Graphs

CMPT 225

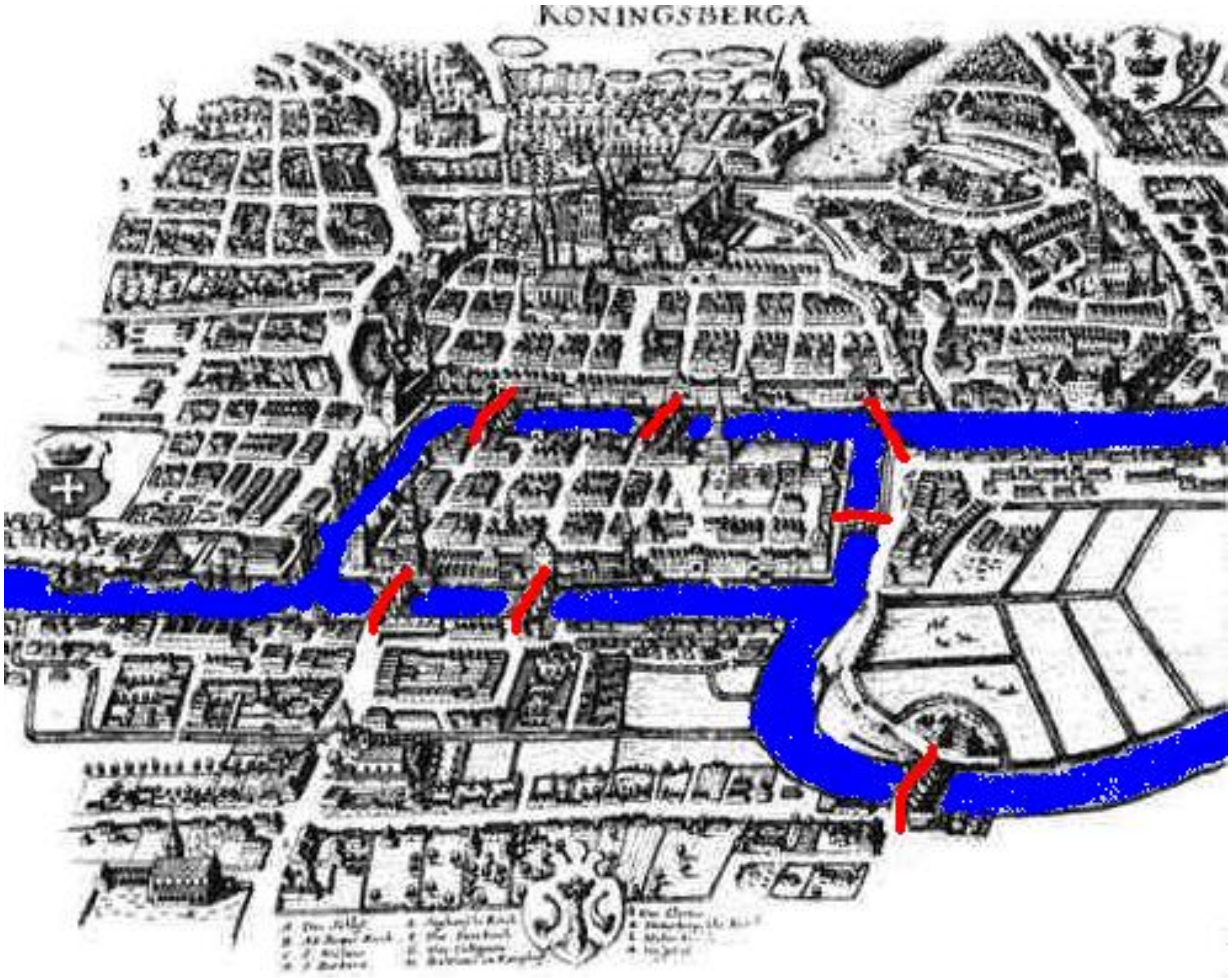
Objectives

- Understand graph terminology
- Implement graphs using
  - Adjacency lists and
  - Adjacency matrices
- Perform graph searches
  - Depth first search
  - Breadth first search
- Perform shortest-path algorithms
  - Disjkstra's algorithm
  - A* algorithm

