CS 106B Section 8 (Week 9)

This week is about graphs with vertices and edges. The first couple pages are cheat sheets for graph terminology and common search algorithms

Recommended problems: #4, #5
Extra practice problems: 18.7, 18.12 (from Textbook)

---

graph: A data structure containing:

- a set of vertices V (sometimes called "nodes"),
- a set of edges E ("arcs"), where each is a connection between 2 vertices.

degree: number of edges touching a given vertex.

path: A path from vertex A to B is a sequence of edges that can be followed starting from A to reach B.

- can be represented as vertices visited, or edges taken

neighbor or **adjacent**: Two vertices connected directly by an edge.

reachable: Vertex A is reachable from B if a path exists from A to B.

connected graph: A graph is connected if every vertex is reachable from every other.

cycle: A path that begins and ends at the same vertex.

- **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a vertex to itself.

weight: Cost associated with a given edge.

- weighted graph: One where edges have weights *(see graph adjacent)*.

directed graph: A graph where edges are one-way connections.

undirected graph: A graph where edges don’t have a direction.

depth-first search (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking

- Often implemented recursively

breadth-first search (BFS): Finds a path between two vertices by taking one step down all paths and then immediately backtracking.

- Often implemented by maintaining a queue of vertices to visit.)

Dijkstra's algorithm: Finds paths between one vertex and all other vertices by maintaining information about how to reach each vertex (cost and previous vertex) and continually improving that information until it reaches the best solution.

- Often implemented by maintaining a *priority queue* of vertices to visit.

A* algorithm: A variation of Dijkstra's algorithm that incorporates a heuristic function to prioritize the order in which to visit the vertices.

minimum spanning tree: the set of connected edges with the smallest total weight that covers every vertex in the graph

Kruskal’s algorithm: An algorithm to find the minimum spanning tree of a graph

CS 106B Section 8 (Week 9)

Depth-first search (DFS) pseudocode:

```
stack = Stack()
stack.push(newPath(startNode))
seen = Set();
while !stack.isEmpty():
   currPath = stack.pop()
   currState = last(currPath);
   if(currState is goal) return currPath;
   if(seen contains currState) continue;
   seen.add(currState);
   for nextState in getNextStates(currState)
      path = newPath(currPath, nextState);
      stack.push(path);
   }
}
```

Breadth-first search (BFS) pseudocode:

```
queue = Queue()
queue.enqueue(newPath(startNode))
seen = Set();
while !queue.isEmpty():
   currPath = queue.dequeue()
   currState = last(currPath);
   if(currState is goal) return currPath;
   if(seen contains currState) continue;
   seen.add(currState);
   for nextState in getNextStates(currState)
      path = newPath(currPath, nextState);
      queue.enqueue(path);
   }
}
```

Dijkstra's algorithm pseudocode:

```
pQueue = PriorityQueue()
pQueue.enqueue(newPath(startNode), 0)
seen = Set();
while !pQueue.isEmpty():
   currPath = pQueue.dequeue()
   currState = last(currPath);
   if(currState is goal) return currPath;
   if(seen contains currState) continue;
   seen.add(currState);
   for nextState in getNextStates(currState)
      path = newPath(currPath, nextState);
      pQueue.enqueue(path, getCost(path));
   }
}
```

A* algorithm pseudocode:

```
pQueue = PriorityQueue()
pQueue.enqueue(newPath(startNode),
               H(startNode, goal))
seen = Set();
while !pQueue.isEmpty():
   currPath = pQueue.dequeue()
   currState = last(currPath);
   if(currState is goal) return currPath;
   if(seen contains currState) continue;
   seen.add(currState);
   for nextState in getNextStates(currState)
      path = newPath(currPath, nextState);
      pQueue.enqueue(path, getCost(path) +
                     H(nextState, goal);
   }
}
```

*Note: A **path** in the above pseudocode is a Vector of vertices from the graph and a **state** is a vertex in the graph*

