CS 261-020
Data Structures

Lecture 15

Graphs

3/7/24, Thursday

Oregon State University

Graph

- Graph – a collection of objects or states, where some pairs of those objects are related or connected in some way.

- Graphs examples in computer science:
    - Social networks like Facebook or Twitter
    - Computer graphics
    - Machine learning
    - Computer vision
    - Logistics and optimization
    - Computer networking

Graph

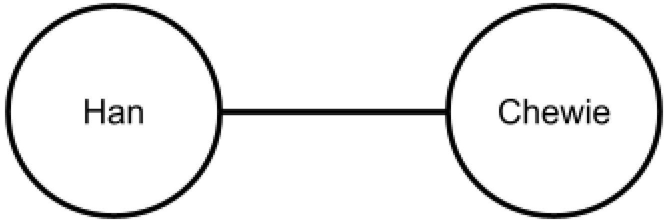
- A graph is composed of vertices (or nodes or points) and edges (or arcs or lines).

- Vertices represent objects, states (i.e. conditions or configurations), locations, etc.
  - Form a set where each vertex is unique: $V = \{v_1, v_2, v_3, \ldots, v_n\}$

Graph

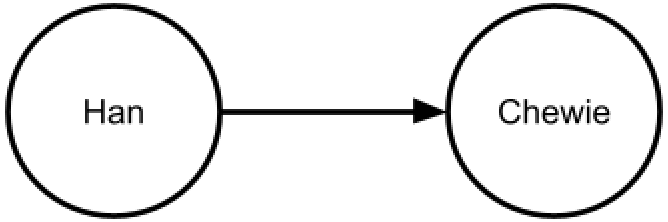
- Edges represent relationships or connections between vertices.
  - These are represented as vertex pairs: $E = \{(v_i, v_j), \ldots\}$
  - Edges can be directed or undirected.
    - If there is an edge between v_i and v_j, then v_i and v_j are said to be adjacent (or they are neighbors).
    - Edges can be weighted or unweighted.

- An undirected edge is like a friend relationship in Facebook:



Graph

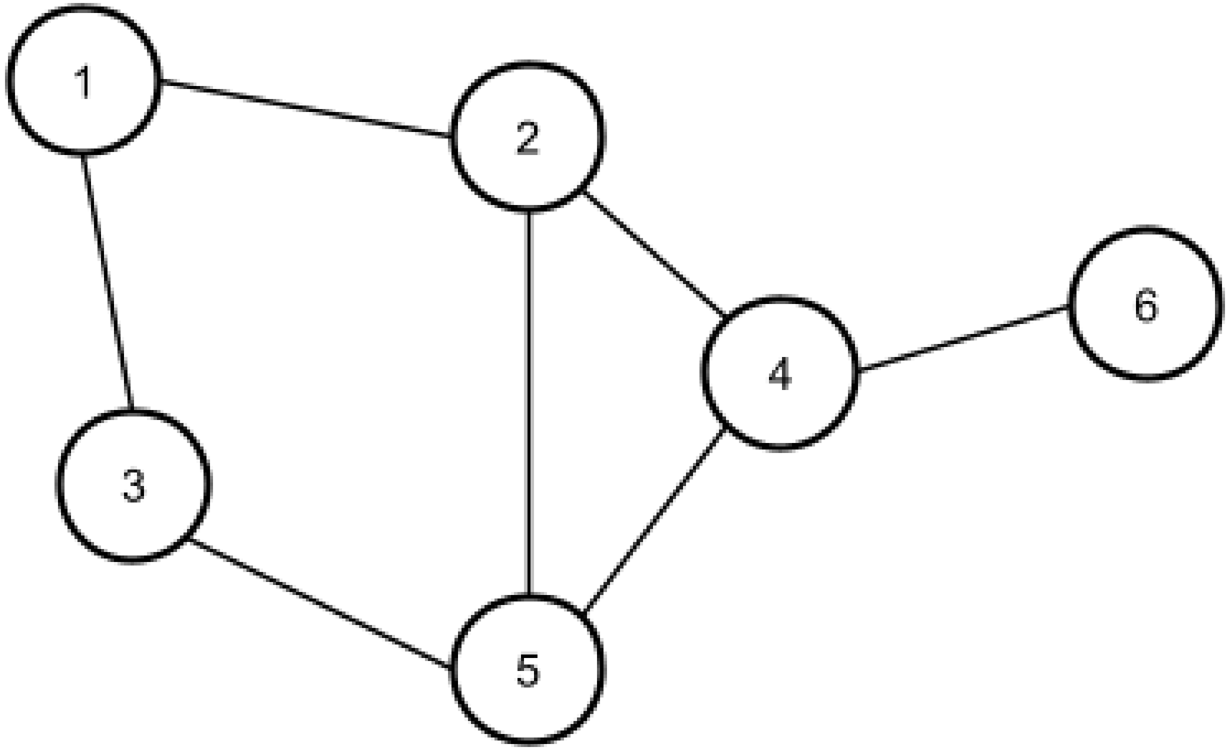
- A directed edge is like a “follows” relationship in Twitter,



  - The edge is directed from Han to Chewie.
  - Han is the head of this edge and that Chewie is its tail.
  - Chewie is a direct successor of Han and that Han is a direct predecessor of Chewie.
  - Chewie is reachable from Han.

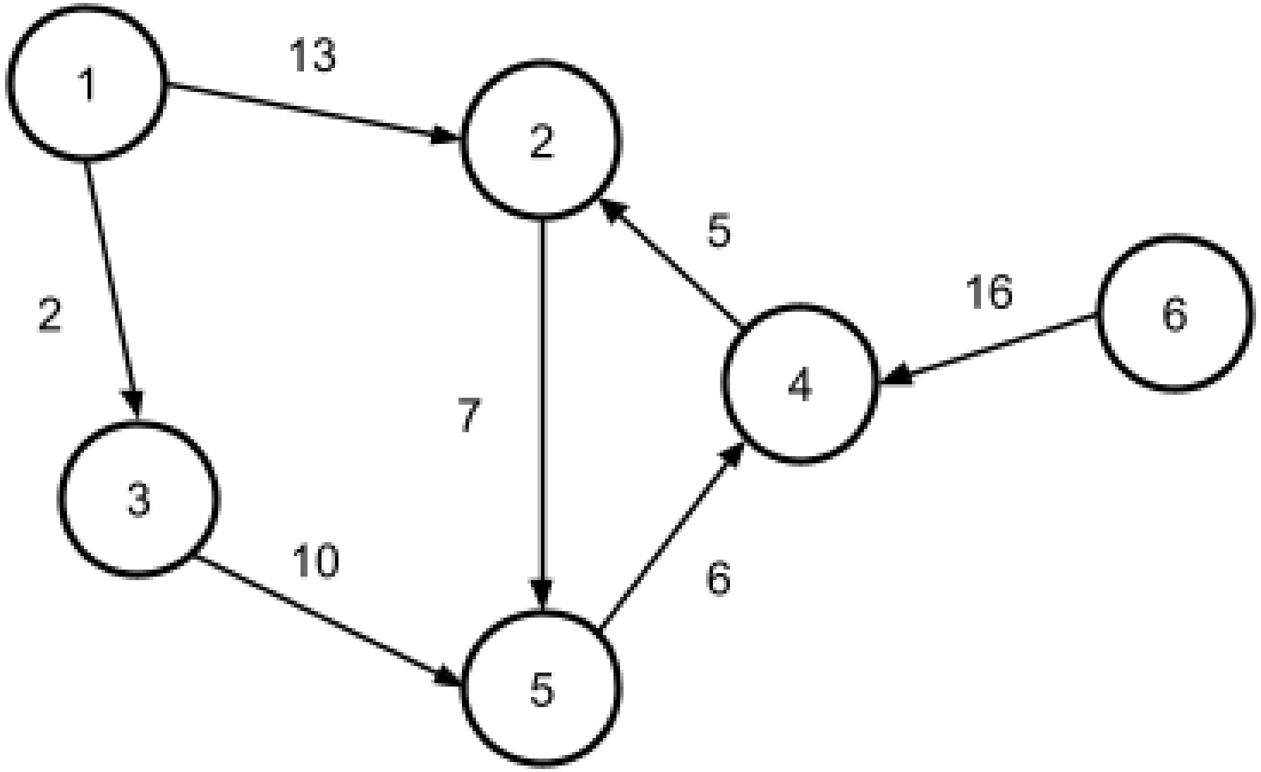
Graph

- An example graph with 6 vertices and 7 undirected, unweighted edges:



Graph

- An example of a similar graph with directed, weighted edges:



Graph

- Graphs represent general relationships between objects.
  - A node may have connections to any number of other nodes.
  - There can be multiple paths (or no path) from one node to another.
  - There can be cycles (loops) in the graph, where there is a path from one node back to itself.

