Exploration: Graphs

Introduction

A graph is a structure representing a collection of objects or states, where some pairs of those objects are related or connected in some way.

Graphs are used in lots and lots of places in computer science:

- Social networks like Facebook or Twitter
- Computer graphics
- Machine learning
- Computer vision
- Logistics and optimization
- Computer networking

Graphs are made up of vertices, representing things in the graph, and edges, representing the connections between those things.

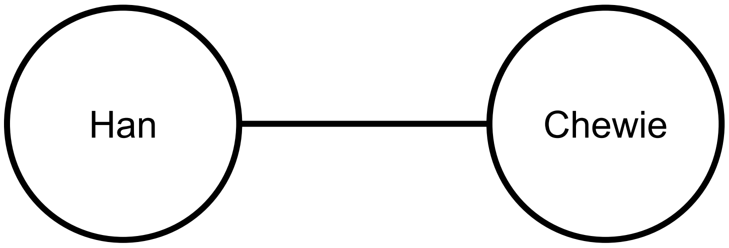
Graph Components

Vertices represent objects, states (i.e. conditions or configurations), locations, etc. These form a set where each vertex is unique (i.e. no two vertices represent the same object/state): $V = \{v_1, v_2, v_3, \ldots, v_n\}$

Edges represent relationships or connections between vertices.

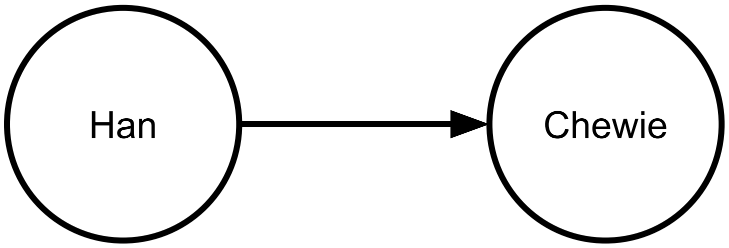
- These are represented as vertex pairs: $E = \{(v_i, v_j), \ldots\}$
- Edges can be directed or undirected.
- If there is an edge between v_i and v_j, then v_i and v_j are said to be adjacent (or they are neighbors).
- Edges can be weighted or unweighted. Weighted edges represent something about that edge. For example, it might represent the distance between vertices.

An undirected edge is like a friend relationship in Facebook; e.g., if Han and Chewie are friends, there would be an undirected edge between them in the Facebook graph:



A relationship between two friends is undirected

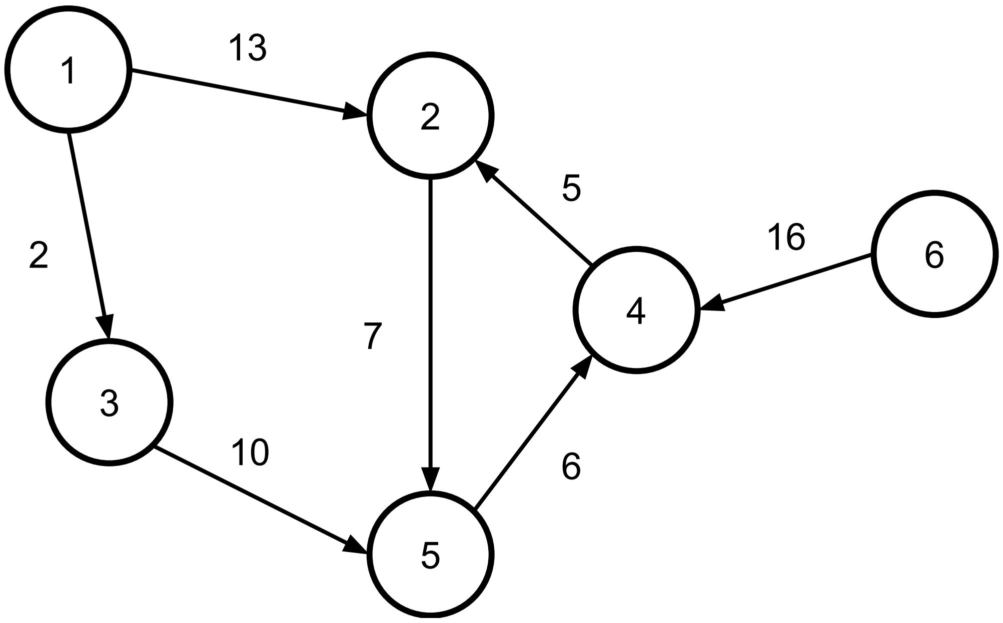
A directed edge is like a follows relationship in Twitter; e.g., if Han follows Chewie, there would be a directed edge between them in the Twitter graph:



A directed relationship between a Twitter user and their follower

- Here, we say the edge is directed from Han to Chewie.
- We can also say that Han is the head of this edge and that Chewie is its tail.
- We can also say that Chewie is a direct successor of Han and that Han is a direct predecessor of Chewie.
- We can also say that Chewie is reachable from Han.

Here's an example of a small graph with 6 vertices and 7 directed, weighted edges:



A graph with 6 vertices and 7 directed and weighted edges connecting them

Graphs represent general relationships between objects.

- A node may have connections to any number of other nodes. An undirected graph is connected if all vertices are reachable from all other vertices. A directed graph is strongly connected if all vertices are reachable from all other vertices.
- There can be multiple paths (or no path) from one node to another. A path in an undirected graph is a sequence of vertices, where each adjacent pair of vertices are adjacent in the graph;

informally, we can also think of a path as a sequence of edges. Also, it visits each vertex at most once. A directed path is a sequence of directed edges in the graph.

- A component of an undirected graph is an induced subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the rest of the graph. A vertex with no incident edges is itself a component. A graph that is itself connected has exactly one component, consisting of the whole graph. Components are also sometimes called connected components.
- There can be cycles in the graph. A cycle is a closed walk that enters and leaves each vertex at most once. An undirected graph is acyclic if no subgraph is a cycle; acyclic graphs are also called forests. A directed graph is acyclic if it does not contain a directed cycle; directed acyclic graphs are often called dags.

We have actually already dealt with many graphs. Trees are a special, more restricted subclass of graphs where there cannot be cycles or multiple paths to a node.

There are lots of different kinds of questions we might want to ask about a graph:

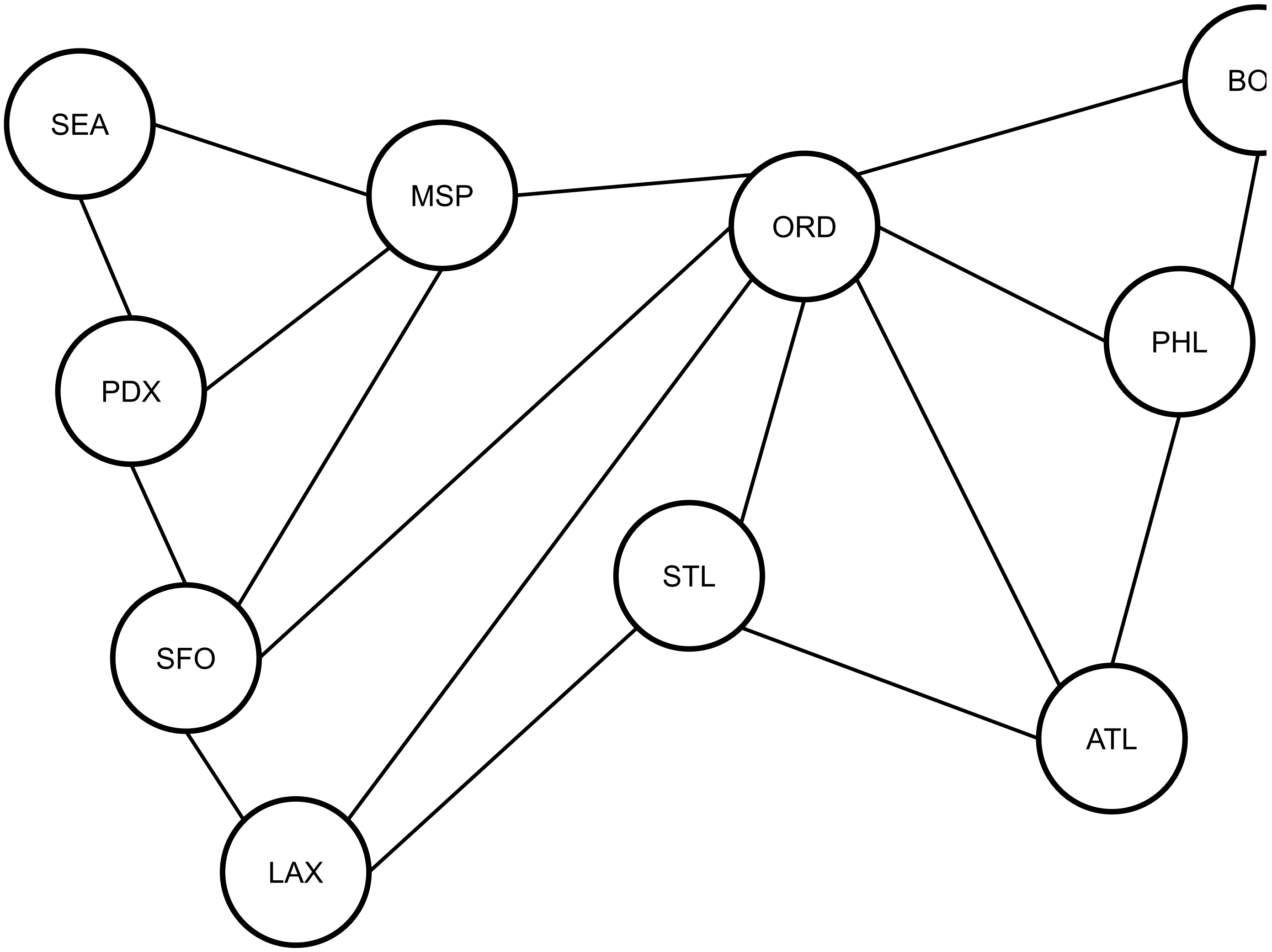
- Is node X in the graph?
- Is node Y reachable from node X?
- What nodes are reachable from node X?
- Are X and Y adjacent nodes?
- What's the shortest path from node X to node Y?
- How many edges are there between node A and node Y?
- How many connected components are there in the graph?

Representing Graphs

There are two main ways to represent a graph in practice:

- An adjacency list, in which the each vertex stores a list of its adjacent vertices.
- An adjacency matrix, which is a two-dimensional matrix whose rows and columns represent vertices. If there is an edge between v_i and v_j, the value at location *(i, j)* in the matrix will be non-zero.

Let's consider this graph as an example, where flights between US airports are represented:



Graph of airport locations in the US

As an adjacency list, this graph would look like this:

```
ATL: [ORD, PHL, STL],
BOS: [ORD, PHL],
LAX: [ORD, SFO, STL],
MSP: [ORD, PDX, SEA, SFO],
ORD: [ATL, BOS, LAX, MSP, PHL, SFO, STL],
PDX: [MSP, SEA, SFO],
PHL: [ATL, BOS, ORD],
SEA: [MSP, PDX],
SFO: [LAX, MSP, ORD, PDX],
STL: [ATL, LAX, ORD]
```

As an adjacency matrix, the graph would look like this:

	ATL	BOS	LAX	MSP	ORD	PDX	PHL	SEA	SFO	STL
ATL	0	0	0	0	1	0	1	0	0	1
BOS	0	0	0	0	1	0	1	0	0	0
LAX	0	0	0	0	1	0	0	0	1	1
MSP	0	0	0	0	1	1	0	1	1	0