Important parts of Stanford Graph library: (more online)

```
BasicGraph()
g.addEdge(v1, v2);
g.addVertex(vertex);
g.clear();
g.getEdge(v1, v2)
g.getEdgeSet()
g.getEdgeSet(vertex)
g.getNeighbors(vertex)
```

```
g.getVertex(name)
g.getVertexSet()
g.isConnected(v1, v2)
g.isEmpty()
g.removeEdge(v1, v2);
g.removeVertex(vertex);
g.size()
g.toString()
```

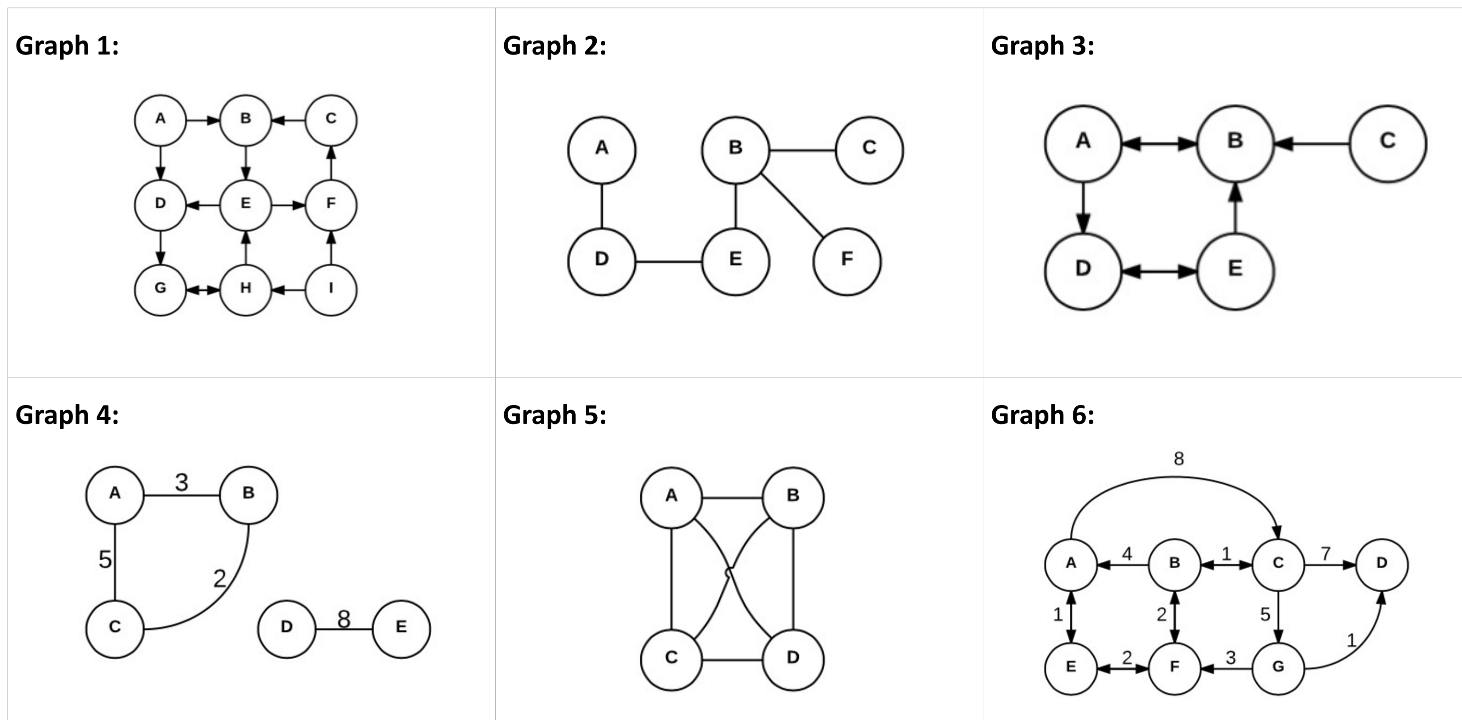
```
struct Vertex {
    string name;
    Set<Edge*> edges;

    double cost;  // initially 0.0
    bool visited;  // initially false
    Node* previous;  // initially NULL
};
```

```
struct Edge {
    Vertex* start;
    Vertex* finish;
    double cost;

    bool visited;    // initially false
};
```

