# Exploration: Working with Graphs

## Introduction

In almost every case, the thing we are most interested in with graphs is if we can get from node A to node B. We might also be interested in knowing the distance between nodes and which path is the shortest. There are other things we might ask about graphs, but for this class, we are going to limit our discussion to these questions.

## Single source reachability

One important question we want to be able to ask about a graph is which nodes are reachable from some specific node. For example, from our previous airport graph, we might ask: which airports are reachable from PDX?

We can use a very simple algorithm to answer this question. It looks like this, if we're trying 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.

Looking for airports reachable from PDX would look like this:

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


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


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


v: PDX (already reachable)
```

```
successors: --
reachable: {PDX, SFO}
stack: [SEA, ORD]


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


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


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


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


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


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


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


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


v: PDX, MSP (both already reachable)
```

```
Successors: --
reachable: {LAX, MSP, ORD, PDX, SEA, SFO, STL}
stack: []


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

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. This would result in a different order of exploration of the graph. We'll look at this in the next section.

## Depth-first search and Breadth-first search

The reachability algorithm we saw above was an instance of depth-first search (or DFS). DFS is an algorithm for exploring a tree where we travel a particular path as far as we can take it 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 in the reachability algorithm above. If we replace the stack with a queue, this results in an exploration known as breadth-first search (or BFS).

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.

The general algorithm for DFS and BFS is below. For DFS, we use a stack, and, for BFS, we use a queue:

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.

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.

We can make some 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 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. We will briefly go over it here, but expect it to be covered more in your other classes.

In our previous airport graph, you could use Dijkstra's algorithm to say not only what airports could be reached from PDX, but also what the cheapest cost to reach each of those airports was.

Dijkstra's algorithm is structured very much like DFS and BFS, except for, in this 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 priority queue.
- Thus, we are always exploring the remaining node with the minimum cumulative cost.

Here's the 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) $d + d_i$.

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. You can do this by augmenting 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|). The inner cost comes from inserting into the priority queue.