CS151 Intro to Data Structures

Graphs

Announcements

HW8 and Lab 10 Due Sunday Dec 15th

Final Exam Practice Questions on Piazza

- review before next lecture

Today: LAST TOPIC!

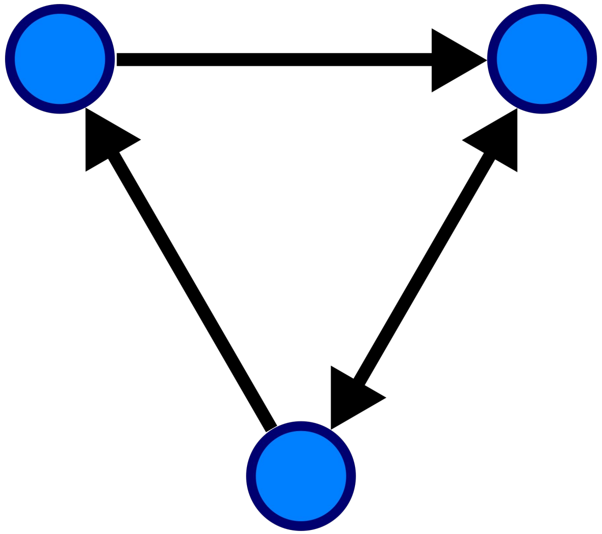
Course Evaluations

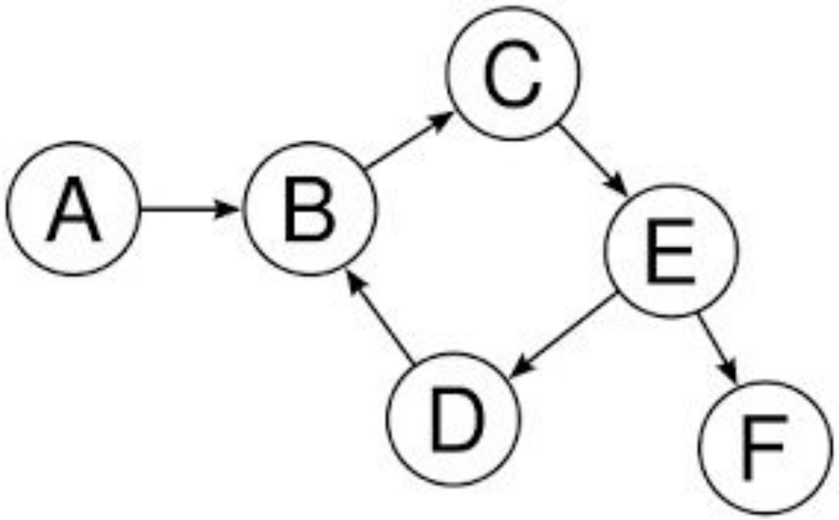
Graphs

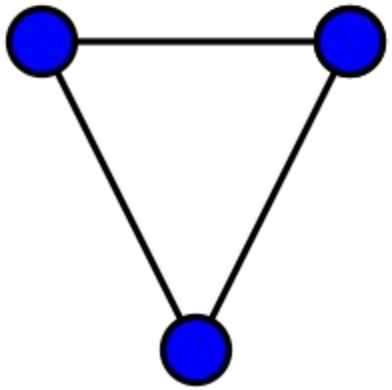
- Terminology
- Data Structures for Graphs
  - Adjacency Lists
  - Adjacency Matrix
- Shortest Paths
  - Djikstra’s Algorithm

Graphs

- A way of representing relationships between pairs of objects
- Consist of ***Vertices (V)*** with pairwise connections between them ***Edges (E)***
- A ***Graph G*** is a set of vertices and edges **(V, E)**

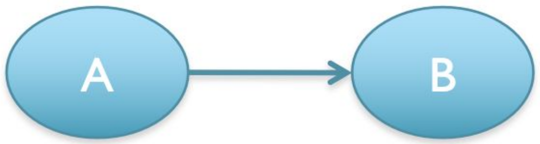






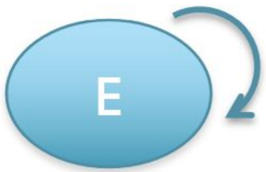
Edges

- An edge (u, v) connects vertices u and v
- Edges can be ***directed*** or ***undirected***
- An edge is said to be ***incident*** to a vertex if the vertex is one of the endpoints