- Trees are a special, more restricted subclass of graphs.

Graph

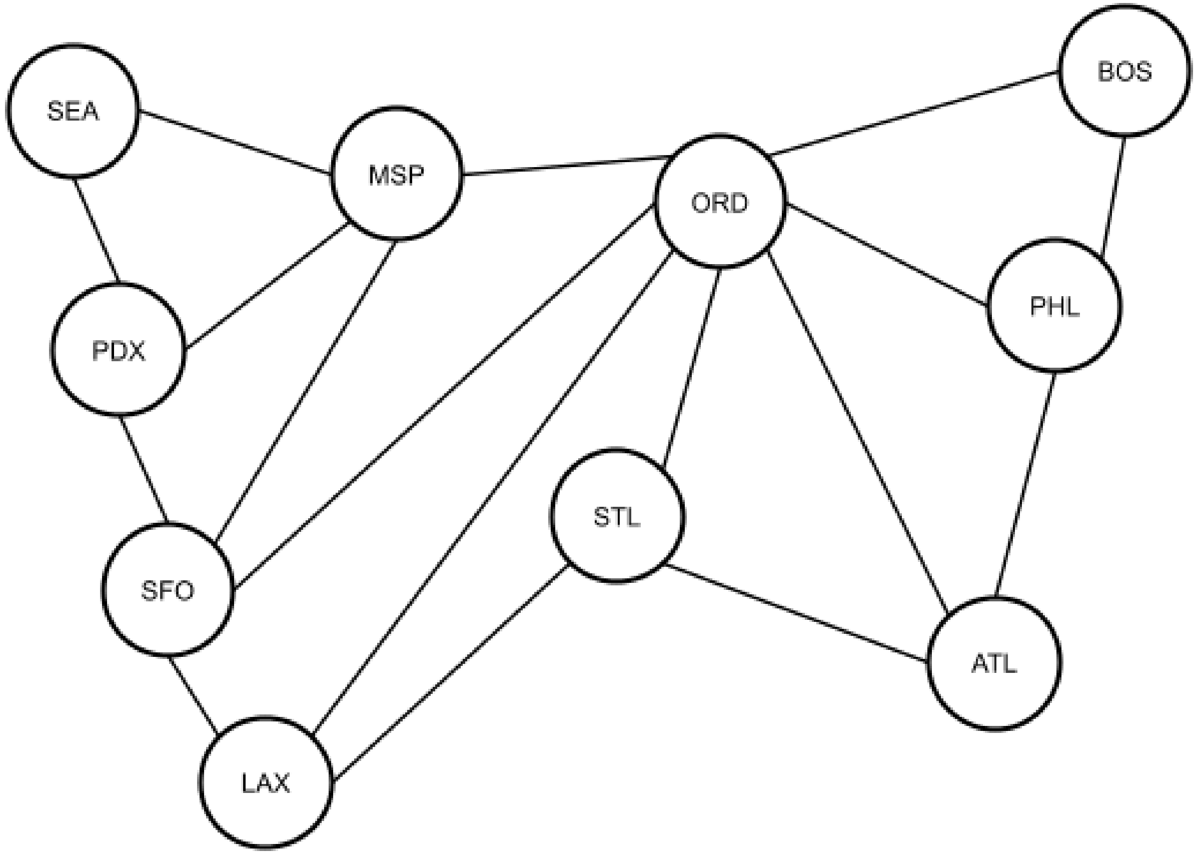
- Questions we might want to ask about a graph:
  - Is X in the graph?
  - Is Y reachable from X?
  - What nodes are reachable from X?
  - Are X and Y adjacent?
  - What’s the shortest path from X to Y?
  - How many edges between A and Y?

Representing Graphs

- Two main ways to represent a graph in practice:
  - An **adjacency list:** each vertex stores a list of its adjacent vertices.
  - An **adjacency matrix**: 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.

Representing Graphs

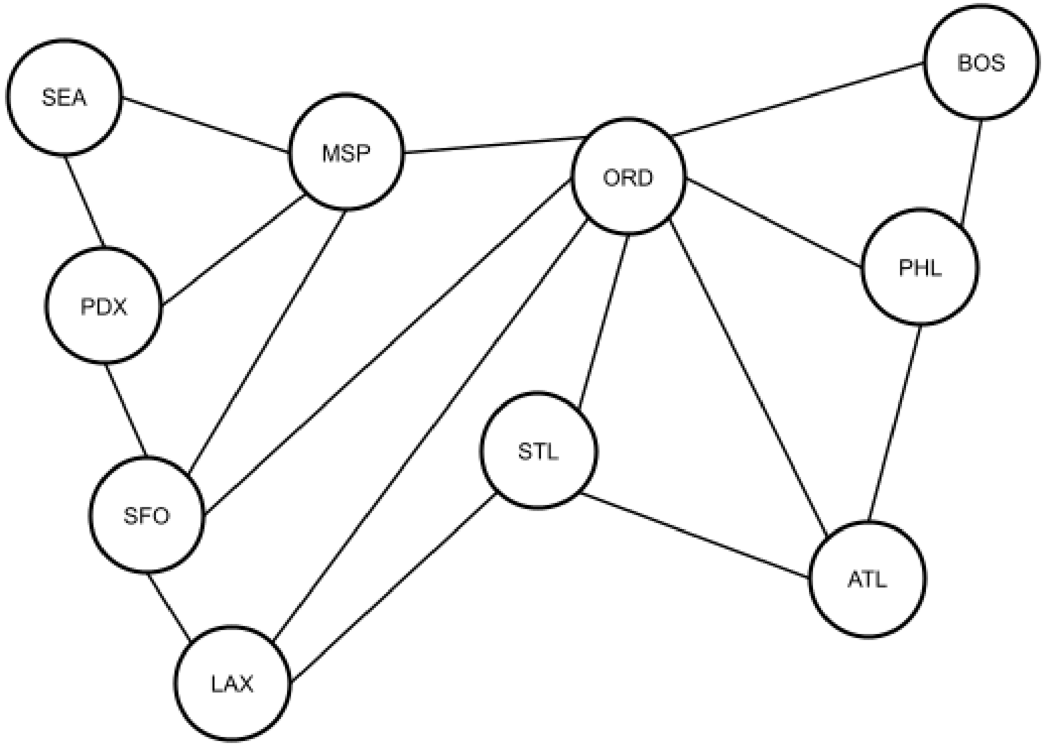
- Consider this graph, where flights between US airports are represented, as an example:



Representing Graphs

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

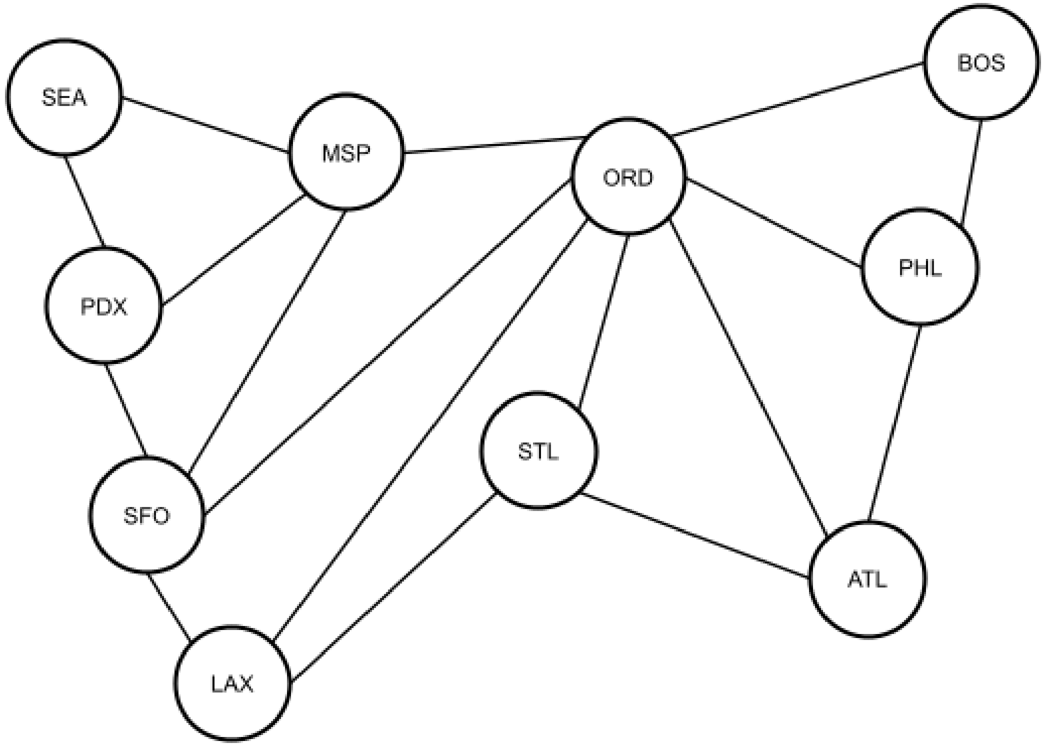
```
ATL: [ORD, PHL, STL],
BOS: [ORD, PHL],
LAX: [ORD, SFO, STL],
MSP: [ORD, PDX, SEA, SFO],
ORD: [ATL, BOS, LAX, MSP, PHL, SFO, STL],
PDX: [MSP, SEA, SFO],
PHL: [ATL, BOS, ORD],
SEA: [MSP, PDX],
SFO: [LAX, MSP, ORD, PDX],
STL: [ATL, LAX, ORD]
```



