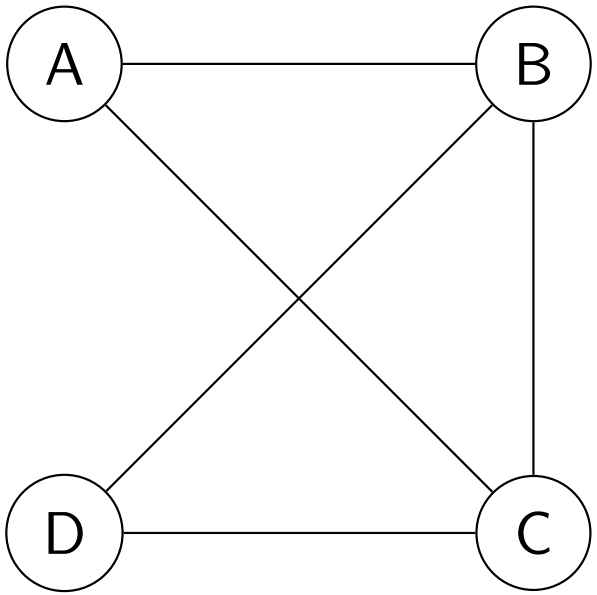
Graphs

Graphs

- A data structure made of nodes (vertices) and a set of edges that connect nodes together
- Formally, A graph $G = (V, E)$, where V is the set of vertices and E is the set of edges
- Vertices are usually referenced via a name or a label
- Edges are represented as a pair of vertices
  - Edge (A,B) connects vertices A and B
- Two kinds of graphs
  - Undirected Graph (frequently known as Graphs)
  - Directed Graph (or Digraph)

Undirected Graphs

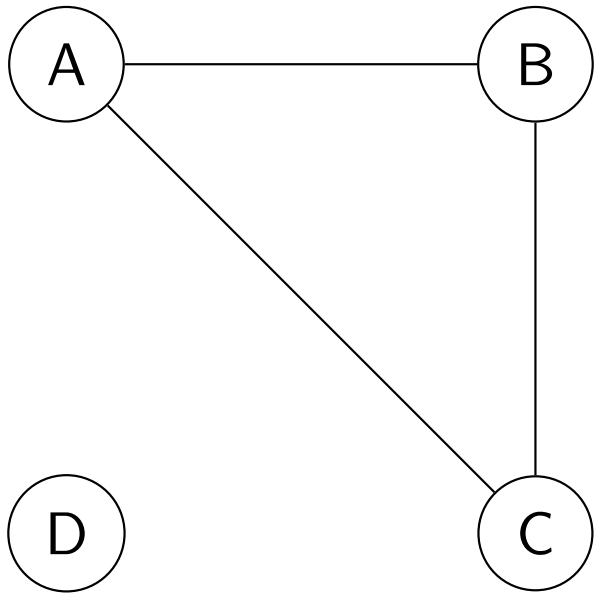
- A graph in which edges can be traversed in either direction
  - Edge (A,B) means that you can go from A to B, or from B to A



Undirected Graphs

- Two vertices are said to be adjacent if there is an edge connecting them
  - Sometimes called neighbors
- An edge of a graph that connects a vertex to itself is called a self-loop or sling
- A path is a sequence of edges that connects two vertices in a graph
  - The length of the path is the number of edges (or number of vertices - 1)
- An undirected graph is considered connected if, for any two vertices in the graph, there is a path between them

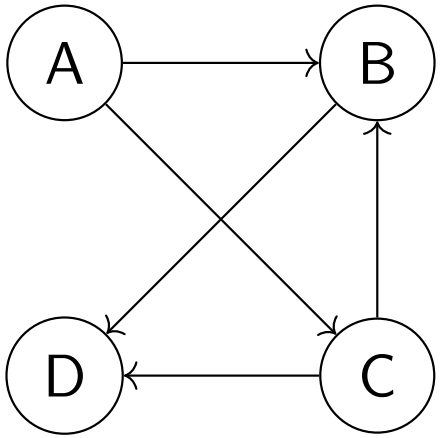
Path and Connectivity Example



- $[(A, B), (B, C)]$ is a *path* from vertex A to vertex C
- The graph is not *connected* because there is no path from D to any other node

Directed Graphs (Digraphs)

- A graph where the edges are one way (and determined by the ordering of the vertices)
  - (A,B) and (B,A) are different, directional edges