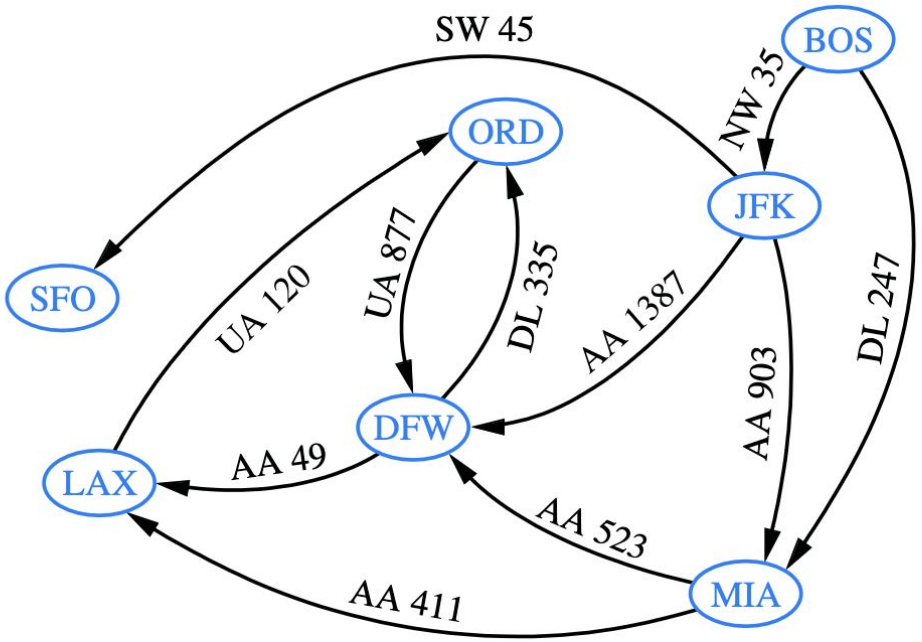
Directed Edge

Undirected Edge



Self Edge
(Unusual but usually allowed)

Directed vs Undirected Graphs



Example of a directed graph representing a flight network.

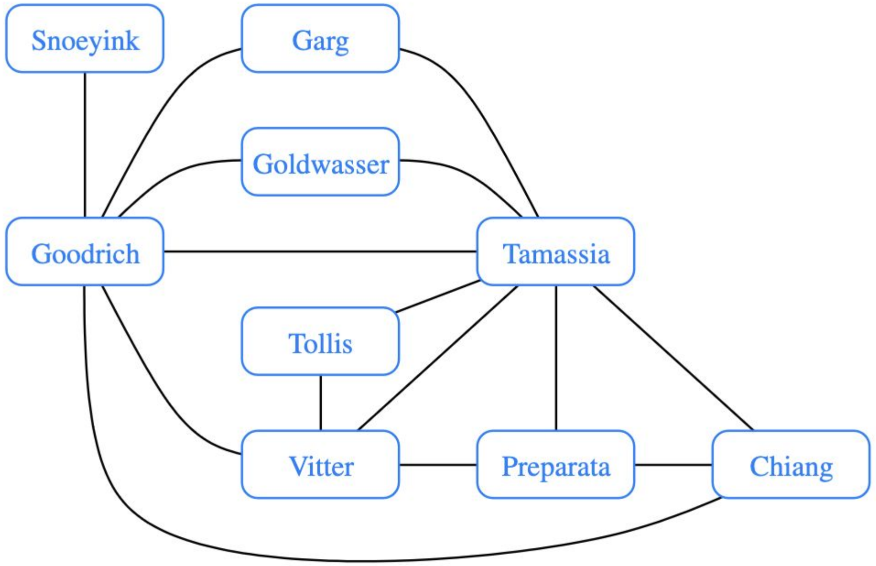


Figure 14.1: Graph of coauthorship among some authors.