Representing Graphs

- 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
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.

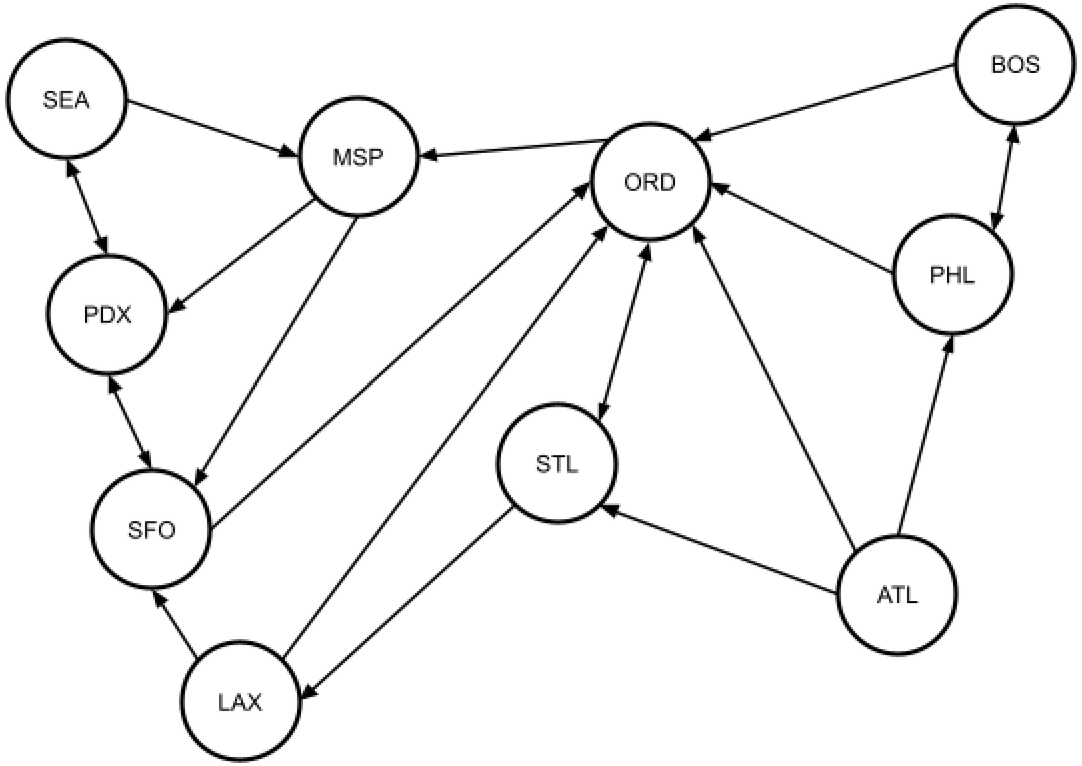
Representing Graphs

- 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.

Representing Graphs

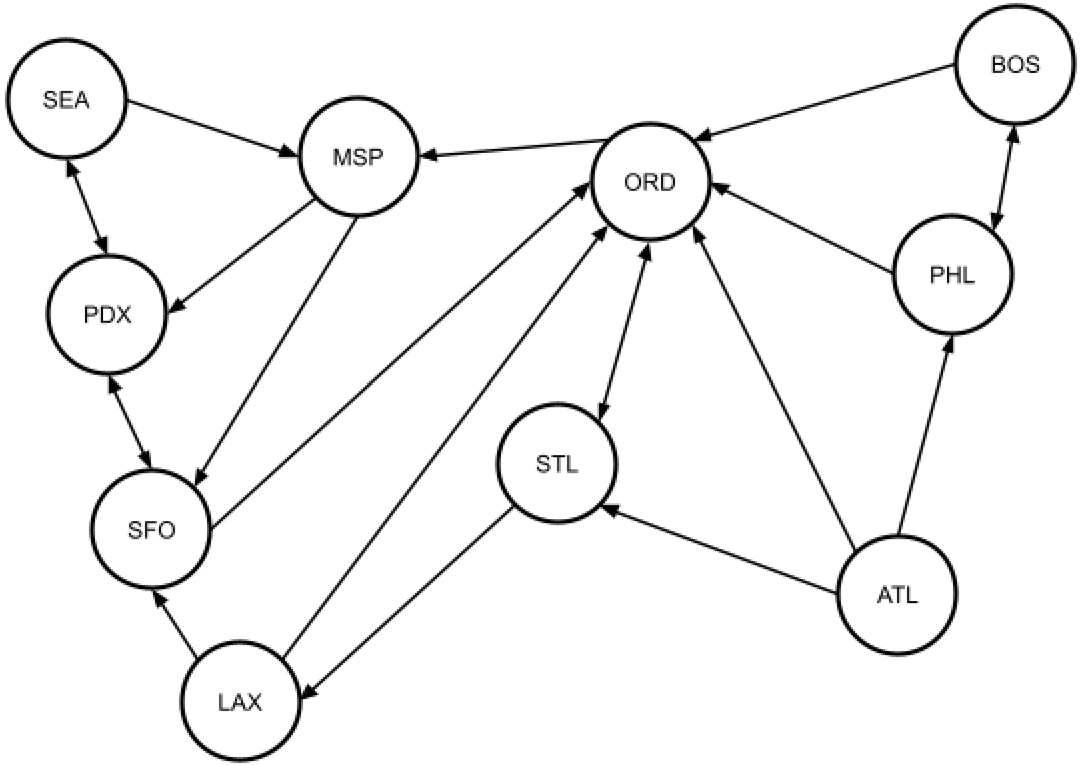
- What if our graph is a directed graph, e.g. if we have a flight from airport A to airport B but not a return flight?
- Each of these representations can still be used. For example, say we have this graph:



Representing Graphs

- The adjacency list:

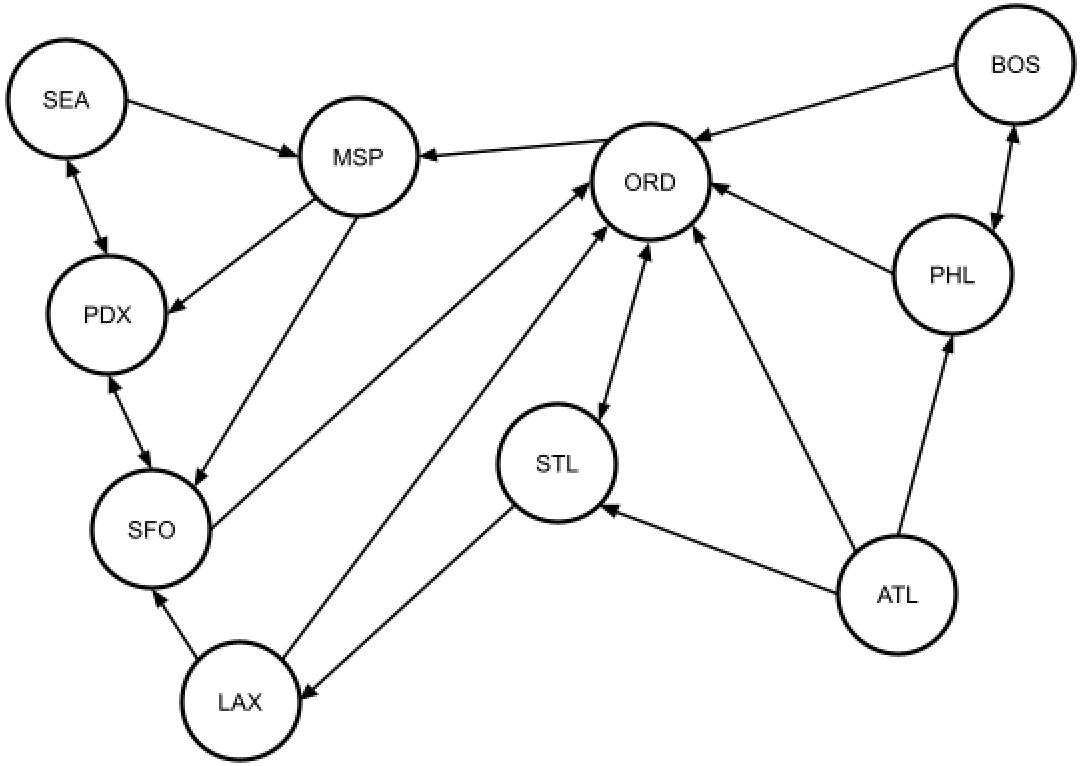
```
ATL: [ORD, PHL, STL],
BOS: [ORD, PHL],
LAX: [ORD, SFO],
MSP: [PDX, SFO],
ORD: [MSP, STL],
PDX: [SEA, SFO],
PHL: [BOS, ORD],
SEA: [MSP, PDX],
SFO: [ORD, PDX],
STL: [LAX, ORD]
```



Representing Graphs

- The adjacency matrix for this graph:

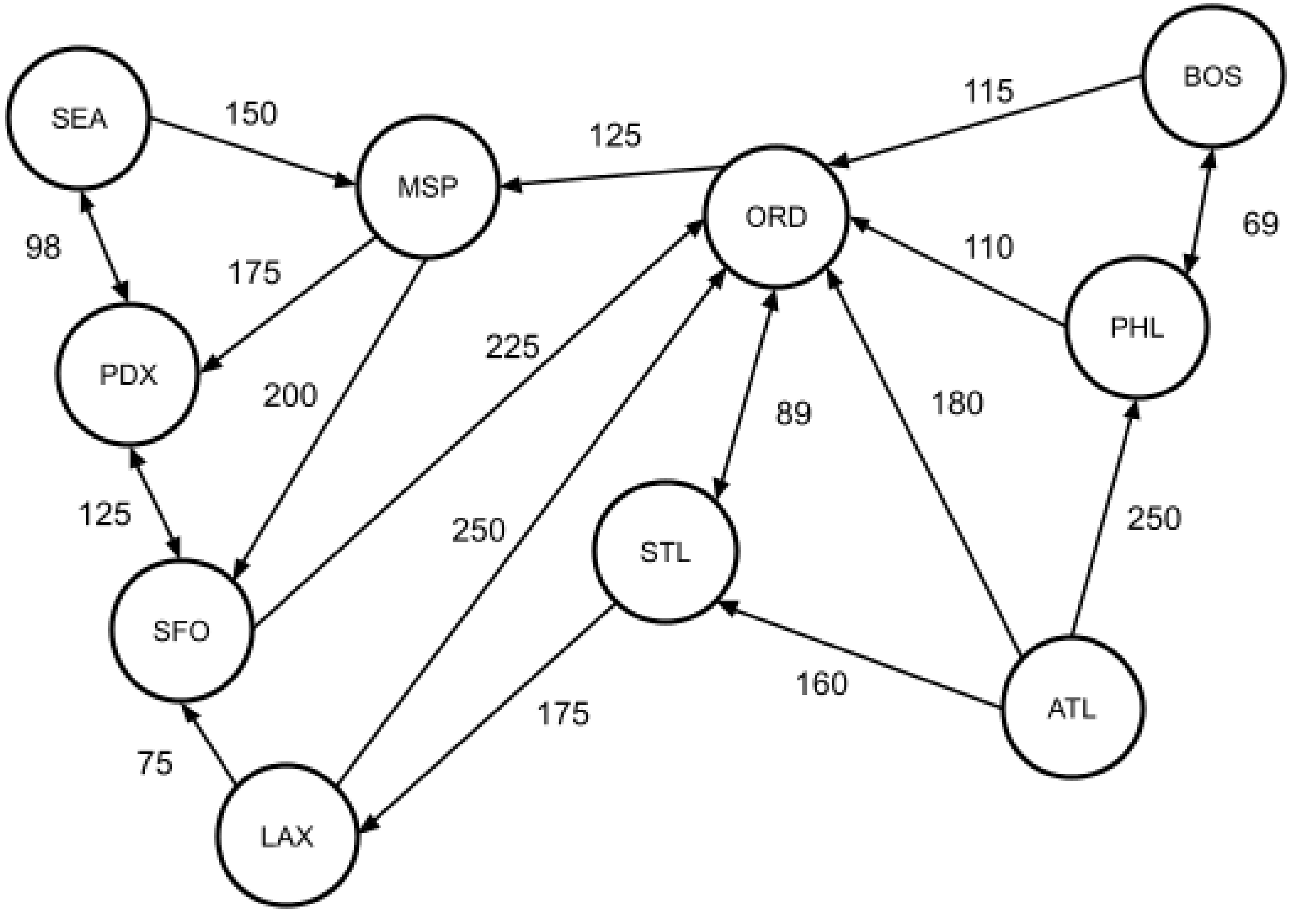
	ATL	BOS	LAX	MSP	ORD	PDX	PHL	SEA	SFO	STL
ATL	0	0	0	0	1	0	1	0	0	1
BOS	0	0	0	0	1	0	1	0	0	0
LAX	0	0	0	0	1	0	0	0	1	0
MSP	0	0	0	0	0	1	0	0	1	0
ORD	0	0	0	1	0	0	0	0	0	1
PDX	0	0	0	0	0	0	0	1	1	0
PHL	0	1	0	0	1	0	0	0	0	0
SEA	0	0	0	1	0	1	0	0	0	0
SFO	0	0	0	0	1	1	0	0	0	0
STL	0	0	1	0	1	0	0	0	0	0



- Note that this matrix is no longer symmetric.

Representing Graphs

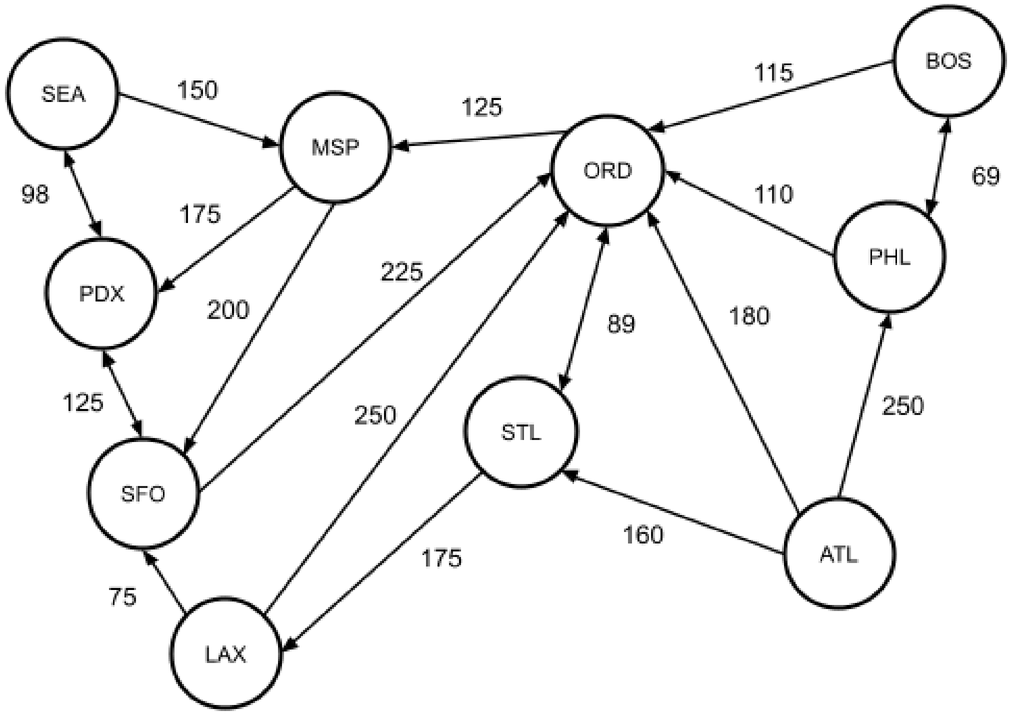
- Adding weights to the graph. Say our graph contains the costs of flights between cities:



Representing Graphs

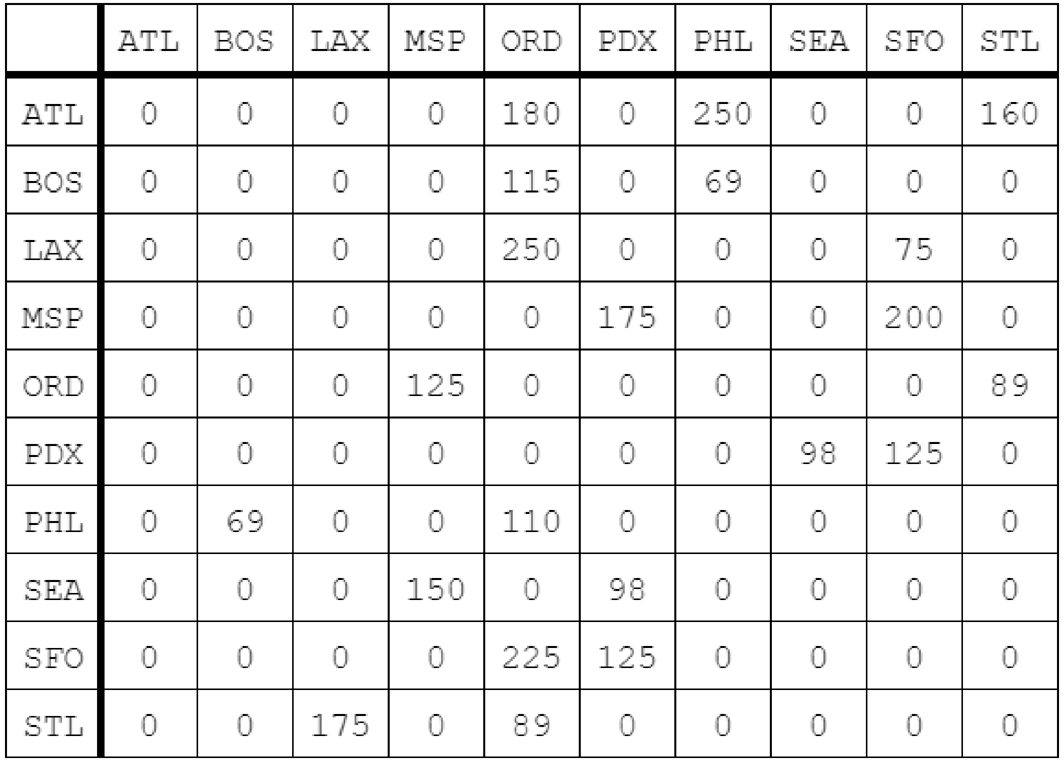
- The adjacency list would store the weights/costs along with the edges:

```
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}]
```

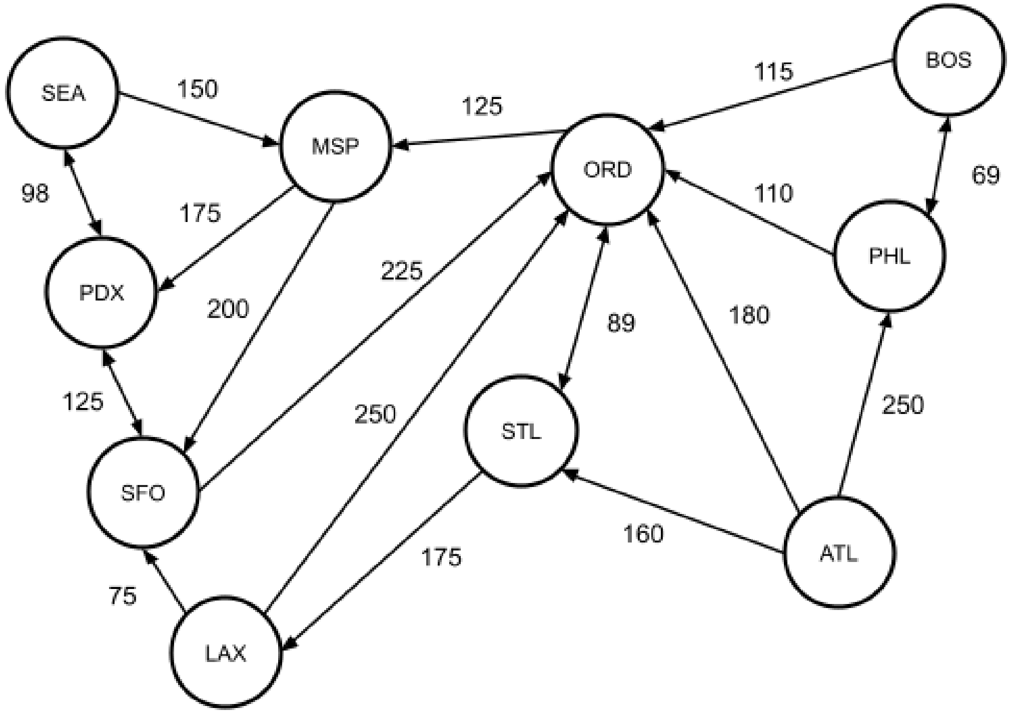


Representing Graphs

- The adjacency matrix would hold these weights/costs instead of binary values:



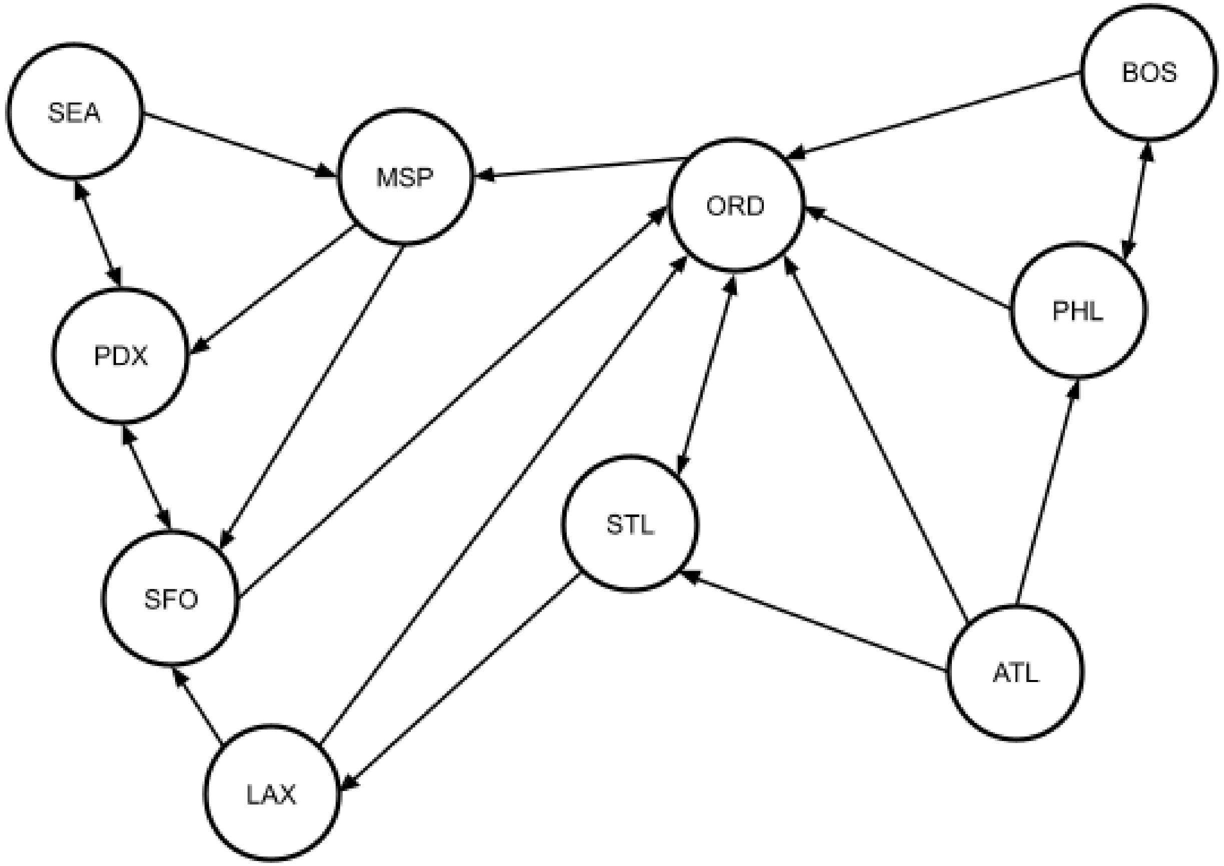
	ATL	BOS	LAX	MSP	ORD	PDX	PHL	SEA	SFO	STL
ATL	0	0	0	0	180	0	250	0	0	160
BOS	0	0	0	0	115	0	69	0	0	0
LAX	0	0	0	0	250	0	0	0	75	0
MSP	0	0	0	0	0	175	0	0	200	0
ORD	0	0	0	125	0	0	0	0	0	89
PDX	0	0	0	0	0	0	0	98	125	0
PHL	0	69	0	0	110	0	0	0	0	0
SEA	0	0	0	150	0	98	0	0	0	0
SFO	0	0	0	0	225	125	0	0	0	0
STL	0	0	175	0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