- $V = [A, B, C, D]$
- $E[(A, B), (A, C), (B, D), (C, B), (C, D)]$

Digraphs

- A path in a directed graph is a sequence of directed edges
  - In the previous slide, $[(A, B), (B, D)]$ is a path, but $[(A, B), (B, D)]$ is not
- The definition for path remains unchanged
  - The above graph is *not* connected, as it is impossible to get to A from anywhere, or from D to anywhere

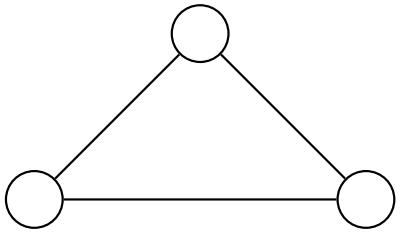
Complete Graphs

- A graph is Complete if it has the maximum number of edges allowed
  - A graph with n vertices can have a maximum of $\frac{n(n-1)}{2}$ edges.
- A complete graph with n vertices is sometimes called K_n

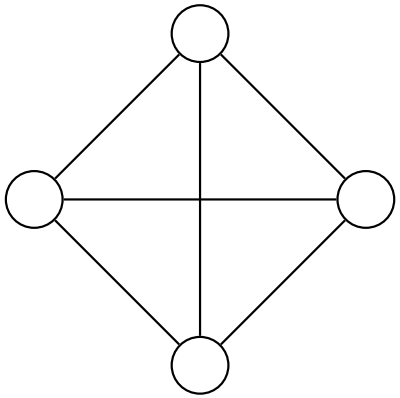
Complete Graphs

K_2

K_3



K_4



Cycles

- A path that begins and ends on the same vertex (and repeats no edges) is called a cycle
- A graph that has no cycles is called acyclic
- K_2 is acyclic, but K_3 contains cycles (start from any vertex)

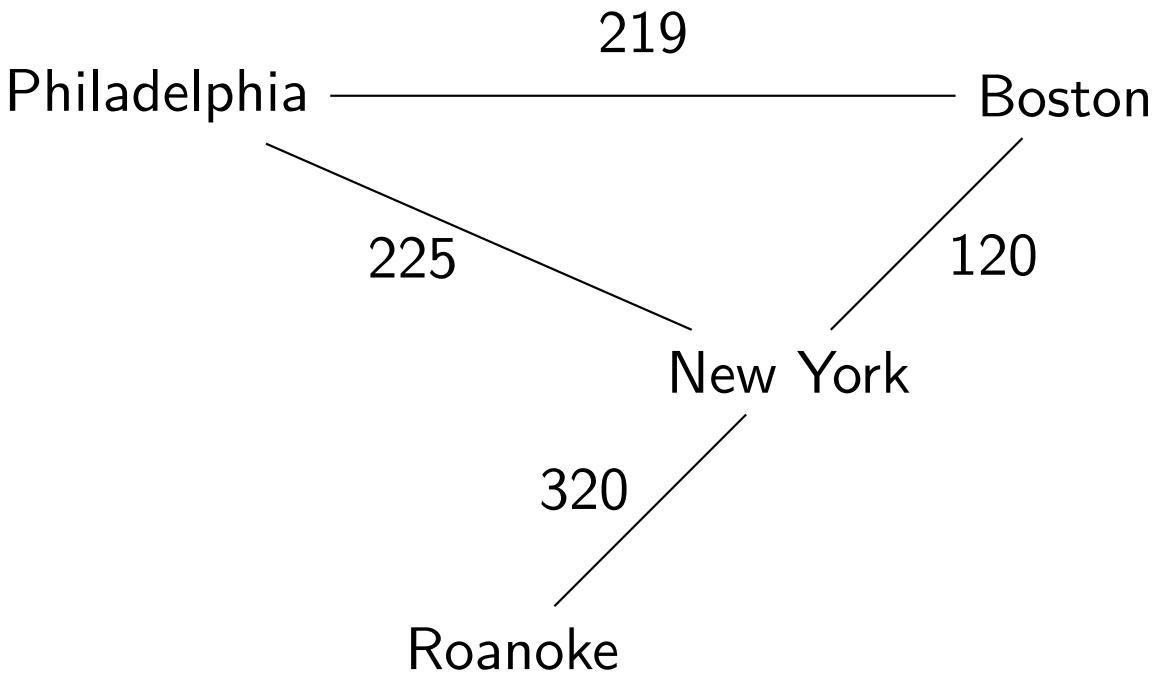
Trees as Graphs?

