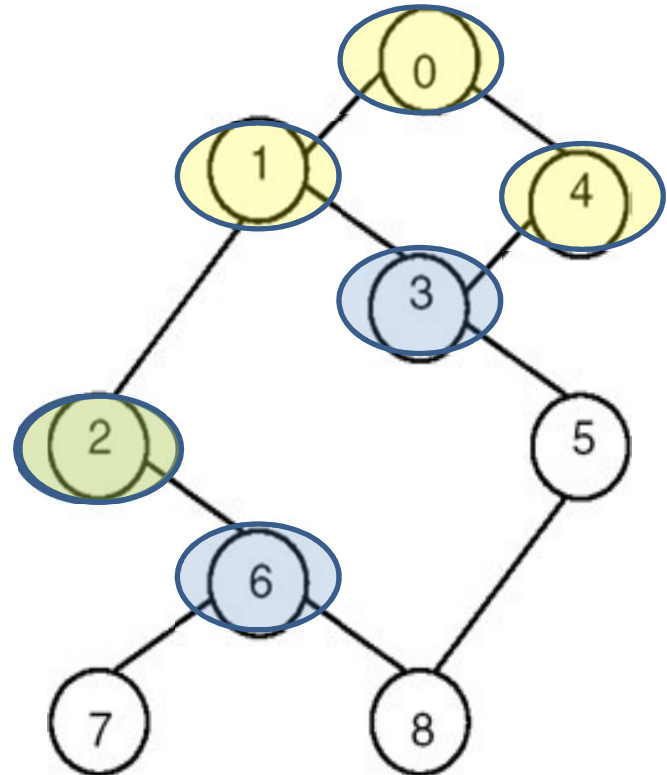
Continuing bfs (2)

- $Q = [1, 4]$
- $v = 1$
- $all = [0,1]$
- Neighbours of 1: 0,2,3
 - $Q = [4,2,3]$



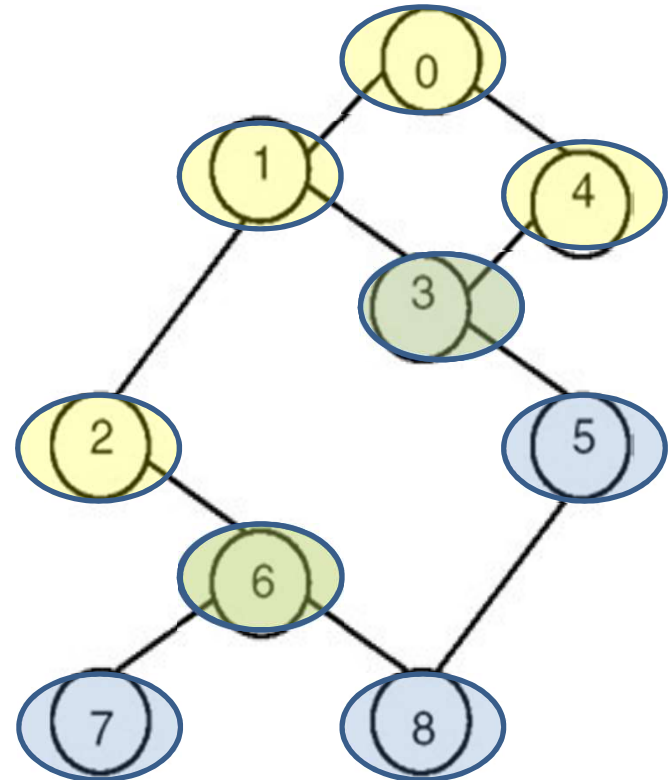
Continuing bfs (3)

- $Q: [4, 2, 3]$
- $v = 4$
- $all = [0, 1, 4]$
- Neighbours of 4: 0, 3
 - No vertices added to Q
- $Q = [2, 3]$
- $v = 2$
- $all = [0, 1, 4, 2]$
- Neighbours of 2: 1, 6 $\rightarrow Q = [3, 6]$