	ATL	BOS	LAX	MSP	ORD	PDX	PHL	SEA	SFO	STL
ORD	1	1	1	1	0	0	1	0	1	1
PDX	0	0	0	1	0	0	0	1	1	0
PHL	1	1	0	0	1	0	0	0	0	0
SEA	0	0	0	1	0	1	0	0	0	0
SFO	0	0	1	1	1	1	0	0	0	0
STL	1	0	1	0	1	0	0	0	0	0

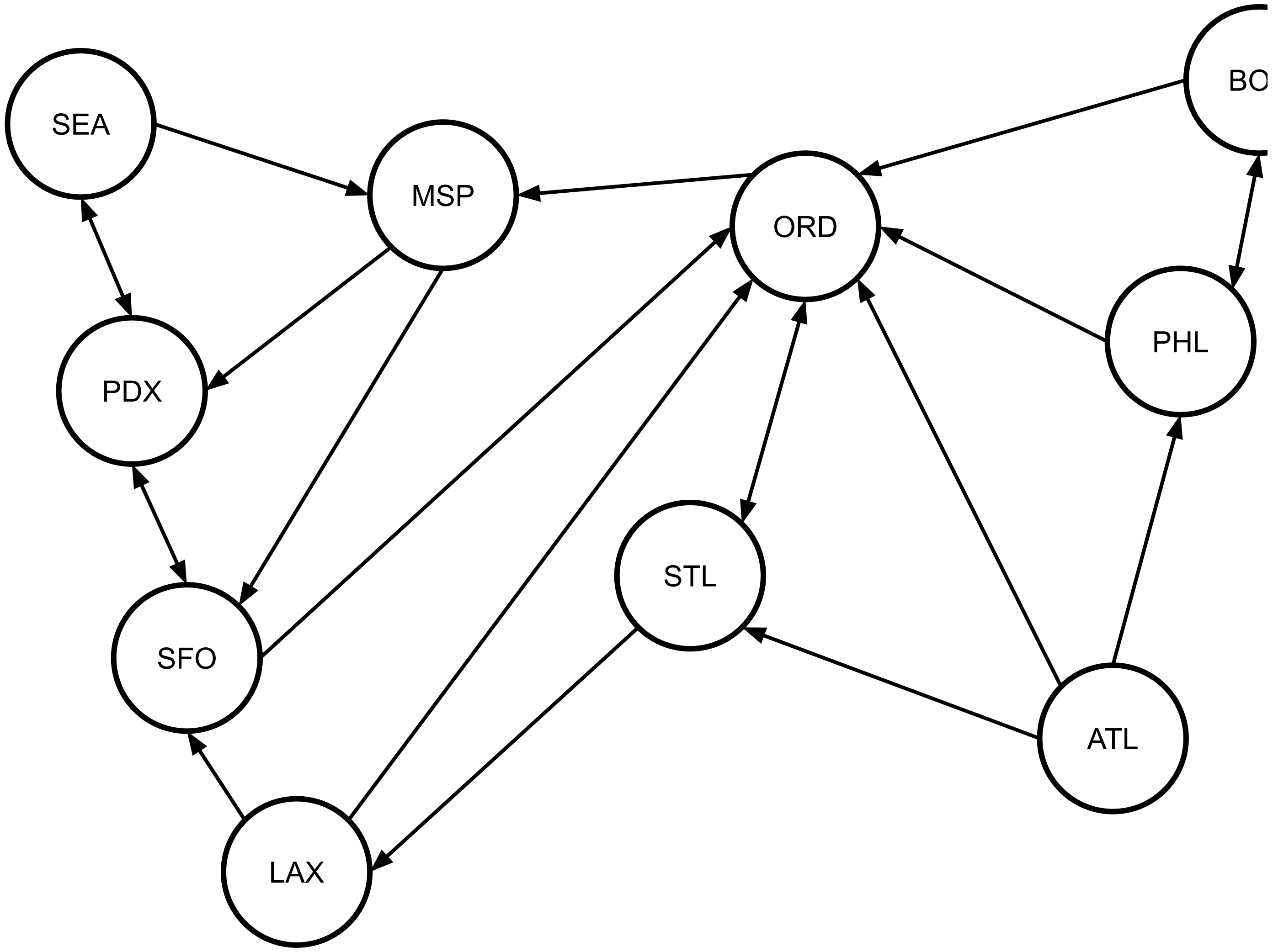
Note that this matrix is symmetric.