- We could also use a special value here (e.g. -1) to indicate there is no edge.

Single Source Reachability

- Question: what nodes are reachable from some specific node?
- For example, what airports are reachable from PDX?



Single Source Reachability

- Algorithm to find reachable vertices from some vertex v_i:
  1. Initialize an empty set of reachable vertices.
  2. Initialize an empty stack. Add v_i to the stack.
  3. If the stack is not empty, pop a vertex v from the stack.
  4. If v is not in the set of reachable vertices:
     - Add it to the set of reachable vertices.
     - Add each vertex that is direct successor of v to the stack.
  5. Repeat from 3.

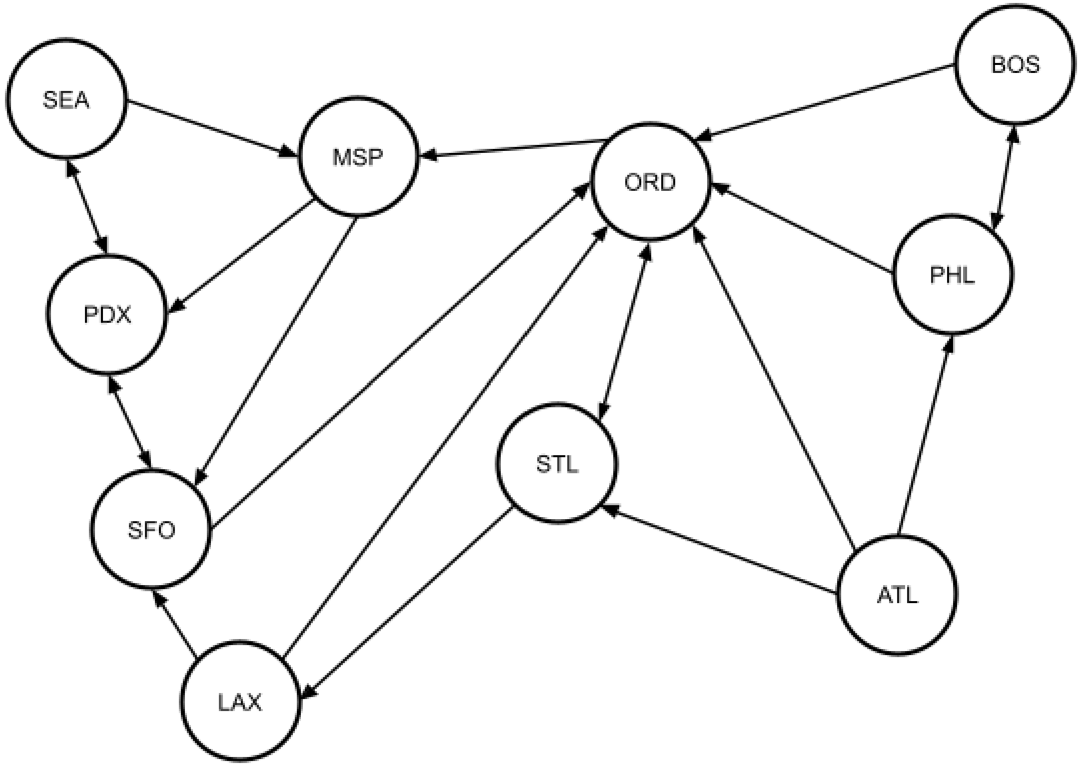
Single Source Reachability

- Looking for airports reachable from PDX would look like this:

```
1.reachable: {}
  stack: [PDX]

2.v: PDX
  successors: [SEA, SFO]
  reachable: {PDX}
  stack: [SEA, SFO]

3.v: SFO
  successors: [ORD, PDX]
  reachable: {PDX, SFO}
  stack: [SEA, ORD, PDX]
```



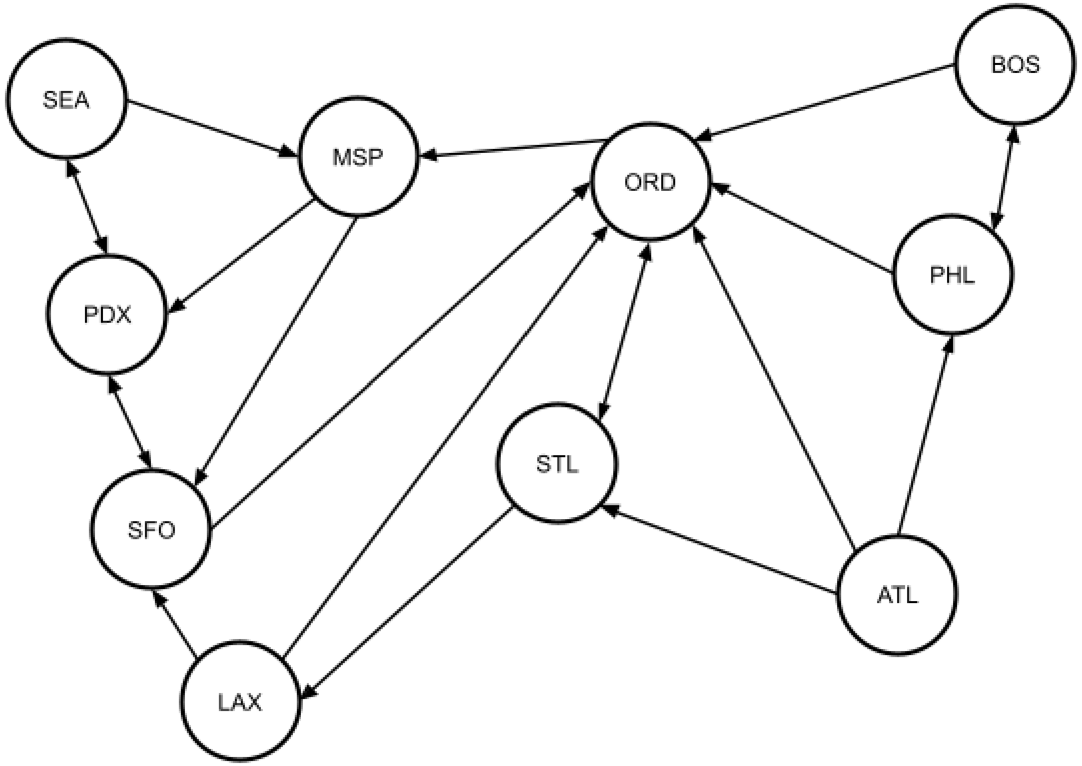
Single Source Reachability

- Looking for airports reachable from PDX would look like this:

```
4. v: PDX (already reachable)
   successors: --
   reachable: {PDX, SFO}
   stack: [SEA, ORD]

5. v: ORD
   successors: [MSP, STL]
   reachable: {ORD, PDX, SFO}
   stack: [SEA, MSP, STL]

6. v: STL
   successors: [LAX, ORD]
   reachable: {ORD, PDX, SFO, STL}
   stack: [SEA, MSP, LAX, ORD]
```