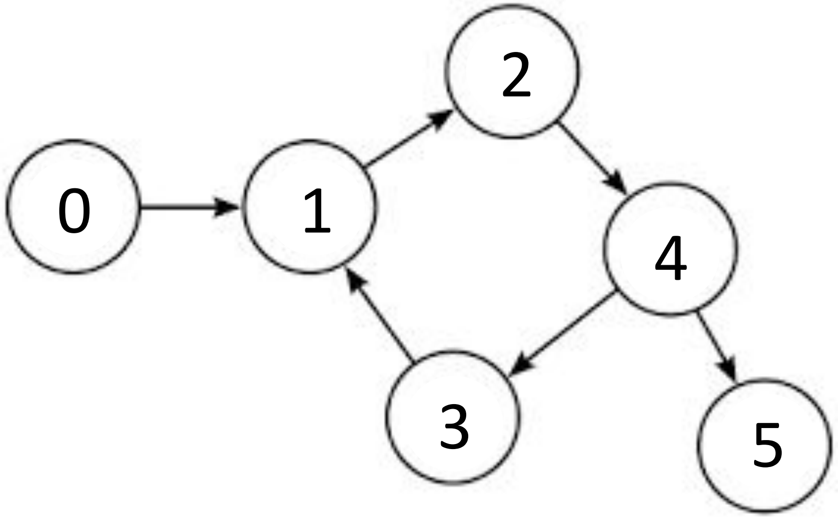
Graphs

- Terminology
- **Data Structures for Graphs**
  - Adjacency Lists
  - Adjacency Matrix
- Shortest Paths
  - Djikstra’s Algorithm

Representing a graph

Adjacency List -

For each vertex v, we maintain a separate list containing the edges that are outgoing from v



Graph ADT

numVertices(): Returns the number of vertices of the graph.

vertices(): Returns an iteration of all the vertices of the graph.

numEdges(): Returns the number of edges of the graph.

edges(): Returns an iteration of all the edges of the graph.

getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).

endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.

opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.

outDegree(v): Returns the number of outgoing edges from vertex v.

inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).

Graph ADT

outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.

incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).

insertVertex(x): Creates and returns a new Vertex storing element x.

insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.

removeVertex(v): Removes vertex v and all its incident edges from the graph.

removeEdge(e): Removes edge e from the graph.

Representing a graph as an **AdjacencyList**

For each vertex v, we maintain a separate list containing the edges that are outgoing from v

Given that we will be inserting and removing vertices and edges, which data structure should we use?

Representing a graph as an **AdjacencyList**

How might we implement the following methods with an AdjacencyList representation?

1. addVertex
2. addEdge
3. removeVertex
4. removeEdge

Representing a graph - Adjacency List

Runtime Complexity: (In terms of V and E rather than n)

- addVertex:
  - O(V*E)
- addEdge:
  - O(E) if we check for duplicates and add to tail
  - O(1) if we add to head
- removeVertex:
  - O(V*E)
- removeEdge:
  - O(E)

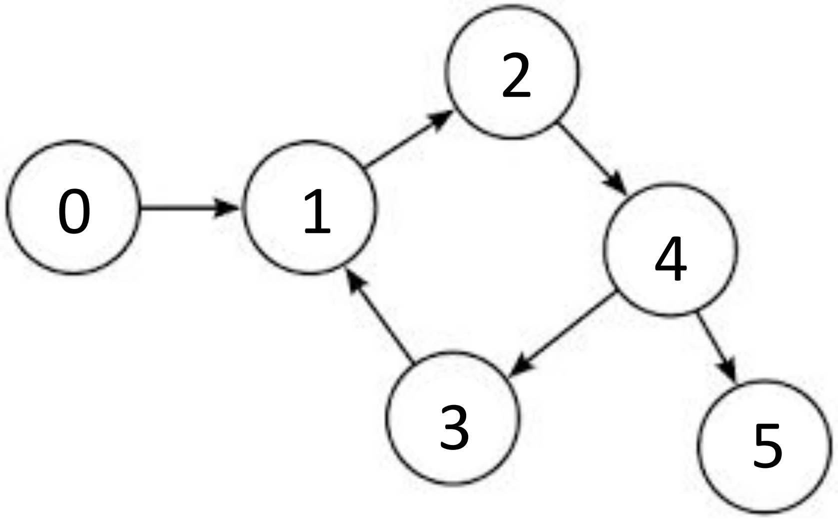
Representing a graph

Adjacency Matrix -

each index in the array is another array

Maintains an VxV matrix

where each slot (i,j) represents an outgoing edge from i to j



	1				
		1			
				1	
	1				
			1		1

Representing a graph

Let's implement a graph as an Adjacency Matrix

Representing a graph - Adjacency Matrix

Runtime Complexity: (In terms of V and E rather than n)

