Problem 8-1. (Graph Representation)

(a) Discuss the advantages and disadvantages of using an adjacency matrix to represent a graph as opposed to an adjacency list. In which cases would you use one over the other?

Solution:

Adjacency Matrix:

- Efficient edge access: $\Theta(1)$ to check if edge is in the graph
- Space inefficient since allocate space for every possible edge in the graph: $\Theta(V^2)$.
- Ideal for dense graphs ($E \approx V^2$)

Adjacency List:

- Inefficient edge access: $\Theta(V)$ worst case to check if edge is in the graph
- Space efficient since only allocate space for the actual edges in the graph: $\Theta(E)$
- Ideal for sparse graphs ($E << V^2$)

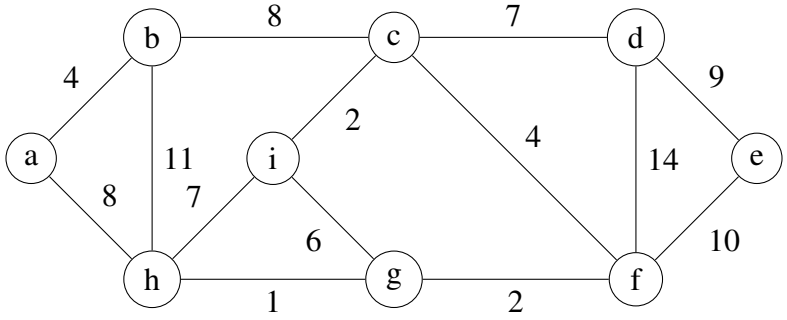
(b) Does the time complexity of BFS and DFS change depending on whether we represent the graph as an adjacency matrix or as an adjacency list?

Solution:

Yes, the time complexity of BFS and DFS for an adjacency matrix implementation is $O(|V|^2)$. For an adjacency list, it is $O(|V| + |E|)$.

Problem 8-2. (Dijkstra's Algorithm)

Consider the following positively weighted undirected graph for the problems below:



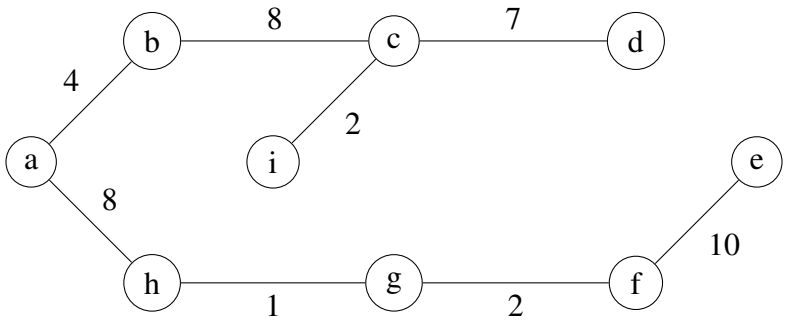
(a) Perform Dijkstra's algorithm on the given graph and provide the (vertex, shortest distance) pairs in the order that they are popped from the priority queue.

Solution:

(a, 0), (b, 4), (h, 8), (g, 9), (f, 11), (c, 12), (i, 14), (d, 19), (e, 21)

(b) Draw the resulting shortest path tree.

Solution:



(c) How can we apply Dijkstra's algorithm to find the minimum weight cycle in a directed graph? Provide pseudocode for an efficient algorithm that returns the weight of this cycle. What is the runtime of your algorithm in terms of the number of vertices and/or edges?

Solution:

We can run Dijkstra's with each vertex v as the source node to retrieve the minimum distances to every other vertex. Any edge $u \to v$ in which u is reachable from v (that is, has a non-infinite distance from v) will indicate a cycle with a weight $w = dist(v,u) + weight(u \to v)$. This algorithm takes $O(|V| \cdot (|V| + |E|)log|V|)$ time.

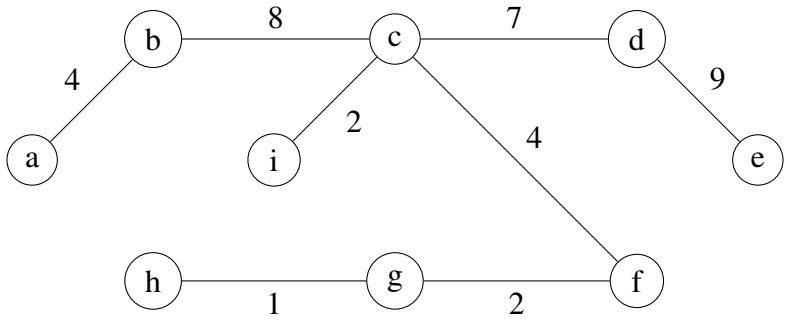
```
function min_weight_cycle(G, V, E):
    min_weight = inf
    for each vertex v in V:
        dist = dijkstras-alg(G, v)
        for each edge e = (t, v) in E:
            min_weight = min(e.weight + dist[t], min_weight)
    return min_weight
```