CS 106B Section 8 (Week 9)



---

1. Depth-First Search (DFS).

Write the paths that a depth-first search would find from vertex A to all other vertices in the following graphs. If a given vertex is not reachable from vertex A, write "no path" or "unreachable".

- in Graph 1
- in Graph 6

---

2. Breadth-First Search (BFS).

Write the paths that a breadth-first search would find from vertex A to all other vertices in the following graphs. Which paths are shorter than the ones found by DFS in the previous problem?

- in Graph 1
- in Graph 6

---

3. Minimum weight paths.

Which paths found by DFS and BFS on Graph 6 in the previous problems are not minimal weight? What are the minimal weight paths from vertex A to all other nodes? *(Just inspect the graph manually.)*

---

4. isReachable.

Write a function named **isReachable** that returns true if a path can be made from the vertex v1 to the vertex v2 , or false if not. If the two vertices are the same, return true. Use either BFS or DFS, described in the reference above. Bonus: do this problem twice with both BFS and DFS.

```
bool isReachable(BasicGraph& graph, Vertex* v1, Vertex* v2) { ...
```

5. isConnected.

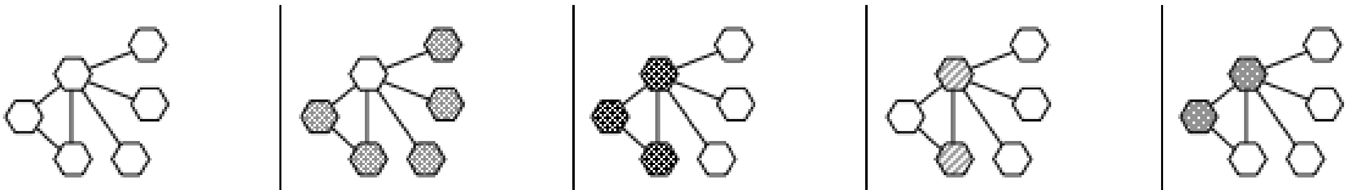
Write a function named **isConnected** that returns true if a path can be made from every vertex to any other vertex, or false if there is any vertex cannot be reached by a path from some other vertex. An empty graph is defined as being connected. You can use the isReachable function from the previous problem to help solve this one.

```
bool isConnected(BasicGraph& graph) { ...
```

6. findMinimumVertexCover.

Write function named **findMinimumVertexCover** that returns a set of vertex pointers identifying a minimum vertex cover. A *vertex cover* is a subset of an undirected graph's vertices such that each and every edge in the graph is incident to at least one vertex in the subset. A *minimum vertex cover* is a vertex cover of the smallest possible size. Consider the following graph on the left:

Graph Vertex Covers



Each of the four illustrations after it on the right shows some vertex cover (shaded nodes are included in the vertex cover, and hollow ones are excluded). Each one is a vertex cover because each edge touches at least one vertex in the cover. The two vertex covers on the right are *minimum* vertex covers, because there is no smaller vertex cover.

Understand that because the graph is undirected, that means for every edge that leads from some vertex v1 to v2, there will be an edge that leads from v2 to v1. If there are two or more minimum vertex covers, then you can return any one of them. Think of this as a backtracking problem. The implementation of this function should consider every possible vertex subset, keeping track of the smallest one that covers the entire graph. Try all possible vertex combinations using a "choose-explore-unchoose" pattern and keep track of state along the way.

```
Set<Vertex*> findMinimumVertexCover(BasicGraph& graph) { ...
```