- Trees are actually special cases of graphs!
  - A tree is an acyclic, connected graph (undirected)

Weighted Graphs

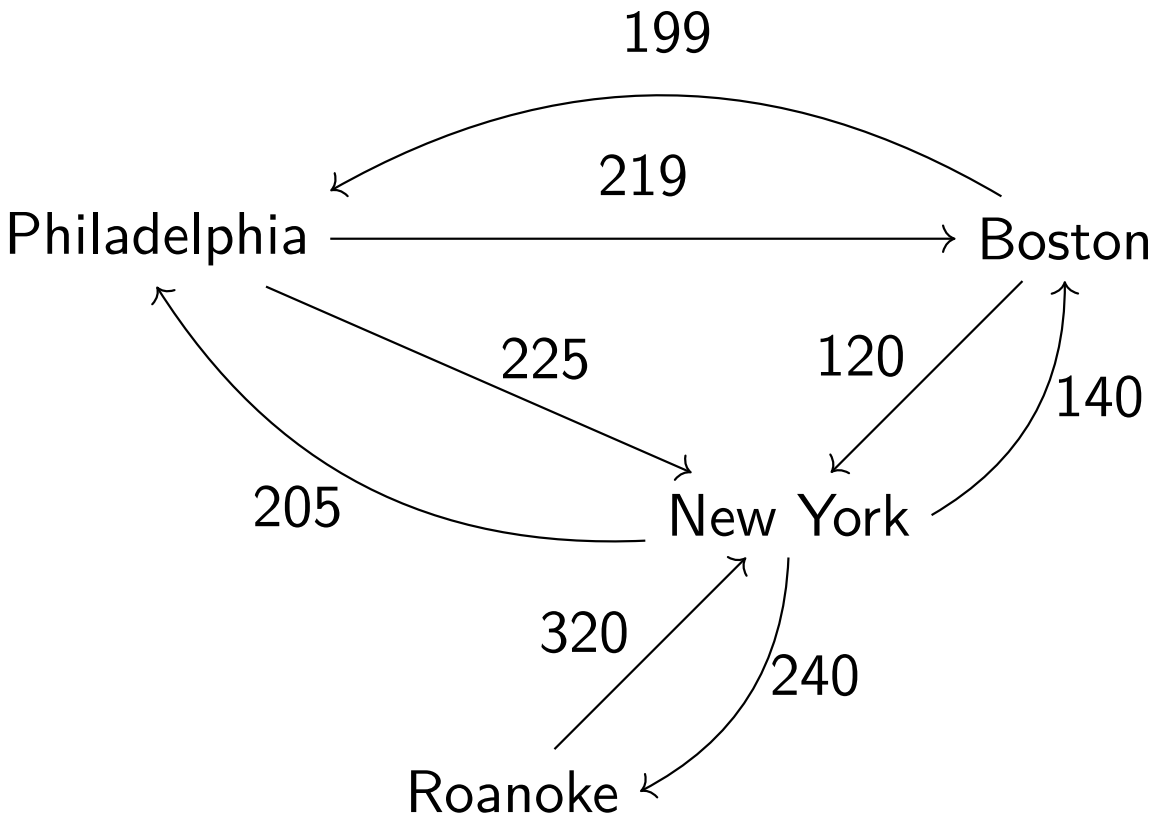
- Sometimes called a network
- Edges have weights or costs associated with each edge
  - The weight of a path is the sum of the weights of the edges in the path
- Weighted graphs may be either undirected or directed
- Edges are represented as triples
  - (from, to, weight)

