Graphs

- 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.

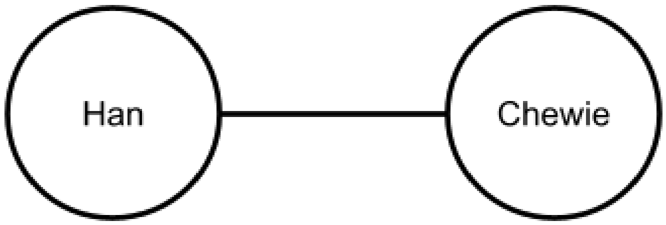
- Graphs are used in lots and lots of places in computer science:
  - Social networks like Facebook or Twitter
  - Computer graphics
  - Machine learning
  - Computer vision
  - Logistics and optimization
  - Computer networking

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

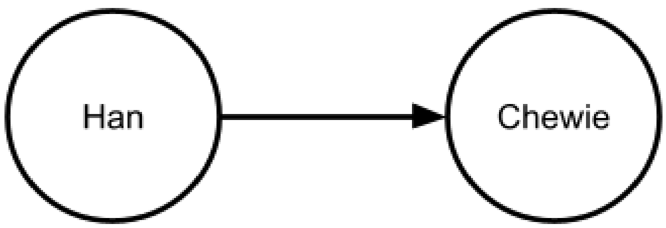
- Vertices represent objects, states (i.e. conditions or configurations), locations, etc.
  - These form a set where each vertex is unique (i.e. no two vertices represent the same object/state): $V = \{v_1, v_2, v_3, \ldots, v_n\}$

- 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***.

- 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:

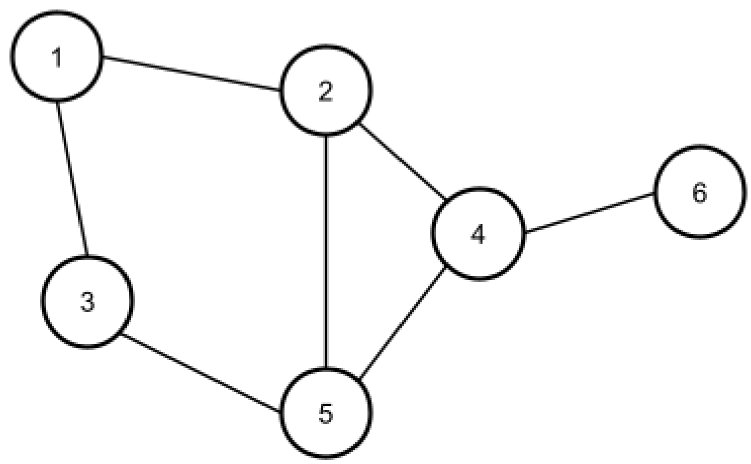


- A directed edge is like a follows relationship in Twitter, e.g. if Han follows Chewie, there would be a directed edge between them in the Twitter graph:

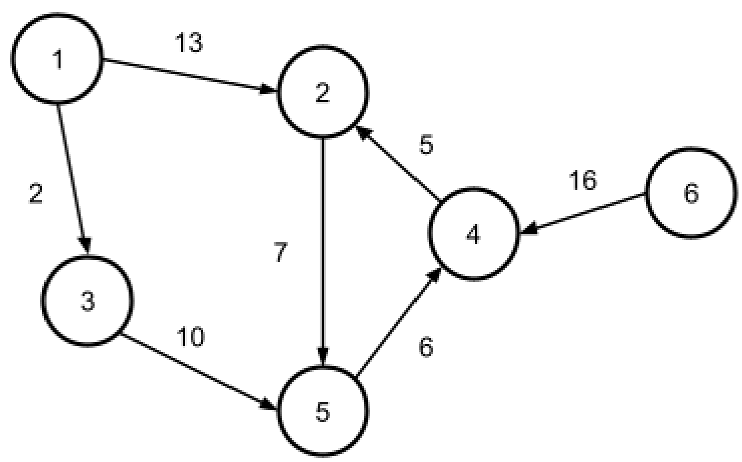


  - Here, we say the edge is directed ***from*** Han ***to*** Chewie.
  - We can also say that Han is the ***head*** of this edge and that Chewie is its ***tail***.
  - We can also say that Chewie is a ***direct successor*** of Han and that Han is a ***direct predecessor*** of Chewie.
  - We can also say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

- Here's an example of a small graph with 6 vertices and 7 undirected, unweighted edges:



- Here's an example of a similar graph with directed, weighted edges:



- 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.