- addVertex:
    - O(V^2)
- addEdge:
    - O(1)
- removeVertex:
    - O(V)
- removeEdge:
    - O(1)

Graphs

- Terminology
- Data Structures for Graphs
  - Adjacency Lists
  - Adjacency Matrix
- **Traversals**
- Shortest Paths
  - Djikstra’s Algorithm

Reachability

Reachability is determining if there exists a path between two vertices in a graph

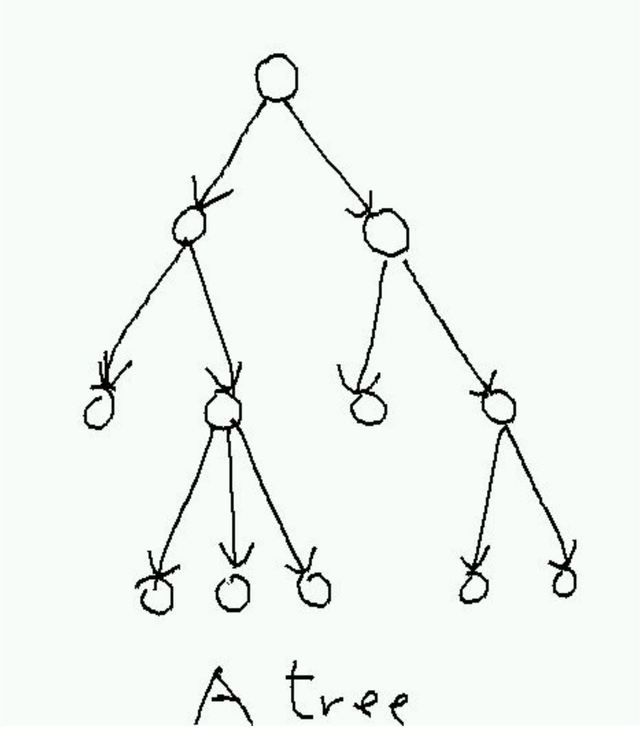
Common questions about graphs involve **Reachability**

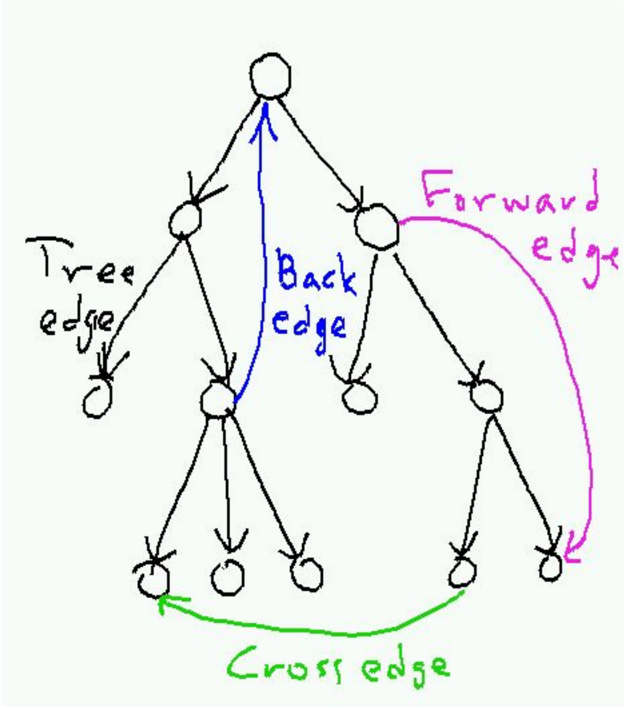
- Does a path exist from vertex *u* to vertex *v*?
- Find all vertices that are reachable from *v*

Depth First Traversal

```
void DFS(root) {
    for each child of root:
        DFS(child)
}
```

Does this work for graphs?