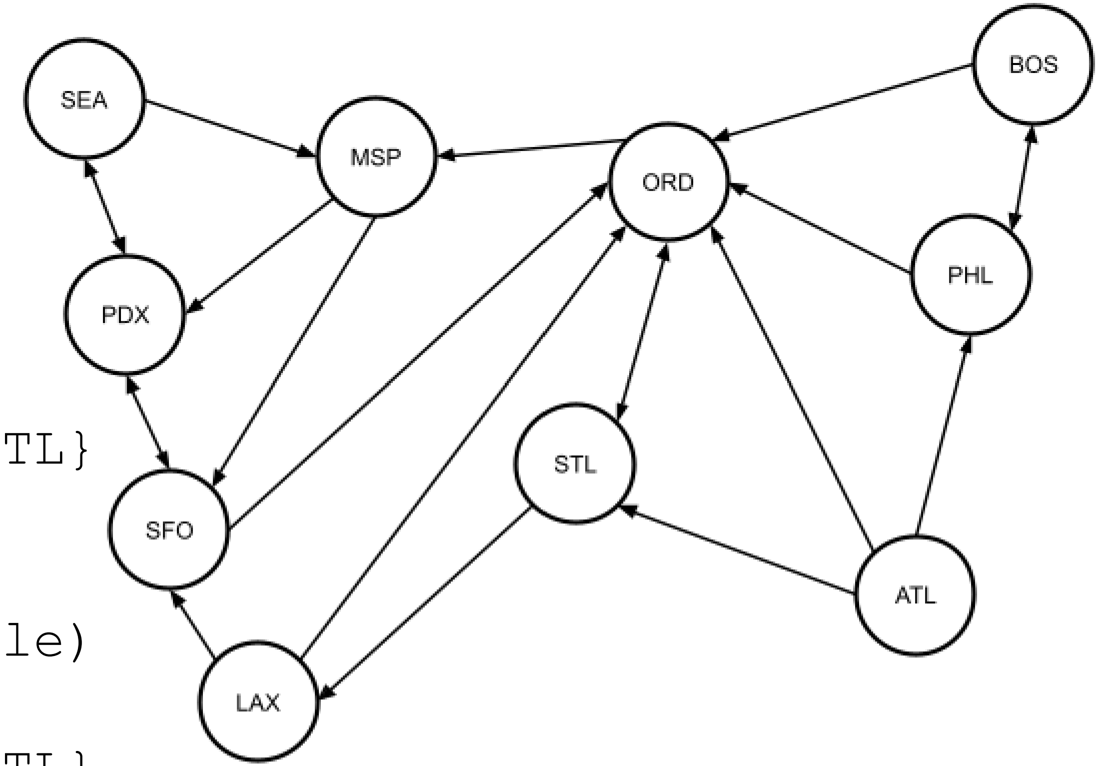
Single Source Reachability

- Looking for airports reachable from PDX would look like this:

```
7. v: ORD (already reachable)
   successors: --
   reachable: {ORD, PDX, SFO, STL}
   stack: [SEA, MSP, LAX]

8. v: LAX
   successors: [ORD, SFO]
   reachable: {LAX, ORD, PDX, SFO, STL}
   stack: [SEA, MSP, ORD, SFO]

9. v: SFO, ORD (both already reachable)
   successors: --
   reachable: {LAX, ORD, PDX, SFO, STL}
   stack: [SEA, MSP]
```



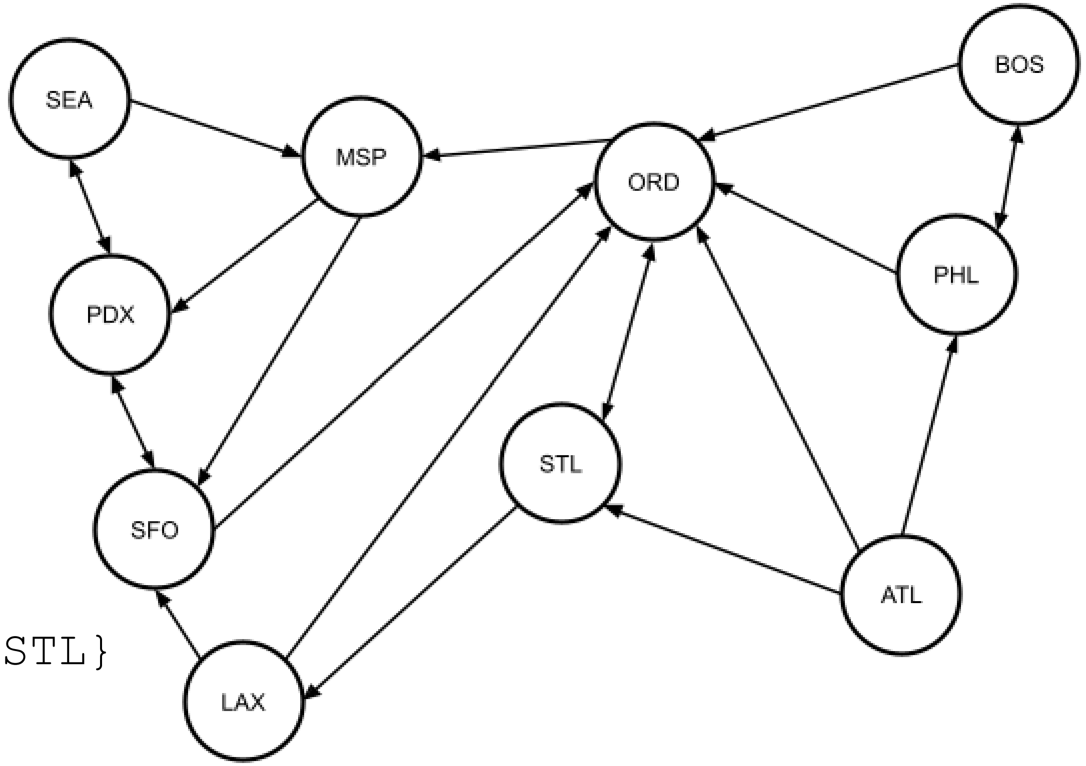
Single Source Reachability

- Looking for airports reachable from PDX would look like this:

```
10. v: MSP
    successors: [PDX, SFO]
    reachable: {LAX, MSP, ORD, PDX, SFO, STL}
    stack: [SEA, PDX, SFO]

11. v: SFO, PDX (both already reachable)
    successors: --
    reachable: {LAX, MSP, ORD, PDX, SFO, STL}
    stack: [SEA]

12. v: SEA
    successors: MSP, PDX
    reachable: {LAX, MSP, ORD, PDX, SEA, SFO, STL}
    stack: [MSP, PDX]
```

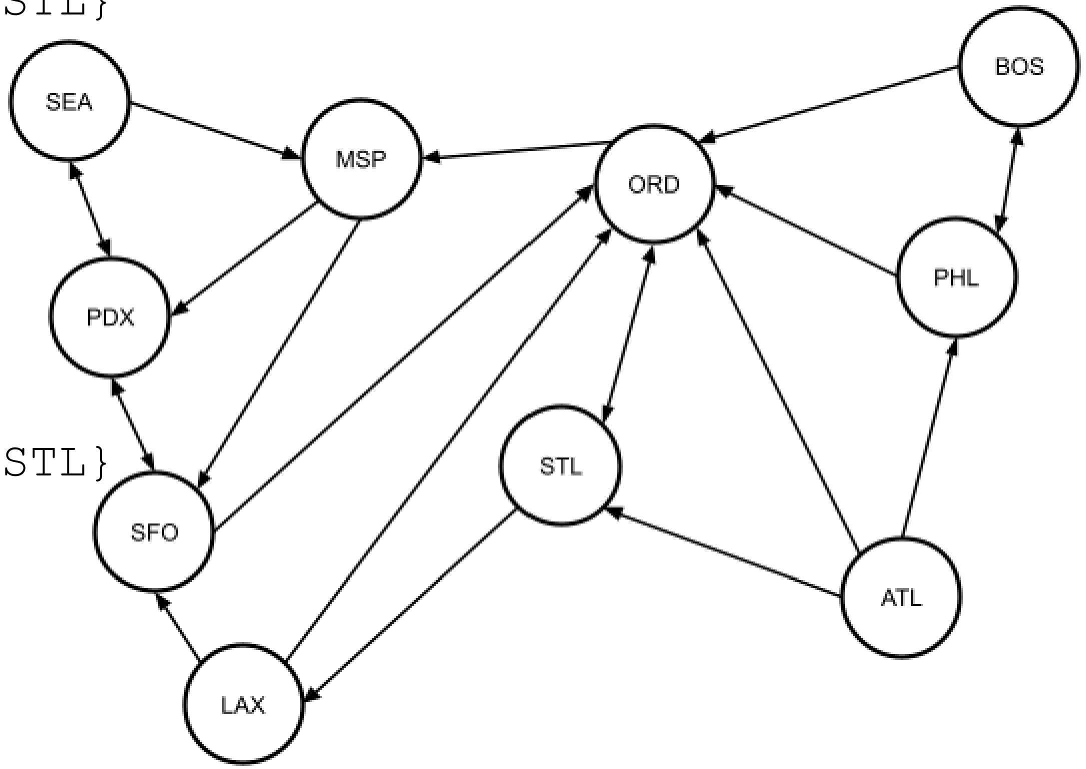


Single Source Reachability

- Looking for airports reachable from PDX would look like this:

```
13. v: PDX, MSP (both already reachable)
    Successors: --
    reachable: {LAX, MSP, ORD, PDX, SEA, SFO, STL}
    stack: []


14. Done (stack empty)
    reachable: {LAX, MSP, ORD, PDX, SEA, SFO, STL}
```



Single Source Reachability

- This algorithm can be implemented using either the adjacency list representation or the adjacency matrix representation.