Weighted Graphs Example



- The weights in this graph could represent flight distances between cities

Weighted Graphs Example

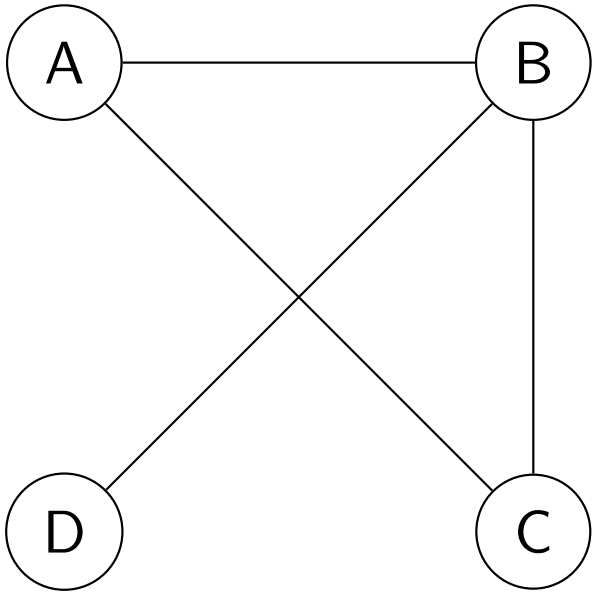


- The weights in this graph could represent costs of flights between cities

Graph Implementation Strategies

- When representing a graph, you should store the vertices in an array or list
- Adjacency Lists
  - Use a list of node which contain a linked list storing the edges within each node
  - For weighted graphs, each edge would be stored as a pair including the weight
- Adjacency Matrices
  - Use a two dimensional array
  - Each position in the array represent an edge
  - Positions in the array are marked with a boolean value (unweighted graph) or a weight (weighted graph) indicating whether or not the edge exists

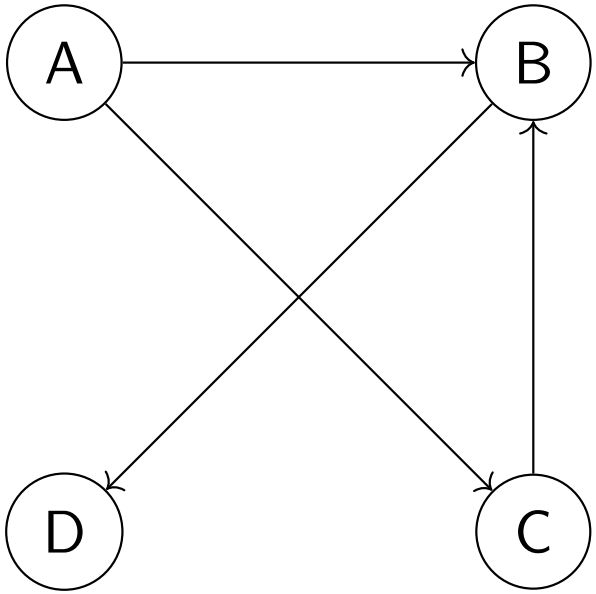
Adjacency List



A		→B, C
B		→A, C, D
C		→A, B
D		→B

Note: The index of the vertex label in the first array corresponds to the posion in the list of lists

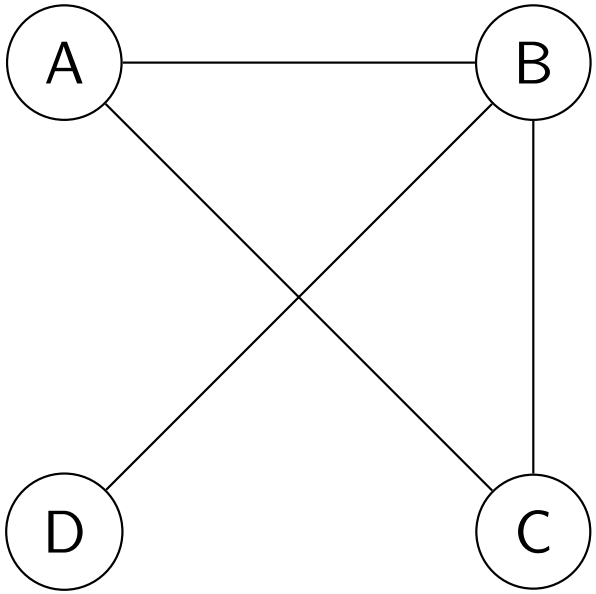
Adjacency List: Digraph



A		→B, C
B		→D
C		→B
D		

Note: The index of the vertex label in the first array corresponds to the posion in the list of lists