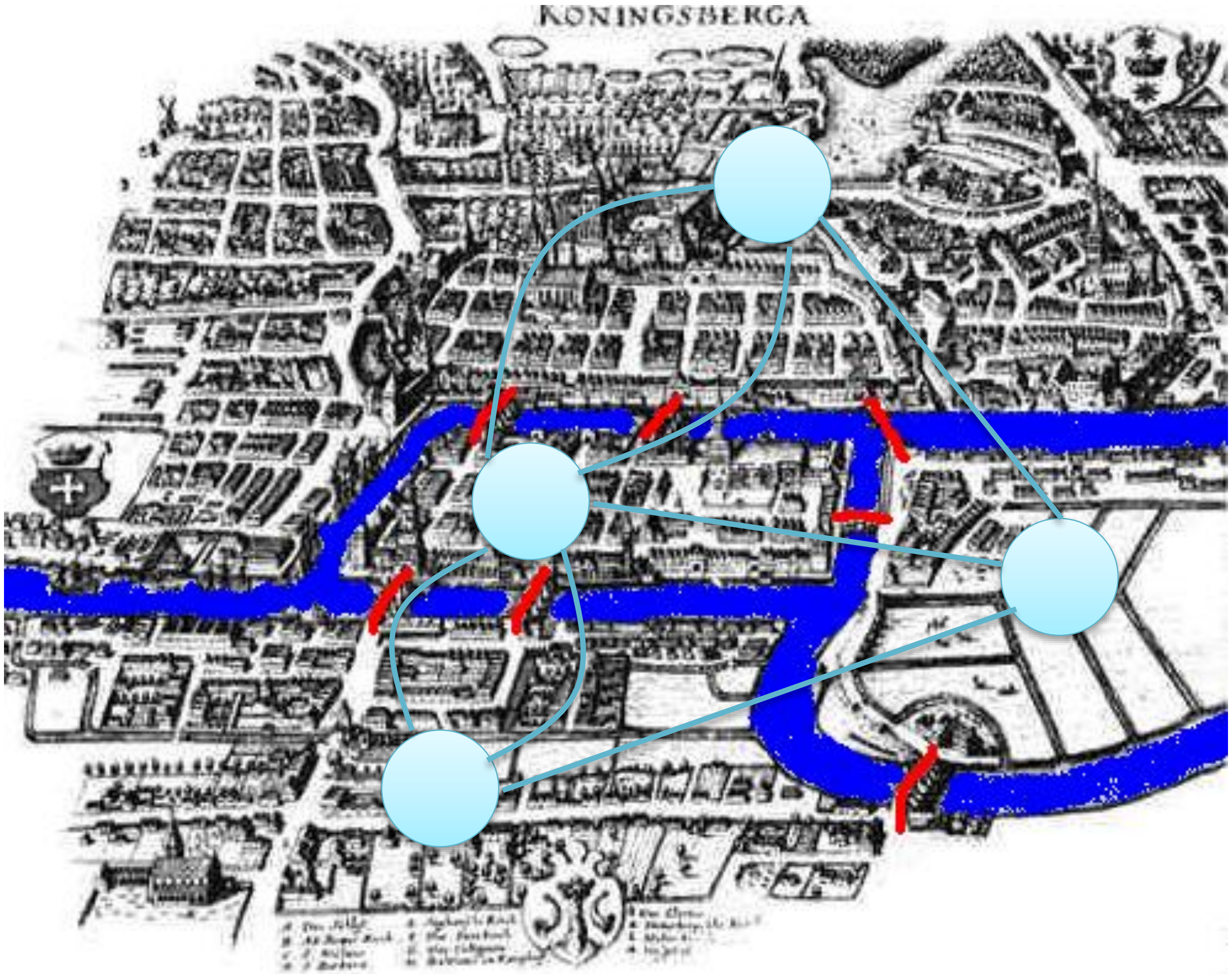
Graph Theory and Euler

- Graph theory is often considered to have been born with Leonhard Euler
  - In 1736 he solved the *Konigsberg bridge problem*
- Konigsberg was a city in Eastern Prussia
  - Renamed Kalinigrad when East Prussia was divided between Poland and Russia in 1945
  - Konigsberg had seven bridges in its centre
    - The inhabitants of Konigsberg liked to see if it was possible to walk across each bridge just once
    - And then return to where they started
  - Euler proved that it was impossible to do this, as part of this proof he represented the problem as a graph