What is the space complexity of each of these representations?

- Adjacency list: O(|V| + |E|)
- Adjacency matrix: O(|V|^2)

Thus, the adjacency list is more space efficient when the graph is sparse (i.e., when it has relatively few edges). What if our graph is a directed graph, e.g., if we have a flight from airport A to airport B but not a return flight?

Each of these representations can still be used. For example, say we have this graph:



Graph of airport locations in the US with one way flights

Now, our adjacency list would look like this:

```
ATL: [ORD, PHL, STL],
BOS: [ORD, PHL],
LAX: [ORD, SFO],
MSP: [PDX, SFO],
ORD: [MSP, STL],
PDX: [SEA, SFO],
PHL: [BOS, ORD],
SEA: [MSP, PDX],
SFO: [ORD, PDX],
STL: [LAX, ORD]
```

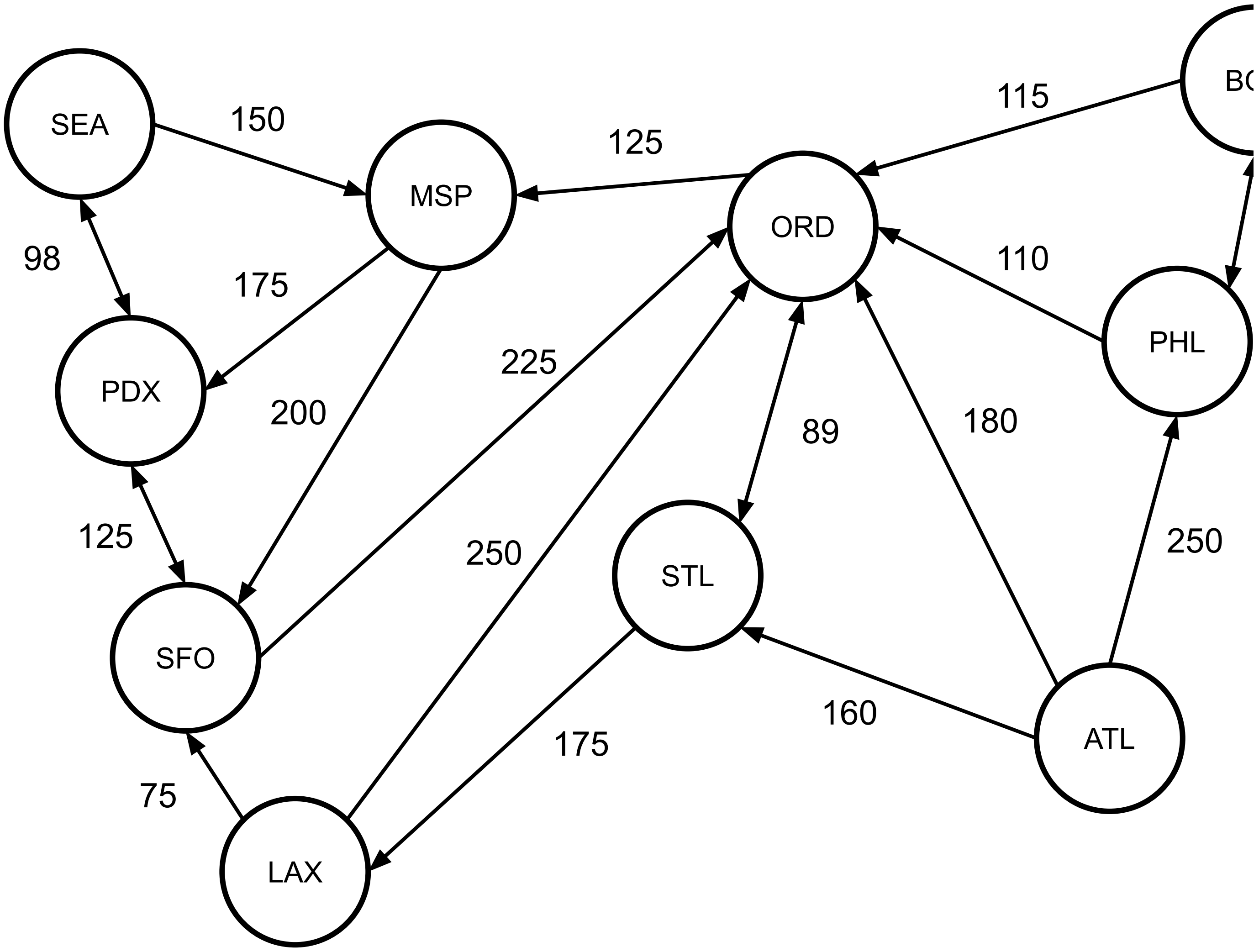
Here, each vertex lists only the edges going **from** it.