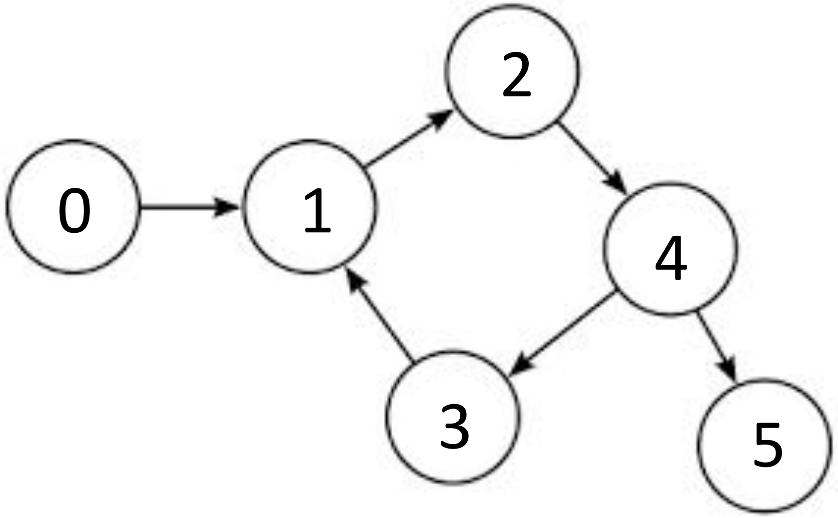




Depth First Traversal

How can we modify the code to deal with cycles?

```
void DFS(root) {
    for each child of root:
        DFS(child)
}
```



Keep track of what we've already visited!

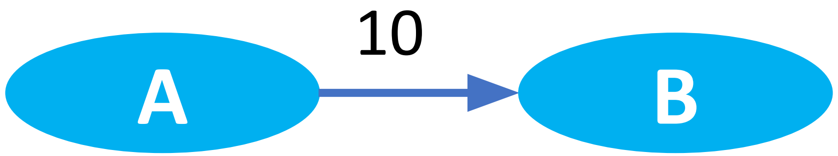
Let's code this for a Matrix Graph

Graphs

- Terminology
- Data Structures for Graphs
  - Adjacency Lists
  - Adjacency Matrix
- **Shortest Paths**
  - Djikstra’s Algorithm

Weighted Graphs

Edges have weights/costs



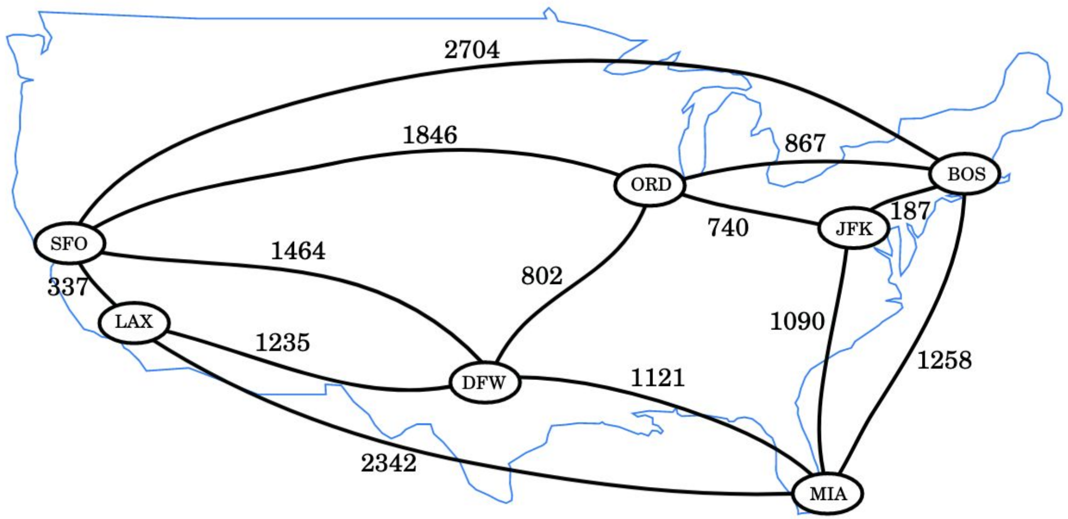


Figure 14.14: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimum-weight path in the graph from JFK to LAX.