Konigsberg Graph

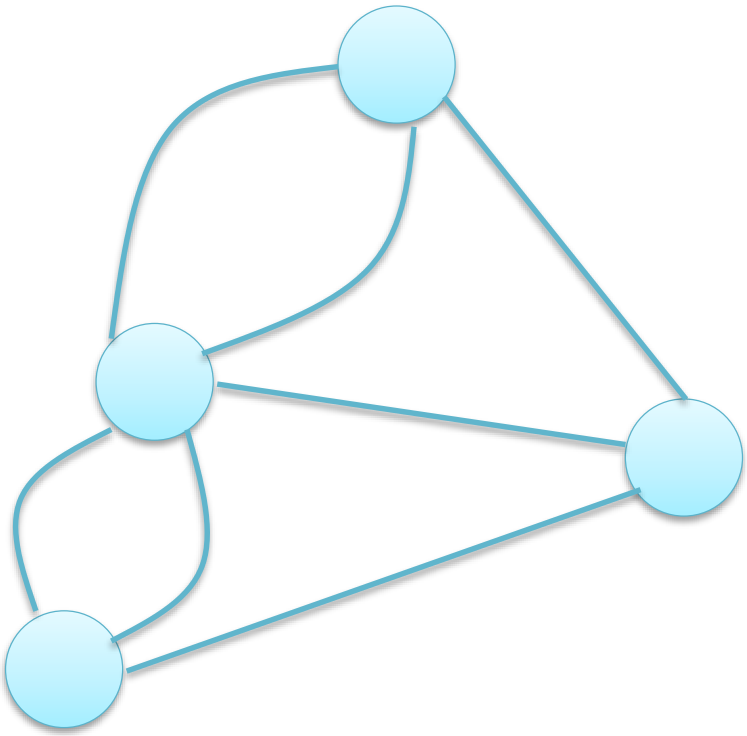


Konigsberg



Multigraphs

- The Konigsberg graph is an example of a *multigraph*
- A multigraph has multiple edges between the same pair of vertices
- In this case the edges represent bridges



Graph Uses

- Graphs are used as representations of many different types of problems
  - Network configuration
  - Airline flight booking
  - Pathfinding algorithms
  - Database dependencies
  - Task scheduling
  - Critical path analysis
  - ...

Graph Terminology

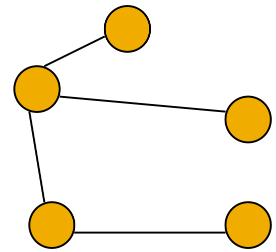
- A graph consists of two sets
  - A set *V* of *vertices* (or nodes) and
  - A set *E* of *edges* that connect vertices
  - $|V|$ is the size of *V*, $|E|$ the size of *E*
- Two vertices may be connected by a *path*
  - A sequence of edges that begins at one vertex and ends at the other
    - A *simple path* does not pass through the same vertex more than once
    - A *cycle* is a path that starts and ends at the same vertex

Numbers of Vertices and Edges

- If a graph has v vertices, how many edges does it have?
  - If every vertex is connected to every other vertex, and we count each direction as two edges
    - $v^2 - v$
  - If the graph is a tree
    - $v - 1$
  - Minimum number of edges
    - 0

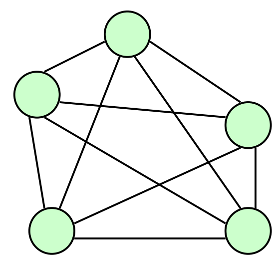
Connected and Unconnected Graphs

- A *connected* graph is one where every pair of distinct vertices has a *path* between them
- A *complete* graph is one where every pair of vertices has an *edge* between them
- A graph cannot have multiple edges between the same pair of vertices
- A graph cannot have *self edges*, an edge from and to the same vertex