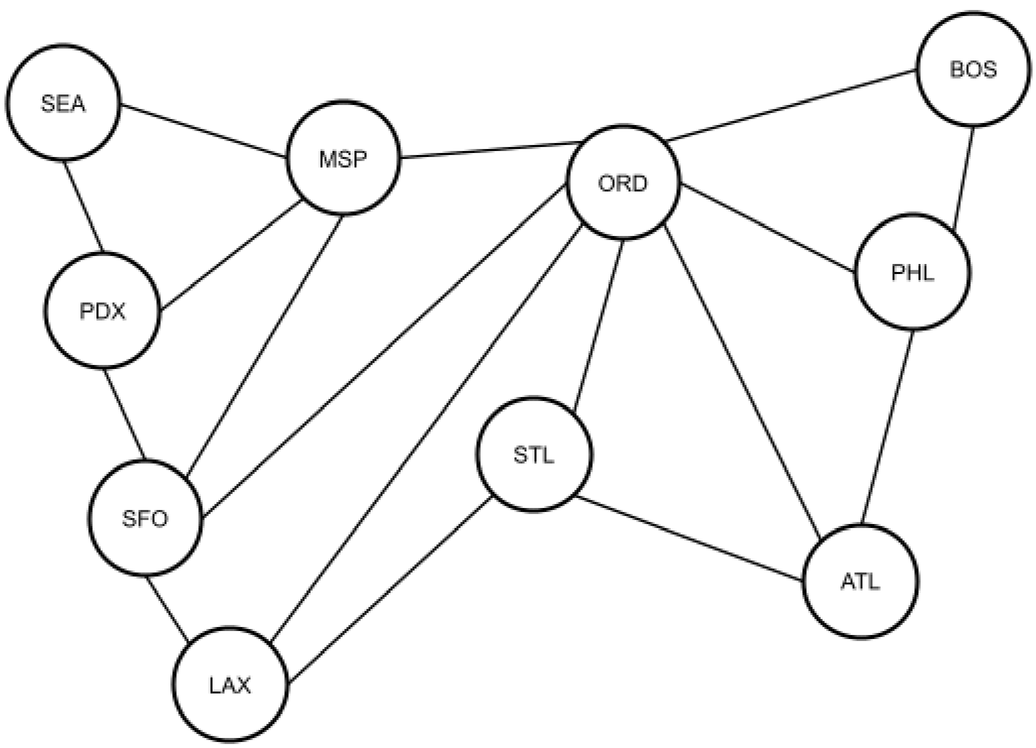
- There are lots of different kinds of questions we might want to ask about a graph:

- 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

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



- 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
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

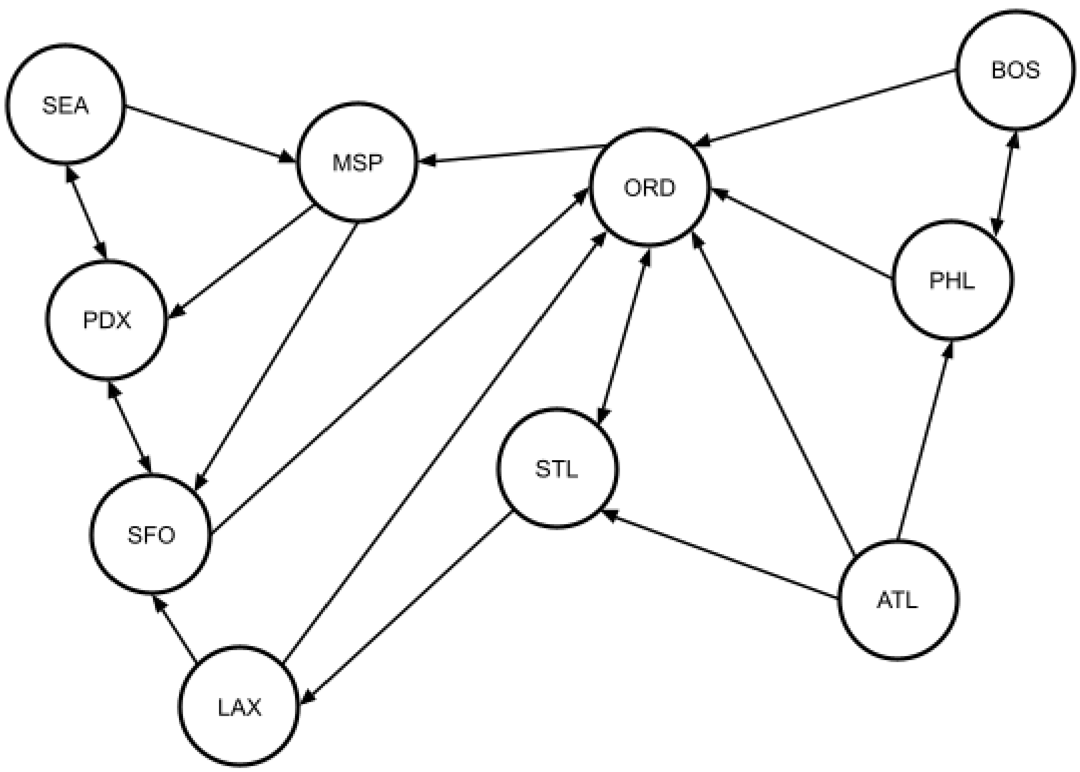
- Note that this matrix is symmetric.