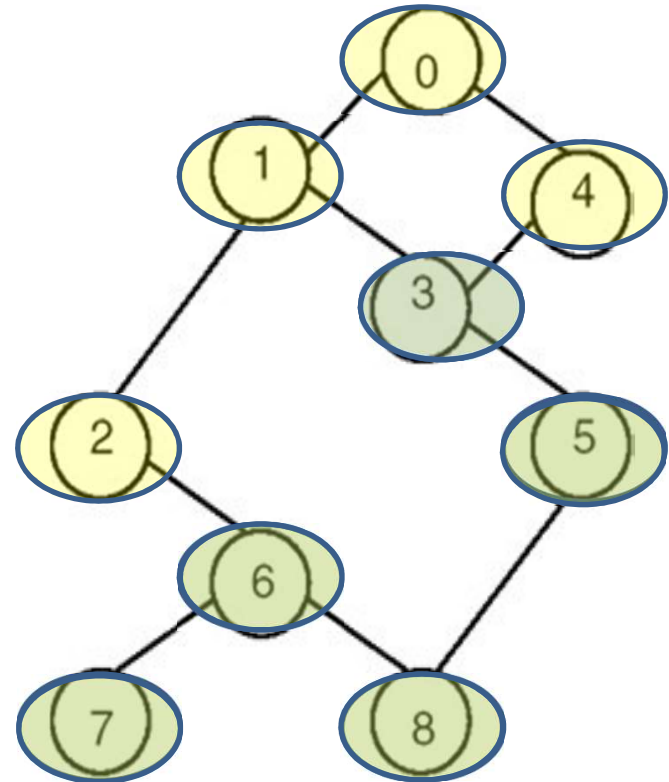
Continuing bfs (4)

- $Q: [3, 6]$
- $v = 3$
- $all = [0, 1, 4, 2, 3]$
- Neighbours of 3: 1, 4, 5
 - $Q = [6, 5]$
- $Q = [6, 5]$
- $v = 6$
- $all = [0, 1, 4, 2, 3, 6]$
- Neighbours of 6: 2, 7, 8
 - $Q = [5, 7, 8]$



Continuing bfs (5)

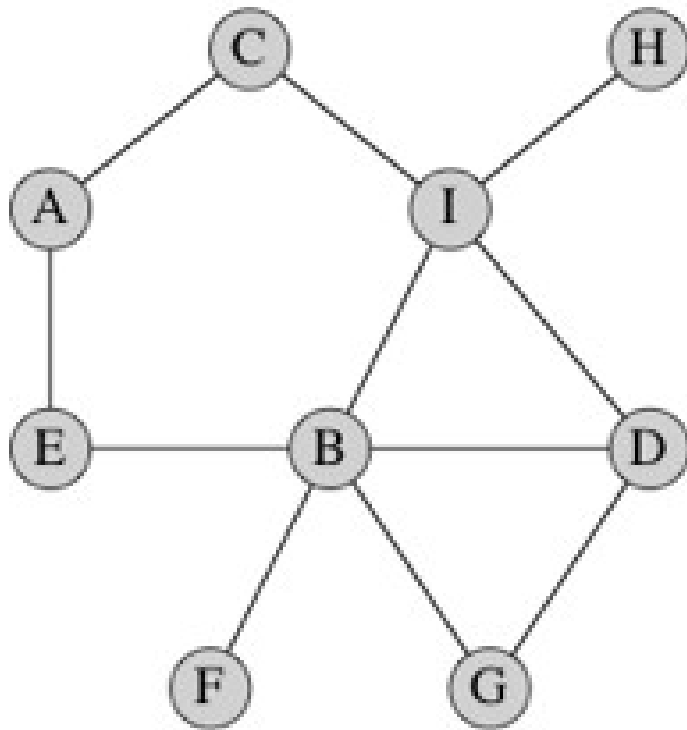
- Q: [5, 7, 8]
- v = 5
- all = [0,1,4,2,3,6,5]
- Neighbours of 5: 3,8 (Q unchanged)
- Q = [7,8]
- v = 7
- all = [0,1,4,2,3,6,5,7]
- Neighbours of 7: 6 (Q unchanged)
- Q = [8]
- v = 8
- all = [0,1,4,2,3,6,5,7,8]
- Neighbours of 8: 5,6 (Q unchanged)
- Q is empty



Another approach: depth-first traversal (dfs)

- Choose a starting point v
- Proceed along a path from v as far as possible
- Then, backup to previous (most recently visited) vertex, and visit its unvisited neighbour (this is called *backtracking*)
 - Repeat while unvisited, reachable vertices remain
- Note: there may be more than one dfs ordering of a graph, starting from v .