Shortest Paths

A **path** is defined as a set of edges

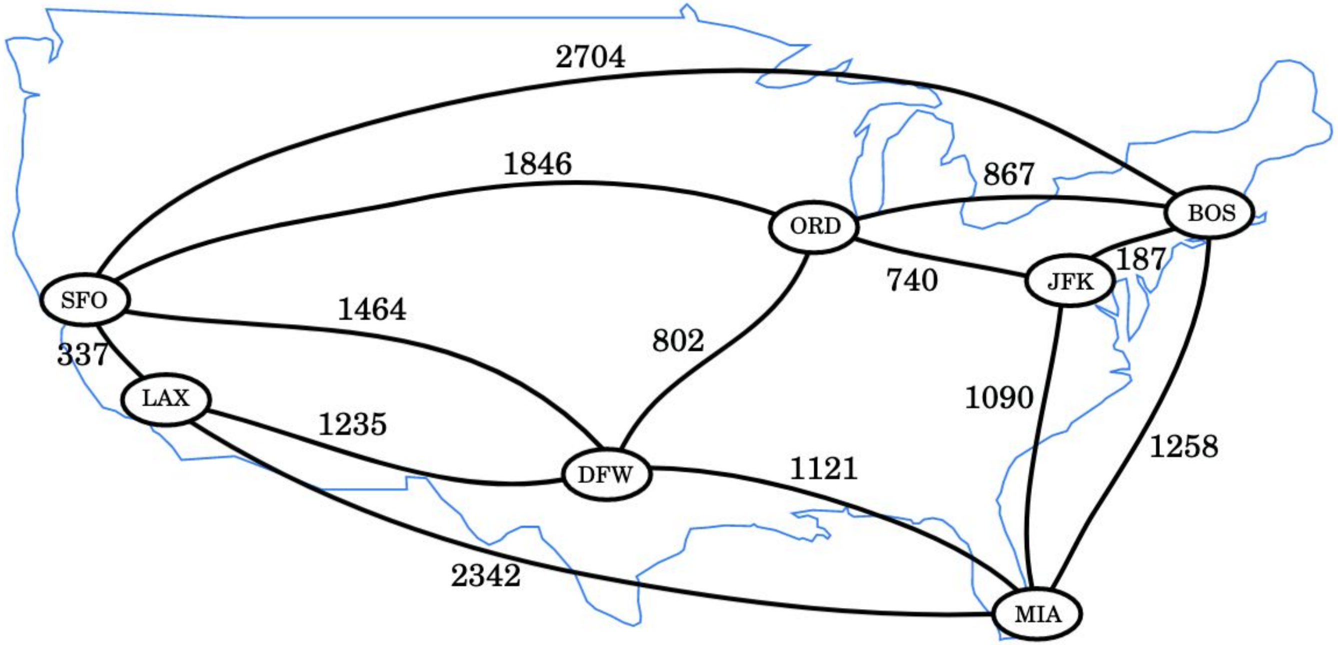
$$P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$$

The *length* of a path is the sum of the weights of the edges

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

Shortest Paths

What is the length of the path P = ((SFO, DFW), (DFW, MIA), (MIA, JFK))



Shortest Paths

What is the shortest path from SFO to JFK?

There are many possible paths...