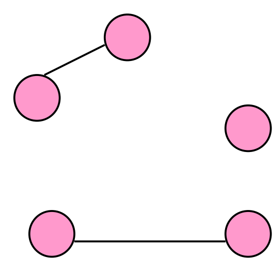


connected graph

and a tree



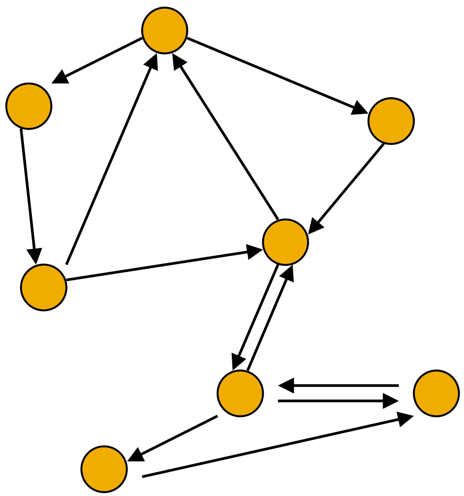
complete graph



unconnected graph

Directed Graphs

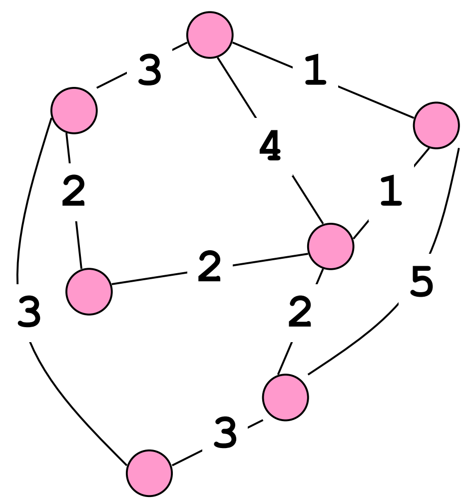
- In a *directed graph* (or digraph) each edge has a direction and is called a directed edge
- A directed edge can only be traveled in one direction
- A pair of vertices in a digraph may have two edges between them, one in each direction



directed graph

Weighted Graphs

- In a *weighted graph* each edge is assigned a weight
  - Edges are labeled with their weights
- Each edge's weight represents the cost to travel along that edge
  - The cost could be distance, time, money or some other measure
  - The cost depends on the underlying problem



weighted graph

Basic Graph Operations

- Create an empty graph
- Test to see if a graph is empty
- Determine the number of vertices in a graph
- Determine the number of edges in a graph
- Determine if an edge exists between two vertices
  - and in a weighted graph determine its weight
- Insert a vertex
  - each vertex is assumed to have a distinct search key
- Delete a vertex, and its associated edges
- Delete an edge
- Return a vertex with a given key

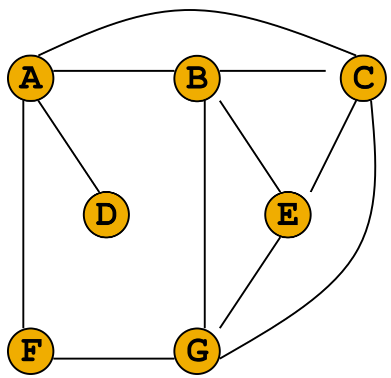
Graph Implementation

- There are two common implementations of graphs
  - Both implementations require a list of all vertices in the set of vertices, *V*
  - The implementations differ in how edges are recorded
- Adjacency matrices
  - Provide fast lookup of individual edges
  - But waste space for sparse graphs
- Adjacency lists
  - Are more space efficient for sparse graphs
  - Can efficiently find all the neighbours of a vertex