Sample dfs orders



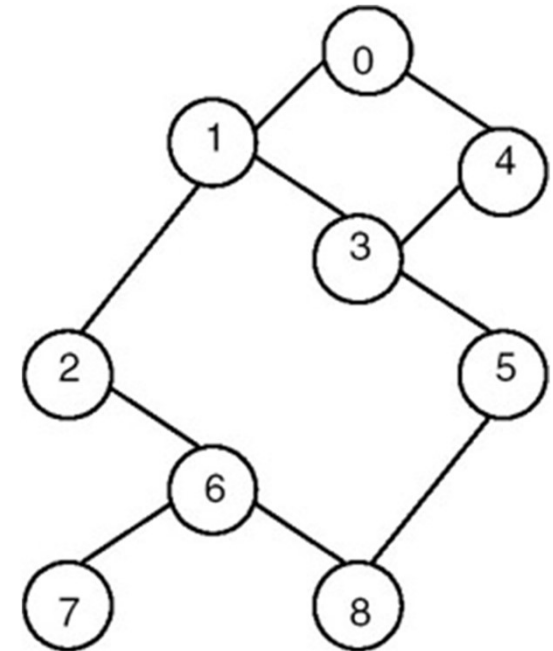
- A, C, I, H, B, F, D, G, E
 - A, E, B, G, D, I, H, C, F
 - B, F, G, D, I, C, A, E, H
 - F, B, G, D, I, H, C, A, E
 - H, I, B, F, G, D, E, A, C
- plus more ...*

Implementing dfs

- We will look at one implementation.
- Assumes an adjacency list representation.
- We use several lists:
 - **a11** includes all "visited" vertices
 - Vertices are appended to the end
 - **S** includes vertices waiting to be "visited" (it will grow and shrink as the algorithm progresses)
 - Vertices are appended to the end and removed from the end of **S** as well

A depth first search traversal solution

```
def dfs(graph, v):  
    all = []  
    S = [v]  
    while S != []:  
        v = S.pop()  
        if v not in all:  
            all.append(v)  
            for w in graph[v]:  
                if w not in all:  
                    S.append(w)  
    return all
```

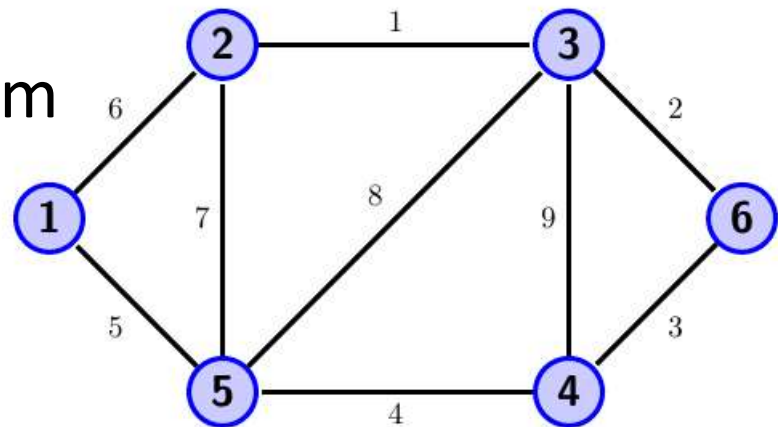


Breadth first vs depth first Searches

- Both need an additional list to store needed information:
 - BFS uses Q:
 - Add to the end and remove from the front
 - Called a Queue
 - DFS uses S:
 - Add to the end and remove from the end
 - Called a Stack
 - Stacks and Queues are both very useful in CS