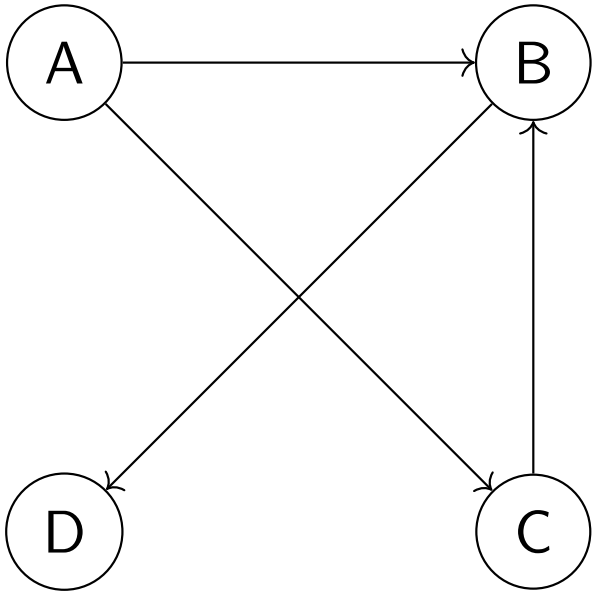
Adjacency Matrix



	A	B	C	D
A	F	T	T	F
B	T	F	T	T
C	T	T	F	F
D	F	T	F	F

Note: the matrix is symmetric across the diagonal.

Adjacency Matrix: Digraph



	A	B	C	D
A	F	T	T	F
B	F	F	F	T
C	F	T	F	F
D	F	F	F	F

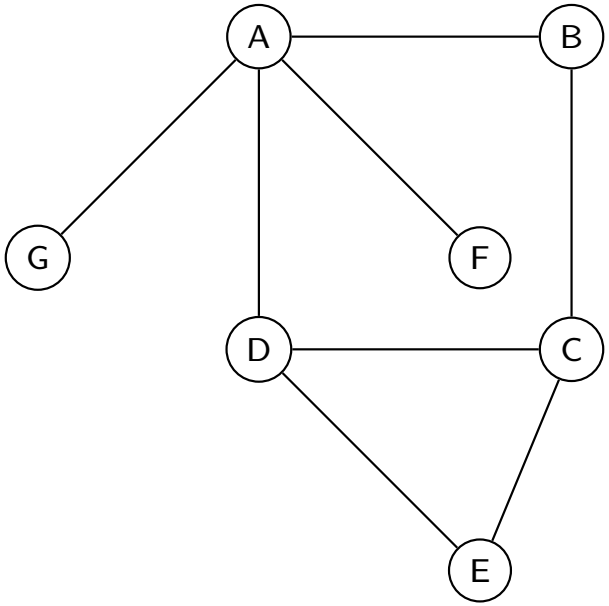
Traversing Graphs

- Traversing a graph is the process of visiting every node once
- Two very common methods
  - Depth First Traversal
  - Breadth First Traversal

Depth First Traversal

- Start at any vertex
- Visit all vertices as far as possible before backtracking
  - Travel as far down a path as you can
  - Back up as little as possible (if you can continue to move forward, do so)
- Visit a vertex, then recursively visit all vertices adjacent to that vertex

Depth First Example

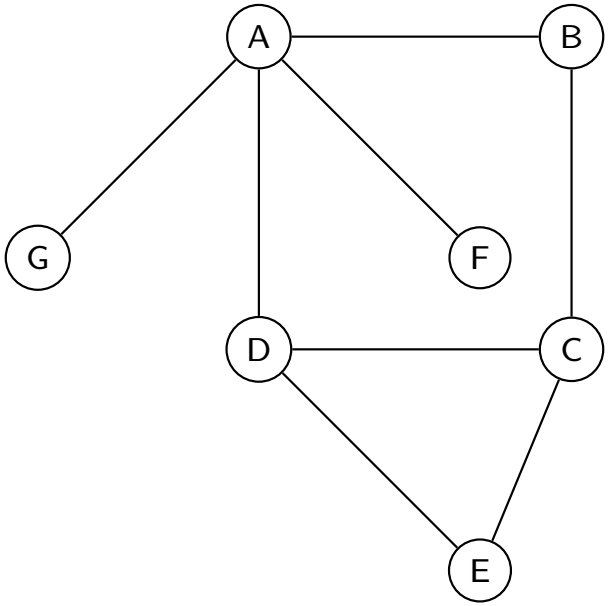


Starting at A:
A, B, C, D, E, (backtrack to A), F, (backtrack to A), G

Breadth First Traversal

- Start at any vertex
- Visit all vertices adjacent to the starting vertex
- Then visit all vertices 2 edges away from the starting vertex
- continue
- Essentially, you are visiting nodes by level from the starting location