Adjacency Matrix

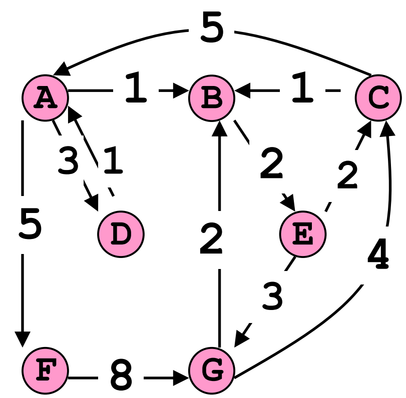
- The edges are recorded in an $|V| * |V|$ matrix
- In an unweighted graph entries in the matrix are
  - 1 when there is an edge between vertices or
  - 0 when there is no edge between vertices
- In a weighted graph entries are either
  - The edge weight if there is an edge between vertices
  - Infinity when there is no edge between vertices
- Adjacency matrix performance
  - Looking up an edge requires O(1) time
  - Finding all neighbours of a vertex requires $O(|\mathbf{V}|)$ time
  - The matrix requires $|V|^2$ space

Adjacency Matrix Examples



line of symmetry

	A	B	C	D	E	F	G
A	0	1	1	1	0	1	0
B	1	0	1	0	1	0	1
C	1	1	0	0	1	0	1
D	1	0	0	0	0	0	0
E	0	1	1	0	0	0	1
F	1	0	0	0	0	0	1
G	0	1	1	0	1	1	0

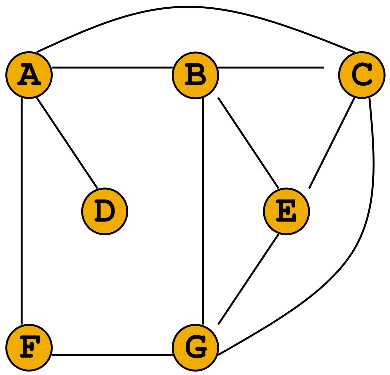


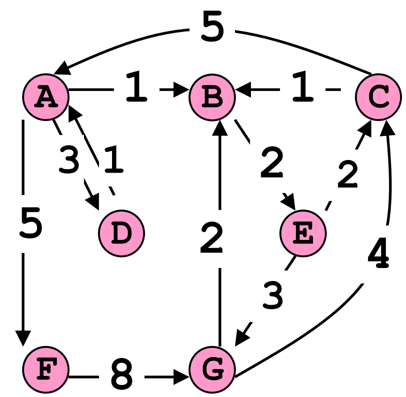
	A	B	C	D	E	F	G
A	∞	1	∞	3	∞	5	∞
B	∞	∞	∞	∞	2	∞	∞
C	5	1	∞	∞	∞	∞	∞
D	1	∞	∞	∞	∞	∞	∞
E	∞	∞	2	∞	∞	∞	3
F	∞	∞	∞	∞	∞	∞	8
G	∞	2	4	∞	∞	∞	∞

Adjacency Lists

- The edges are recorded in an array $|V|$ of linked lists
- In an unweighted graph a list at index *i* records the keys of the vertices adjacent to vertex *i*
- In a weighted graph a list at index *i* contains pairs
  - Which record vertex keys (of vertices adjacent to *i*)
  - And their associated edge weights
- Adjacency List Performance
  - Looking up an edge requires time proportional to the average number of edges
  - Finding all vertices adjacent to a given vertex also takes time proportional to the average number of edges
  - The list requires $O(|E|)$ space