Extension: Weighted edges

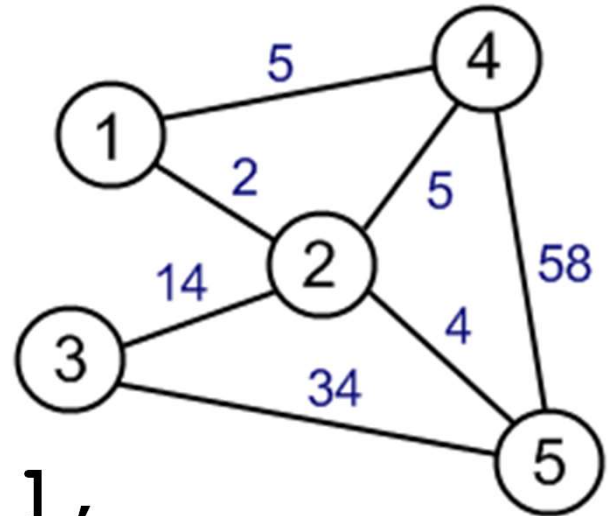
- Each edge has an associated weight. It might represent:
 - Distance between cities
 - Cost to move between locations
 - Capacity of a route
 - Probability of moving from one web page to another



Adjust adjacency list to include weights

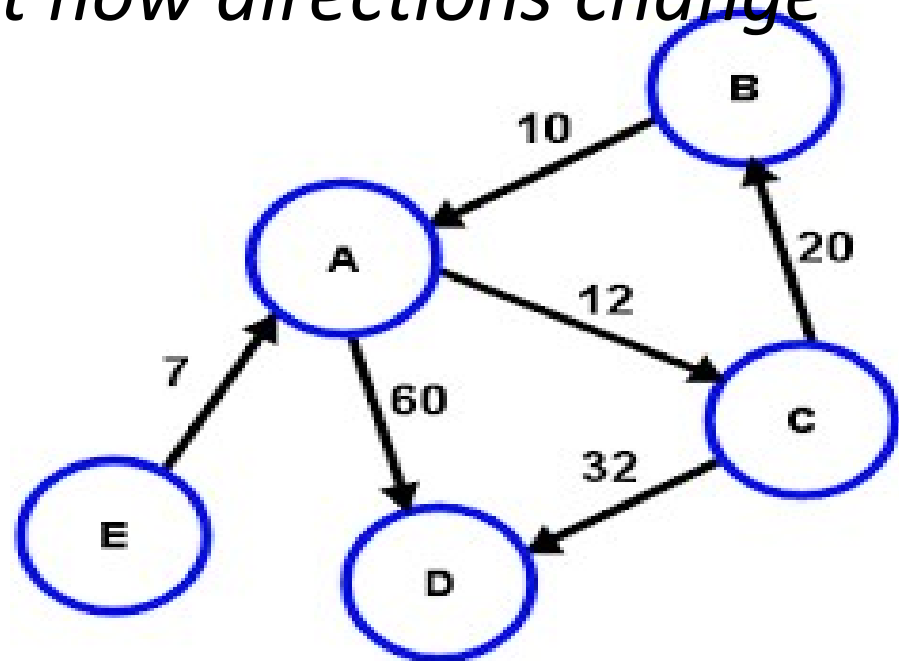
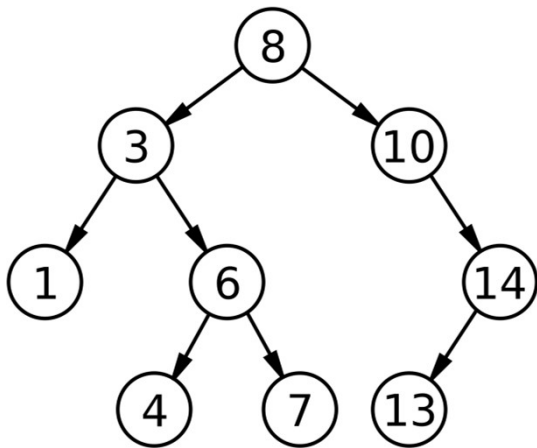
- Adjust our adjacency list to store weights with each edge

- $\{1: [[2, 2], [4, 5]],$
2: $[[1, 2], [3, 14],$
[4, 5], [5, 4]],
- 3: $[[2, 14], [5, 34]],$
4: $[[1, 5], [2, 5], [5, 58]],$
5: $[[2, 4], [3, 34], [4, 58]]\}$



Other types of graphs

- Edges can be directed – from one vertex to another
- Directed edges can have weights as well
- *Exercise: Think about how directions change our representations*



Goals of Module 11

- Understand basic graph terminology
- Understand representation of graphs in Python
- Understand breadth-first and depth-first search traversals