- 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:



- 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]
```

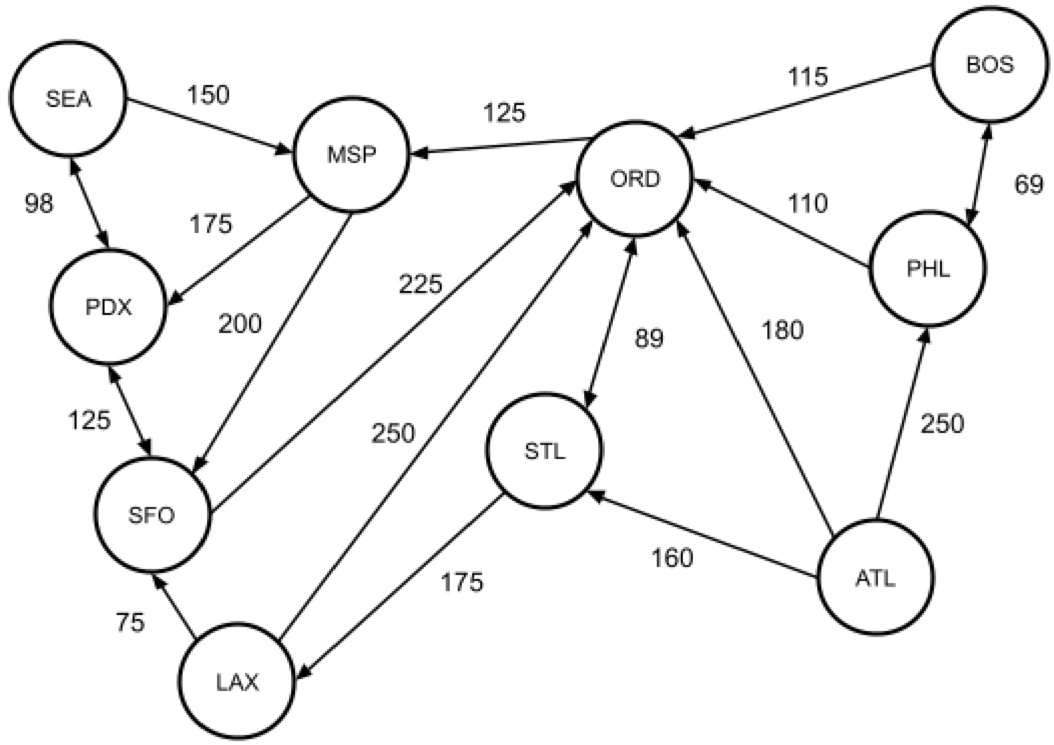
- Here, each vertex lists only the edges going *from* it.

- 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:



- 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:

	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

- One important question we want to be able to ask about a graph is what nodes are reachable from some specific node.

- For example, we might ask about our graph above: what 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:
    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]`

    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]
   ```

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]
   ```

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]
    ```

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}
    ```

- 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 the reachability algorithm above.

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

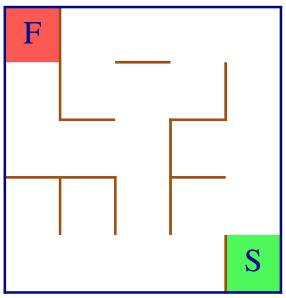
- BFS explores a tree by traveling all paths to a given depth, then travelling all those paths one step deeper, then travelling them one step deeper, etc.
    - In other words, in BFS, all of a node's neighbors are explored before exploring its neighbors' neighbors.
    - That means BFS travels all paths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

- For example, this presentation shows how both DFS and BFS can be used to find the end of the maze below: http://web.engr.oregonstate.edu/~hessro/static/media/GraphAlgorithmsII_DFS_BFS.d8183866.pdf



- 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.

- For example, in our graph above, 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 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 PQ.
    - 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.
  - 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.