The adjacency matrix for this graph would look like this:

	ATL	BOS	LAX	MSP	ORD	PDX	PHL	SEA	SFO	STL
ATL	0	0	0	0	1	0	1	0	0	1
BOS	0	0	0	0	1	0	1	0	0	0
LAX	0	0	0	0	1	0	0	0	1	0
MSP	0	0	0	0	0	1	0	0	1	0
ORD	0	0	0	1	0	0	0	0	0	1
PDX	0	0	0	0	0	0	0	1	1	0
PHL	0	1	0	0	1	0	0	0	0	0
SEA	0	0	0	1	0	1	0	0	0	0
SFO	0	0	0	0	1	1	0	0	0	0
STL	0	0	1	0	1	0	0	0	0	0

Note that this matrix is no longer symmetric.

We can go one step further with each representation, incorporating weights. Say our graph contains the costs of flights between cities:



Graph of airport locations in the US with one way flights and associated costs as weights

Now, our adjacency list would store the weights/costs along with the edges:

```
ATL: [{ORD: 180}, {PHL: 250}, {STL: 160}],
BOS: [{ORD: 115}, {PHL: 69}],
LAX: [{ORD: 250}, {SFO: 75}],
MSP: [{PDX: 175}, {SFO: 200}],
ORD: [{MSP: 125}, {STL: 89}],
PDX: [{SEA: 98}, {SFO: 125}],
PHL: [{BOS: 69}, {ORD: 110}],
SEA: [{MSP: 150}, {PDX: 98}],
SFO: [{ORD: 225}, {PDX: 125}],
STL: [{LAX: 175}, {ORD: 89}]
```

The adjacency matrix for this graph would now hold these weights/costs instead of just binary values:

We could also use a special value here (e.g., –1) to indicate there is no edge if we wanted to allow free flights. This generally wraps up the options we have to store graphs. The next exploration will look at how we might go about answering questions, modifying, and generally working with graphs.

	ATL	BOS	LAX	MSP	ORD	PDX	PHL	SEA	SFO	STL
ATL	0	0	0	0	180	0	250	0	0	160
BOS	0	0	0	0	115	0	69	0	0	0
LAX	0	0	0	0	250	0	0	0	75	0
MSP	0	0	0	0	0	175	0	0	200	0
ORD	0	0	0	125	0	0	0	0	0	89
PDX	0	0	0	0	0	0	0	98	125	0
PHL	0	69	0	0	110	0	0	0	0	0
SEA	0	0	0	150	0	98	0	0	0	0
SFO	0	0	0	0	225	125	0	0	0	0
STL	0	0	175	0	89	0	0	0	0	0