Breadth First Example

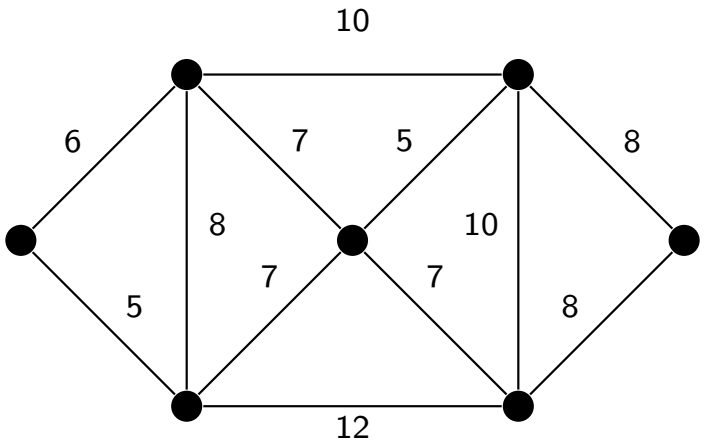


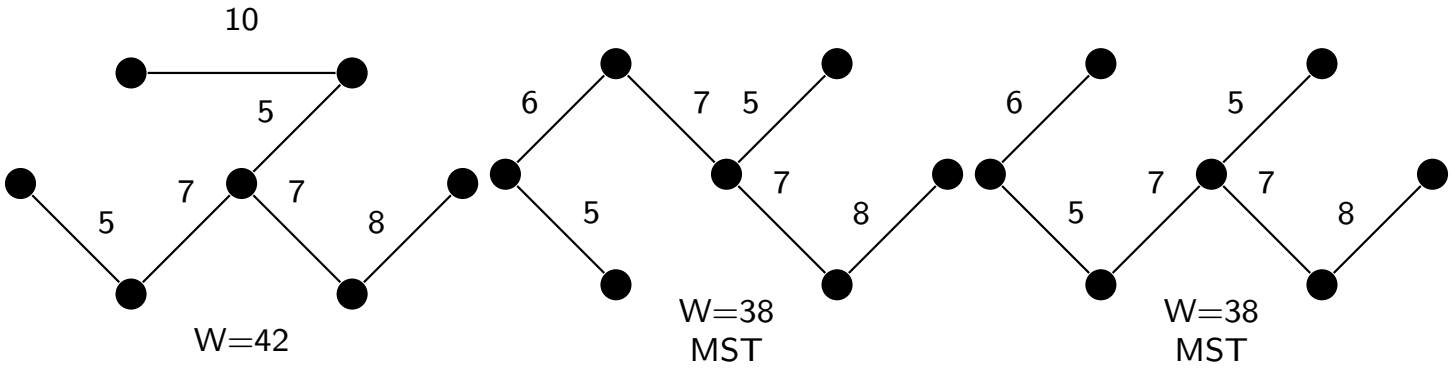
Starting at A:
A, B, D, F, G, C, E

Spanning Trees

- A tree that includes all vertices of a graph and some, but possibly not all, of the edges
- Trees are also graphs
  - The spanning tree may be the entire graph
- A Minimal Spanning Tree (MST) is a spanning tree where the sum of the weights in the spanning tree is less than or equal to the sum of the weights for any other spanning tree for the same graph
- Spanning Trees and Minimal Spanning Trees are not necessarily unique for a given graph

MST Example

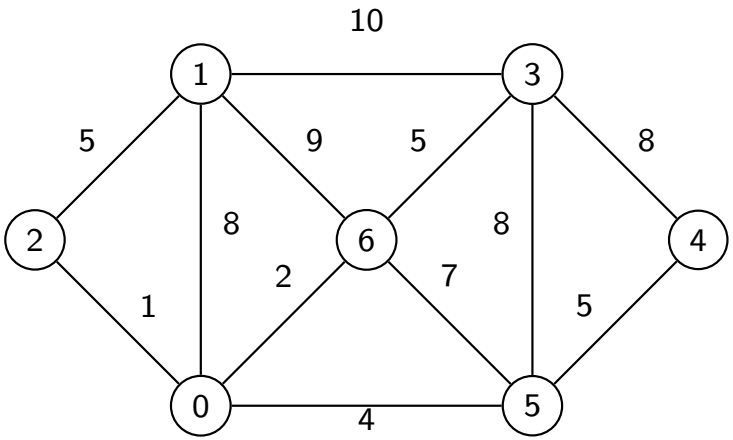




Shortest Paths

- There are two possibilities for determining the "shortest" path in a graph
  - Determine the literal shortest path between a source vertex and a target vertex (fewest number of edges)
  - Find the cheapest path in a wheighted graph (Dijkstra)
- Dijkstra's algorithm finds the shortest path from a single source vertex to every other vertex in the graph (Single Source Shortest Path, or SSSP)