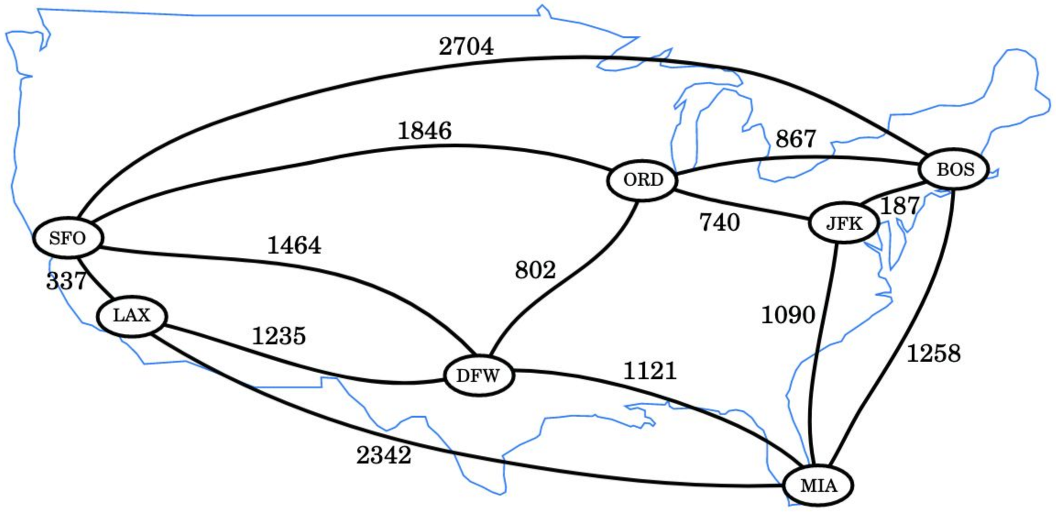
((SFO, ORD), (ORD, JFK))

((SFO, LAX), (LAX, MIA), (MIA, JFK))

((SFO, BOS), (BOS, JFK))

....

((SFO, DFW), (DFW, ORD), (ORD, JFK))

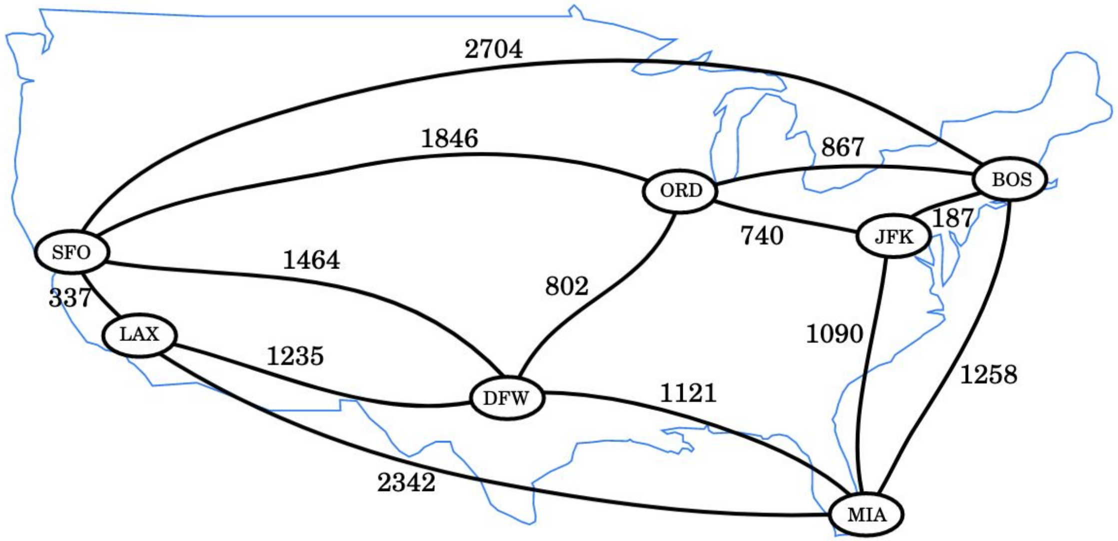


Dijkstra's algorithm

- graph search algorithm that finds the shortest path between nodes in a weighted graph

- maintains a set of vertices whose shortest distance from the *source* has already been determined, which it gradually refines
  - uses a *min heap* to select the vertex with the smallest distance

Dijkstra's algorithm



1. init:
   a. assign a init distance for each node
      i. 0 for src, INF for all other nodes
   b. create a min-heap and add the source

2. while heap is non-empty:
   a. poll node p and mark as visited
   b. For each neighboring node not yet visited:
      i. distance of neighbor = dist(p) + weight of edge (p, neighbor)
      ii. If this distance is less than the current dist, update it.
      iii. insert in heap if distance changed