- We could also use a queue instead of a stack.
  - Result in a different order of exploration of the graph.

Depth-first Search and Breadth-first Search

- The reachability algorithm we saw was an instance of depth-first search (or DFS).

- Recall: DFS: exploring a tree where we travel a particular path as far as we can before trying another path.
    - In other words, in DFS, the neighbors of a node's neighbor are explored before exploring the node's other neighbors.

- DFS can be implemented using a stack, like the reachability algorithm.

Depth-first Search and Breadth-first Search

- If we replace the stack with a queue, that results in an exploration known as breadth-first search (or BFS).

- Recall: BFS explores a tree by traveling all paths to a given depth, then travelling all those paths one step deeper, then travelling them one step deeper, etc.
  - In other words, in BFS, all of a node’s neighbors are explored before exploring its neighbors’ neighbors.
  - That means BFS travels all paths of length 1, then travels all paths of length 2, then travels all paths of length 3, etc.

Depth-first Search and Breadth-first Search

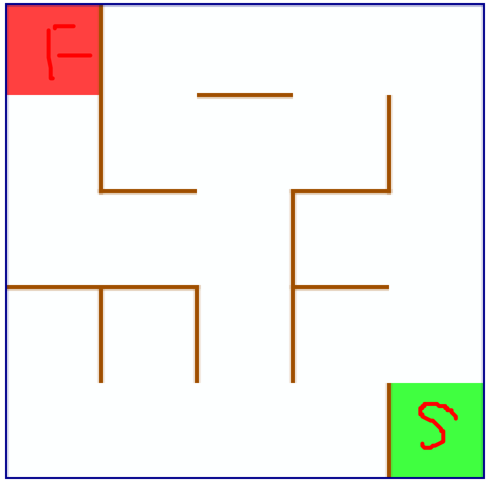
- General algorithm for DFS and BFS is below.

1. Initialize an empty set of visited vertices.
2. Initialize an empty stack (DFS) or queue (BFS). Add v_i to the stack/queue.
3. If the stack/queue is not empty, pop/dequeue a vertex v.
4. Perform any desired processing on v.
    - E.g. check if v meets a desired condition.
5. (DFS only): If v is not in the set of visited vertices:
    - Add v to the set of visited vertices.
    - Push each vertex that is direct successor of v to the stack.
6. (BFS only):
    - Add v to the set of visited vertices.
    - For each direct successor v' of v:
        - If v' is not in the set of visited vertices, enqueue it into the queue
7. Repeat from 3.

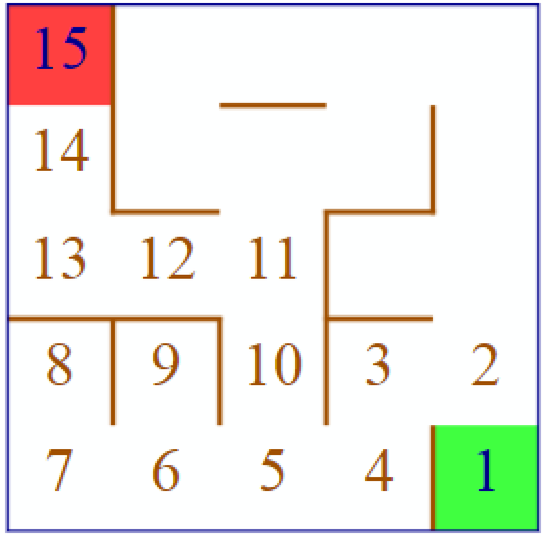
Depth-first Search and Breadth-first Search

- Often, we use BFS or DFS when we are looking for a node with a particular characteristic.

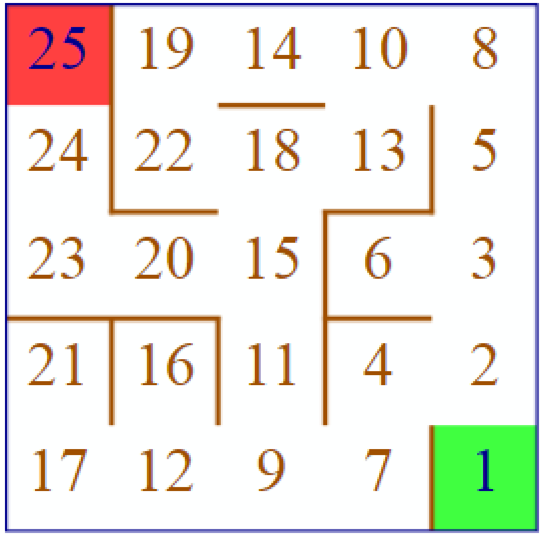
- For example, both algorithms can be used to find a path from start to finish in a maze.



Depth-first Search and Breadth-first Search



Depth-First (**Stack**)



Breadth-First (**Queue**)

DFS vs. BFS

Comparisons between DFS and BFS:

- DFS is a backtracking search: if we're looking for a node with a specific characteristic and DFS takes a path that doesn't contain such a node, it will backtrack to try a different path.
- In an infinite graph, DFS can become lost down an infinite path without ever finding a solution.
- BFS is complete and optimal: if a solution exists in the graph, BFS is guaranteed to find it, and it will find the shortest path to that solution.
- However, BFS may take a long time to find a solution if the solution is deep in the graph.

DFS vs. BFS (cont.)

Comparisons between DFS and BFS:

- DFS may find a deep solution more quickly.
- Both algorithms have O(V) space complexity in the worst case.
- However, BFS may take up more space in practice.
  - If the graph has a high branching factor, i.e. if each node has many neighbors, BFS can take a lot of memory to maintain all of the paths it's exploring on the queue.

Dijkstra’s algorithm: single source lowest-cost paths

- Dijkstra’s algorithm: finds the shortest/lowest-cost path from a specified vertex in a graph to all other reachable vertices in the graph.

- In Dijkstra’s algorithm, we will use a priority queue to order our search.
    - The priority values used in the queue correspond to the cumulative distance to each vertex added to the PQ.
    - Thus, we are always exploring the remaining node with the minimum cumulative cost.

Dijkstra’s algorithm: single source lowest-cost paths

Algorithm, which begins with some source vertex v_s:

- Initialize an empty map/hash table representing visited vertices.
  - Key is the vertex v.
  - Value is the min distance d to vertex v.
- Initialize an empty priority queue, and insert v_s into it with distance (priority) 0.
- While the priority queue is not empty:
  - Remove the first element (a vertex) from the priority queue and assign it to v. Let d be v’s distance (priority).
  - If v is not in the map of visited vertices:
    - Add v to the visited map with distance/cost d.
    - For each direct successor v_i of v:
      - Let d_i equal the cost/distance associated with edge (v, v_i).
      - Insert v_i to the priority queue with distance (priority) d + d_i.

Dijkstra’s algorithm: single source lowest-cost paths

- This version of the algorithm only keeps track of the minimum distance to each vertex, but it can be easily modified to keep track of the min-distance path, too.
  - Augment the visited vertex map and the priority queue to keep track of the vertex previous to each one added.
- The complexity of this version of the algorithm is O(|E|log |E|).
  - The innermost loop is executed at most |E| times, and the cost of the instructions inside the loop is O(log |E|).
    - Inner cost comes from inserting into the PQ.