Problem 8-3. (Prim and Kruskal)

(a) Perform **Prim's algorithm** on the graph from Problem 8-2 to find a minimum spanning tree, supposing we select node a to be the source node. Show the order in which the edges are added and draw the resulting MST. Assume for this problem that ties are broken alphabetically (e.g. if there is a tie between nodes a and e, choose node a)

Solution:

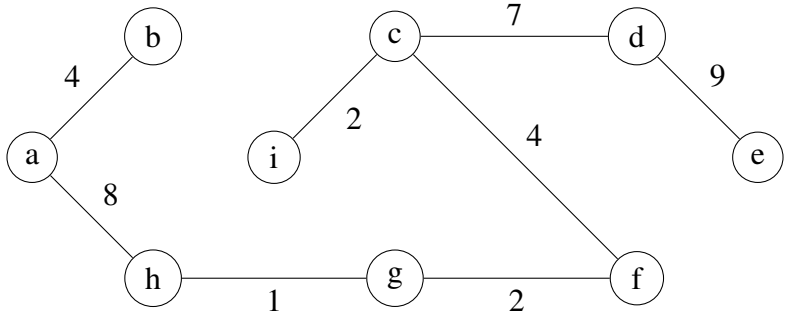
(a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e)



(b) Repeat part (a) using **Kruskal's algorithm**.

Solution:

(h, g), (i, c), (g, f), (a, b), (c, f), (c, d), (a, h), (d, e)



It is also valid to have chosen edge (b, c) in the tie between (a, h) and (b, c).

Problem 8-4. (Decifering Alien Language)

Consider an alien language which uses the latin alphabet (containing the 26 characters in the set {a, b, c,..., z}). However, the order among letters are unknown to you. You receive a potentially incomplete list of non-empty words from a dictionary, where words are sorted lexicographically by the rules of this new language. Give pseudocode for an algorithm to derive the order of letters in this language.

Solution:

We can use a BFS approach, maintaining a queue of characters for which we have already processed all of their predecessors (topological sort). When popping a character c from the queue, decrement the indegree of all characters c' for which we have established a direct ordering with respect to c. If a character's indegree reaches 0, we can push it onto the queue. We continue this process, until all characters have been processed.

```
Function determineOrdering(dict):
        root = None
        for i in 1...(len(dict)-1):
            word1 = dict[i]
```

```
            word2 = dict[i+1]
            # Helper function returns the first pair
            # of characters that differ
            # We know that char1 comes before char2 in the language
            char1, char2 = findFirstDifference(word1, word2)
            # Assume that this function will either return a reference
            # to an existing node or create one otherwise
            node1 = getNode(char1)
            node2 = getNode(char2)
            node1.neighbors.add(node2)
            node2.indegree += 1
            if not root: root = node1

        result = []
        queue.push(root)
        while queue:
            node = queue.pop()
            result.append(node.char)
            for n in node.neighbors:
                n.indegree -= 1
                if n.indegree == 0:
                    queue.push(n)
        return result
```

Problem 8-5. (Course Scheduling)

There are a total of n courses you have to take, labeled from 1 to n.

Some courses may have prerequisites, for example to take course 1 you have to first taken course 2, which is expressed as a pair: [1,2]

Given the total number of courses *N* and a list of prerequisite pairs, give pseudocode for an algorithm to determine if is it possible for you to finish all courses.

Solution:

We can use a DFS approach to check for cycles in the schedule graph.

```
    Function checkSchedule(N, prerequisites):

        visited = {False} * N
        completed = {}

        Function can_complete(i):
```

```
        if i in completed:    # already checked
            return true
        if visited[i]:        # cycle found
            return false

        visited[i] = true

        for c in edges_from[i]:
            if not can_complete(c):
                return false

        completed.add(i)
        return True

    for (c, p) in prerequisites:
        edges_from[p].add(c)

    for i from 1 ... N:
        if not can_complete(i):
            return false
    return true
```

We can use a similar approach as the solution to question 8-4. The difference is that we need to determine whether the number of courses we've popped off the queue is N.

```
Function checkSchedule(N, prerequisites):

    required = N

    for (course, p) in prerequisites:
        indegree[course] ++

    for course in (1 ... N):
        if (indegree[course] == 0):
            queue.push(course)

    while queue not empty:

        c = queue.pop()
        required--

        for (course, prereq) in prerequisites:
```

```
            if (prereq == c):
                indegree[course]--
            if (indegree[course] == 0):
                queue.push(course)

    if (required == 0):
        return true

    return false   # not possible to complete
```