Adjacency List Examples





A	B →	C →	D →	F
B	A →	C →	E →	G
C	A →	B →	E →	G
D	A			
E	B →	C →	G	
F	A →	G		
G	B →	C →	E →	F

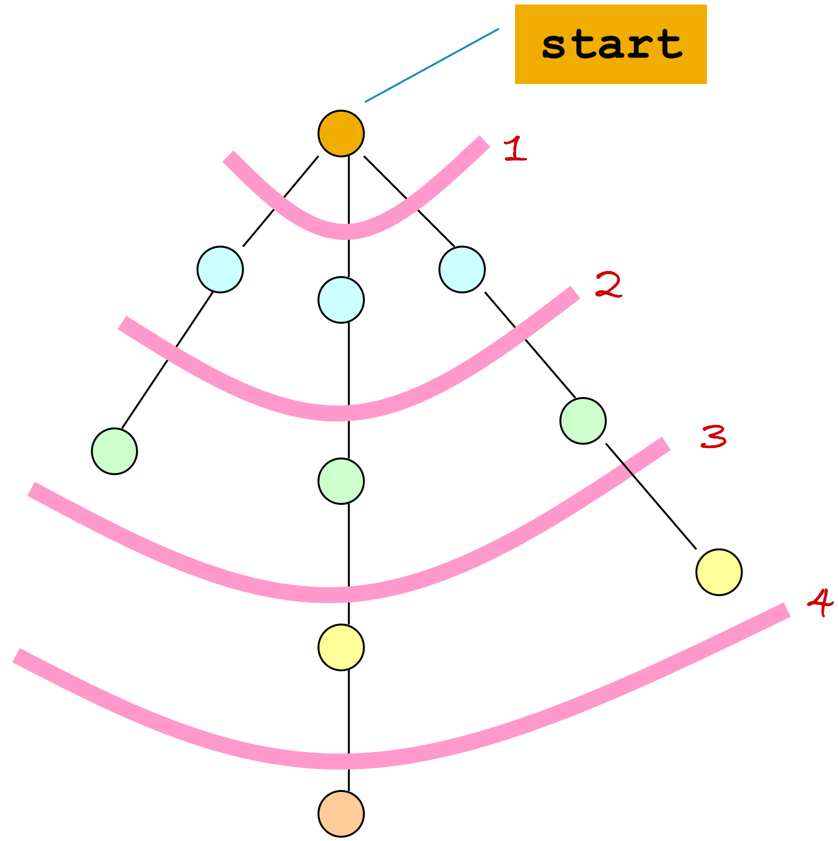
A	B 1 →	D 3 →	F 5
B	E 2		
C	A 5 →	B 1	
D	A 1		
E	C 2 →	G 3	
F	G 8		
G	B 2 →	C 4	

Graph Traversals

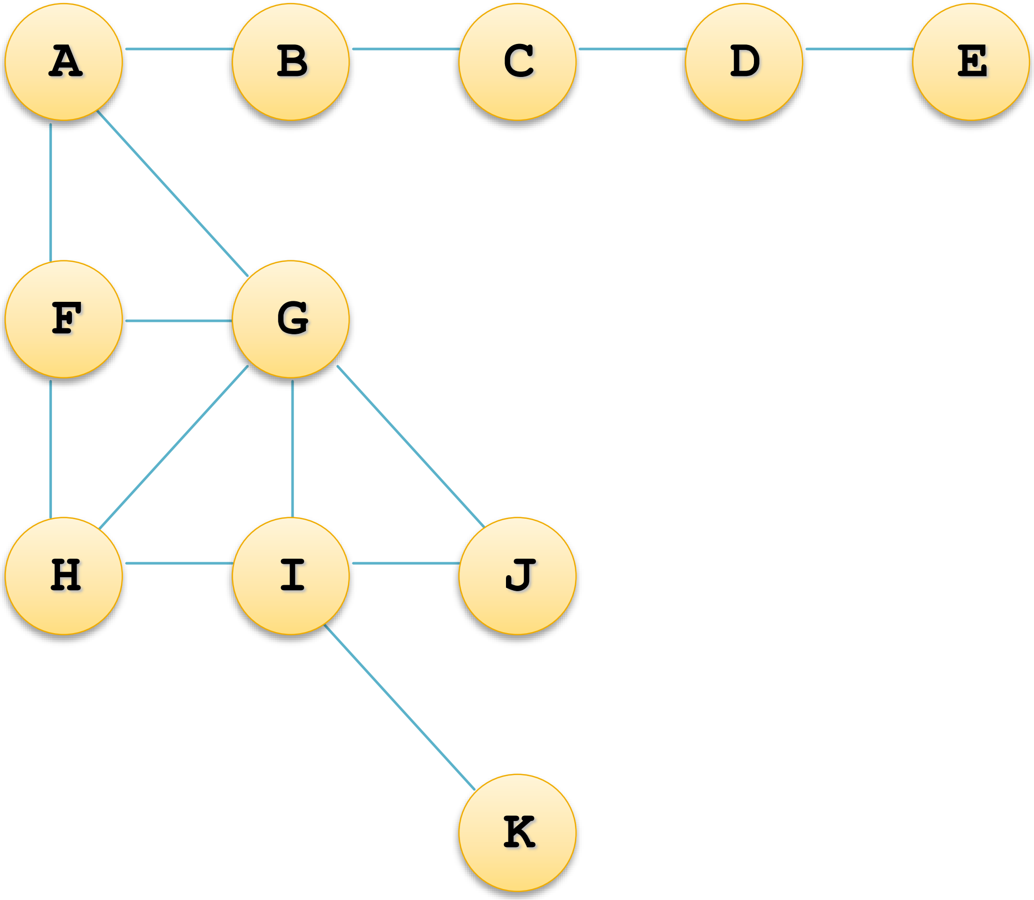
- A graph traversal algorithm visits all of the vertices that can be reached
  - If the graph is not connected some of the vertices will not be visited
  - Therefore a graph traversal algorithm can be used to see if a graph is connected
- Vertices should be marked as *visited*
  - Otherwise, a traversal will go into an infinite loop if the graph contains a cycle