Example: Dijkstra's Algorithm



Starting at vertex 1

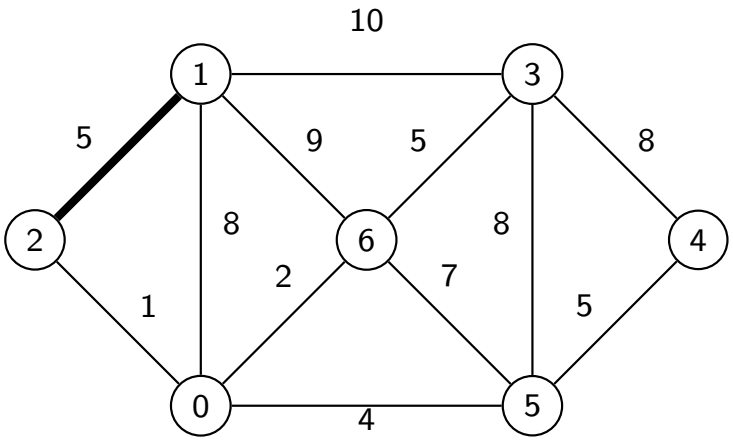
cost:

∞	0	∞	∞	∞	∞	∞
0	1	2	3	4	5	6

parent:

	-1					
0	1	2	3	4	5	6

Example: Dijkstra's Algorithm



Starting at vertex 1

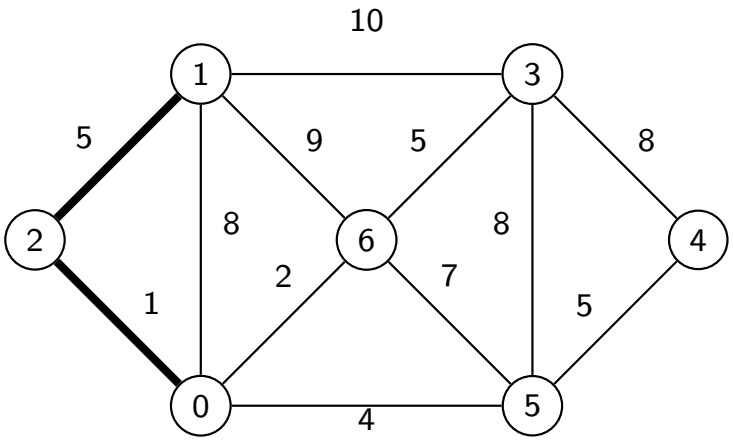
cost:

∞	0	5	∞	∞	∞	∞
0	1	2	3	4	5	6

parent:

	-1	1				
0	1	2	3	4	5	6

Example: Dijkstra's Algorithm



Starting at vertex 1

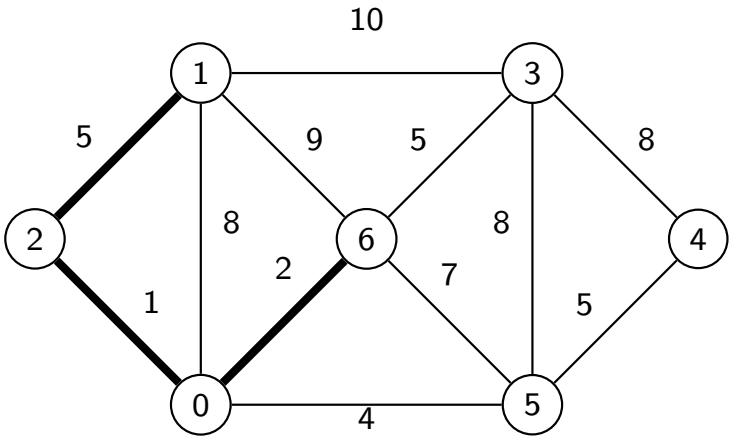
cost:

6	0	5	∞	∞	∞	∞
0	1	2	3	4	5	6

parent:

2	-1	1				
0	1	2	3	4	5	6

Example: Dijkstra's Algorithm



Starting at vertex 1

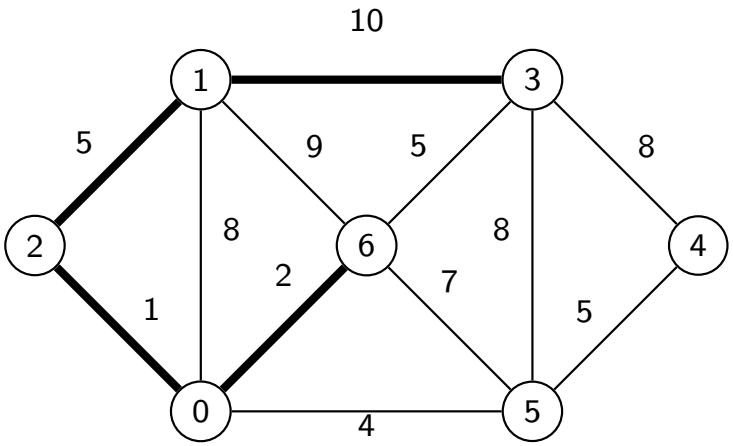
cost:

6	0	5	∞	∞	∞	8
0	1	2	3	4	5	6

parent:

2	-1	1				0
0	1	2	3	4	5	6

Example: Dijkstra's Algorithm



Starting at vertex 1

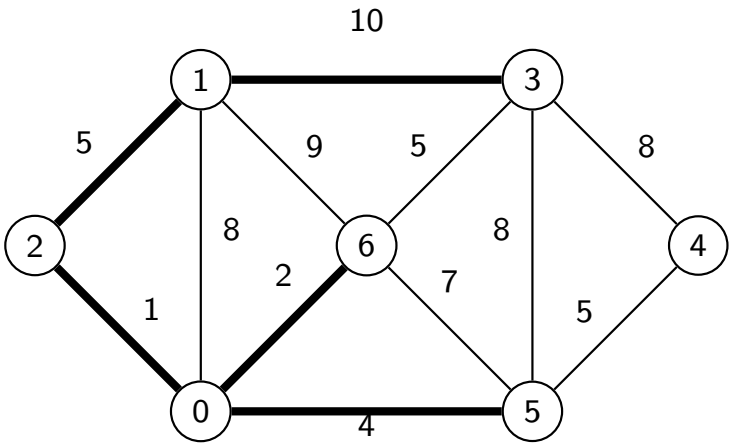
cost:

6	0	5	10	∞	∞	8
0	1	2	3	4	5	6

parent:

2	-1	1	1			0
0	1	2	3	4	5	6

Example: Dijkstra's Algorithm



Starting at vertex 1

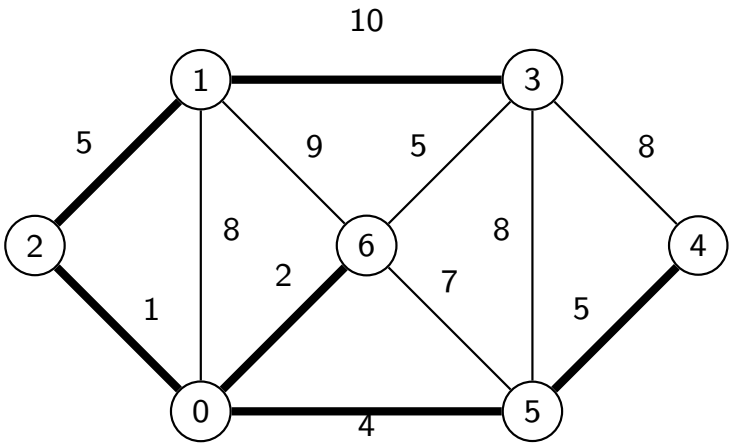
cost:

6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent:

2	-1	1	1		0	0
0	1	2	3	4	5	6

Example: Dijkstra's Algorithm



Starting at vertex 1

cost:

6	0	5	10	14	10	8
0	1	2	3	4	5	6

parent:

2	-1	1	1	5	0	0
0	1	2	3	4	5	6