Breadth First Search

- After visiting a vertex, *v*, visit every vertex adjacent to *v* before moving on
- Use a queue to store nodes
  - Queues are FIFO
- BFS:
  - visit and insert start
  - while (*q* not empty)
  - remove node from *q* and make it *current*
  - visit and insert the unvisited nodes adjacent to *current*



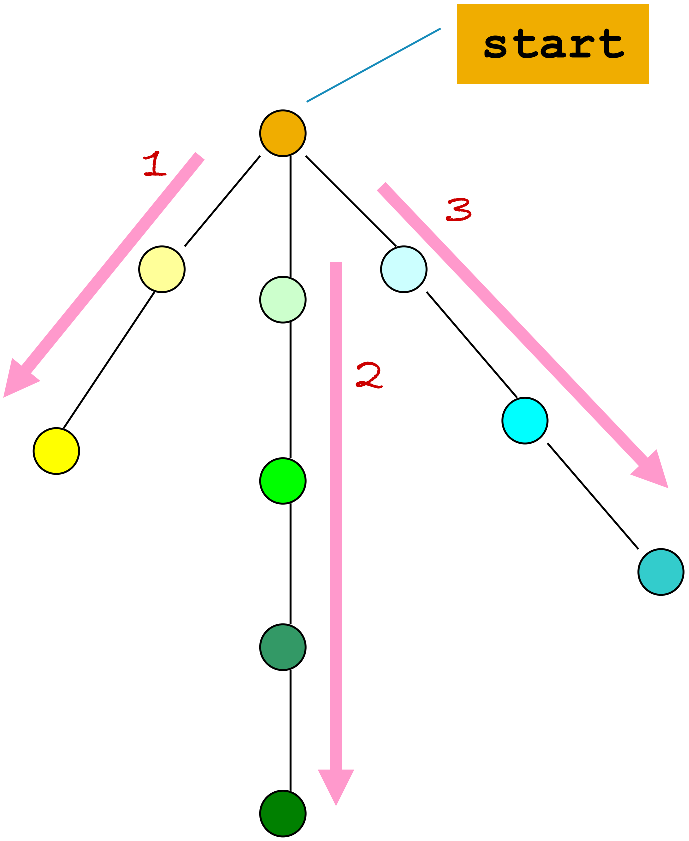
Breadth First Search Example



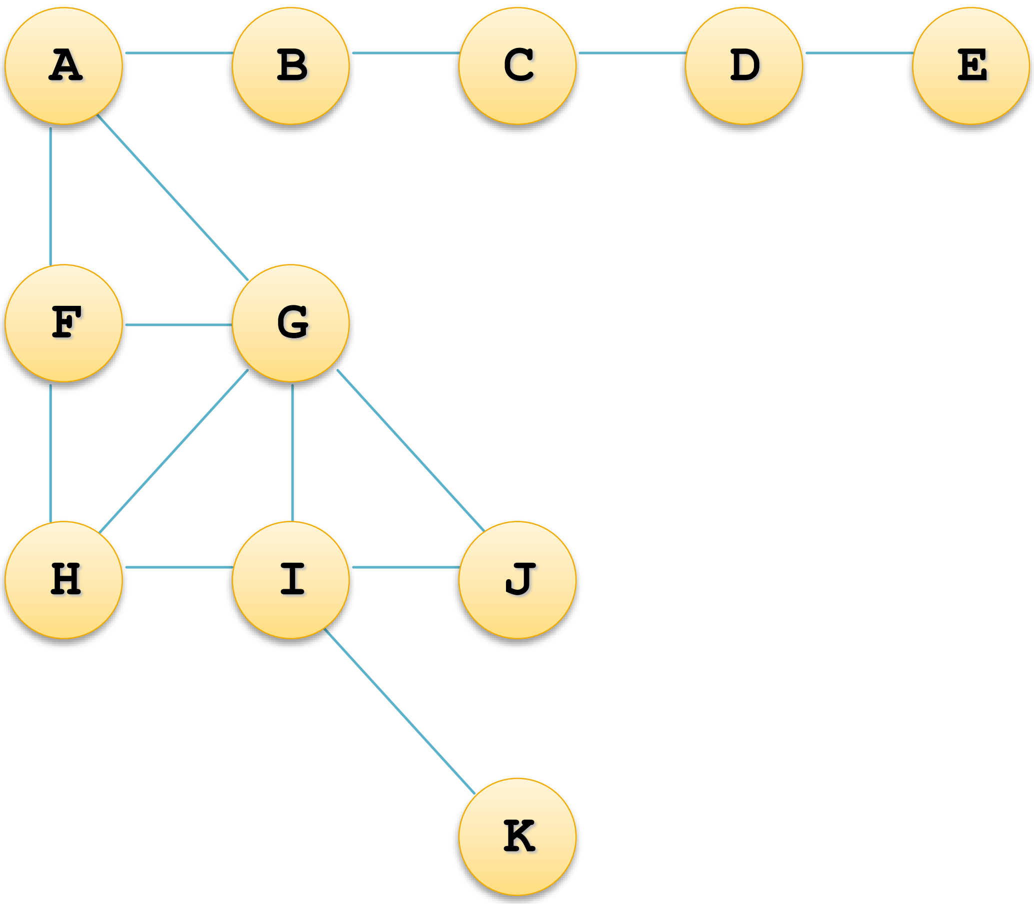
queue	visited
A	A
B	B
F	F
G	G
C	C
H	H
I	I
J	J
D	D
K	K
E	E

Depth First Search

- Visit a vertex, *v*, move from *v* as deeply as possible
- Use a stack to store nodes
  - Stacks are LIFO
- DFS:
  - visit and push start
  - while (*s* not empty)
  - peek at node, *nd*, at top of *s*
  - if *nd* has an unvisited neighbour visit it and push it onto *s*
  - else pop *nd* from *s*



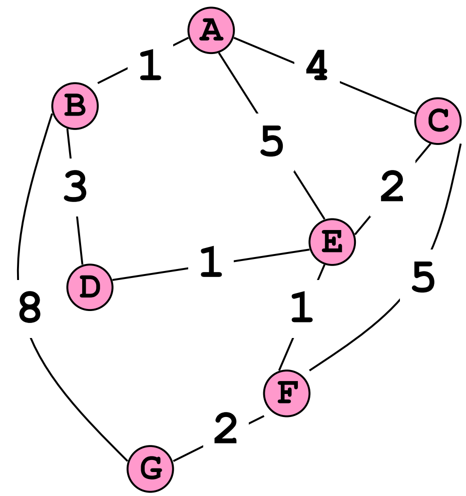
Depth First Search Example



stack			visited
	J	K	A
E	I		B
D	H		C
C	G		D
B	F		E
A			F
			G
			H
			I
			J
			K

Shortest Path Problem

- What is the least cost path from one vertex to another?
  - Referred to as the shortest path between vertices
  - For weighted graphs this is the path that has the smallest sum of its edge weights
- Dijkstra's algorithm finds the shortest path between one vertex and all other vertices
  - The algorithm is named after its discoverer, Edgser Dijkstra



The shortest path between B and G is:

B–D–E–F–G and not

B–G (or B–A–E–F–G)