Problem Set 7

> **Due dates:** Electronic submission of the pdf file of this homework is due on **10/24/2025 before 11:59pm** on canvas. The homework must be typeset with LaTeX to receive any credit. All answers must be formulated in your own words.

Watch out for additional material that will appear on Thursday! Deadline is on Friday, as usual.
Name: Tejas Lipare

Resources.

- Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. 4th ed. Cambridge, MA: The MIT Press, 2022.
- Lecture PDFs.

On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment. Furthermore, I have disclosed all resources (people, books, web sites, etc.) that have been used to prepare this homework.

Signature: Tejas Lipare

Read the chapters on "Elementary Graph Algorithms" and "Single-Source Shortest Paths" in our textbook before attempting to answer these questions.

Problem 1 (20 points)**.** Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of the number $|E|$ of edges.

Solution.
The key to achieving an $O(V)$ runtime independent of $|E|$ lies in a fundamental property of undirected graphs. An undirected graph $G = (V, E)$ is acyclic if and only if it is a forest. A forest with $|V|$ vertices and k connected components has exactly $|V| - k$ edges. Since any graph must have at least one component ($k \geq 1$), any acyclic graph (forest) must satisfy the property $|E| \leq |V| - 1$.

In other words, we can say that, if an undirected graph $G = (V, E)$ has $|E| \geq |V|$, it must contain a cycle.

This property allows us to design an algorithm that bounds its own execution time. We can perform a graph traversal, such as a Depth-First Search (DFS), and simply count the number of edges we explore.

In a standard DFS of an undirected graph, each edge (u, v) is explored twice: once from u to v, and once from v to u. More precisely, the total number of entries iterated over in all adjacency lists is $2|E|$.

If the graph is acyclic, it has at most $|V| - 1$ edges. Therefore, a complete DFS on an acyclic graph will involve a total of at most $2 \times (|V| - 1)$ edge explorations.

If our DFS algorithm at any point performs more than $2 \times (|V| - 1)$ edge explorations, we can immediately stop and conclude that the graph must contain a cycle, as this count is impossible for an acyclic graph. This check effectively bounds our runtime at $O(V)$, regardless of how many total edges ($|E|$) exist.

Algorithm:

We will use a modified Depth-First Search. We need a global counter for edge explorations and a standard visited array. The graph G is assumed to be represented by an adjacency list.

1. Initialize a boolean array $visited[0 \ldots |V| - 1]$ to all false.
2. Initialize an integer counter $edge\_count = 0$.
3. Iterate through each vertex v from 0 to $|V| - 1$:
4. If $visited[v]$ is false:
5. // Pass count by reference

6. $result = DFS(v, null, visited, \&edge\_count)$
7. If $result$ is true (cycle found), then immediately return CYCLE_EXISTS.
8. If the loop finishes without finding a cycle, return NO_CYCLE.

Function DFS($u, parent, visited, \&edge\_count$)

1. Mark $visited[u] = true$.
2. For each neighbor v in the adjacency list of u:
3. Increment $edge\_count$.
4. If $edge\_count > 2 \times (|V| - 1)$, return true. // Cycle is guaranteed
5. If v is equal to parent:
6. Continue to the next neighbor (this avoids trivial $parent \to child \to parent$ cycles).
7. If $visited[v]$ is true:
8. Return true. // A back-edge, and thus a cycle, is explicitly found
9. $sub\_result = DFS(v, u, visited, \&edge\_count)$
10. If sub_result is true:
11. Return true. // A cycle was found
12. Return false. // No cycle found in this branch

Correctness:

There are two ways the algorithm can return CYCLE_EXISTS:

1. **Step 7:** A back-edge to an already visited node (that is not the immediate parent) is found. This is the standard DFS method for finding a cycle in an undirected graph, and it is correct.
2. **Step 4:** This is the step which we have introduced to bound the execution time of the algorithm. The number of edge explorations exceeds $2(|V|-1)$. As established, an acyclic graph can have at most $|V| - 1$ edges. A full DFS on an acyclic graph would result in exactly $2|E|$ edge explorations, and $2|E| \leq 2(|V| - 1)$. Therefore, if this threshold is exceeded, the graph cannot be acyclic and must contain a cycle.

If the graph is acyclic, it has $|E| \leq |V| - 1$. The standard DFS check (Step 7) will never find a back-edge. The total edge explorations will be $2|E|$, which is $\leq 2(|V| - 1)$. Thus, the condition in Step 4 also will never be met. The algorithm will complete and correctly return NO_CYCLE.

Time Complexity:

The algorithm's runtime is $O(V)$, independent of $|E|$.

1. Initialization of the `visited` array (Step 1) takes $O(V)$ time.
2. The main loop (Step 3) iterates V times.
3. The total work done by all calls to DFS is the sum of vertex visits and edge explorations.
4. Each vertex is marked *visited* at most once (due to the check in the main loop). This contributes $O(V)$ to the total runtime.
5. The inner loop (Step 2) is where edge explorations happen. However, the *edge_count* is global and its growth is capped. The total number of times Step 3 ($edge\_count++$) can execute across all recursive calls combined is at most $2 \times (|V| - 1) + 1$, which is $O(V)$.
6. As soon as the exploration count exceeds this $O(V)$ bound, the algorithm terminates.
7. If the graph has $|E| \gg |V|$ (e.g., $|E| = O(V^2)$), the algorithm will not explore all edges. It will simply hit the $O(V)$ exploration cap (Step 4) and terminate, correctly reporting a cycle.

The total time is $O(V)$ (initialization) + $O(V)$ (vertex visits) + $O(V)$ (capped edge explorations) = $O(V)$.

Problem 2 (20 points)**.** Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m+1$ passes, even if m is not known in advance.

Solution.
The Standard Bellman-Ford Property:

The Bellman-Ford algorithm works by iteratively relaxing edges. Let $d_k[v]$ be the shortest path weight found from the source s to vertex v using at most k edges.

The standard algorithm's is based on the following recurrence:

$$d_k[v] = \min(d_{k-1}[v], \min_{(u,v)\in E}\{d_{k-1}[u] + w(u, v)\})$$

After k full iterations over all edges, the algorithm has correctly computed the shortest path weight for all vertices v whose shortest path (by weight) can be found with at most k edges.

Since the graph has no negative-weight cycles, any simple shortest path has at most $|V| - 1$ edges. This is why the standard algorithm runs $|V| - 1$ passes to guarantee finding all shortest path weights.

Introduction of m to Bellman-Ford iterations:

The problem defines m as the maximum over all vertices v of the minimum number of edges in a shortest path from s to v.

Let $\delta(s, v)$ be the true shortest path weight from s to v. By the definition of m, for any vertex $v \in V$, there exists at least one shortest path from s to v (with weight $\delta(s, v)$) that has $k_v \leq m$ edges.

Based on the Bellman-Ford property, after m passes, the algorithm will have found the shortest path for all vertices v, because $d_m[v]$ will be $\delta(s, v)$ for all $v \in V$.

In other words, after m passes, the d array (containing the $d[v]$ values) has its final set of values.

The $(m + 1)$-th iterations:

On $m + 1$-th iteration, the algorithm will again iterate through every edge $(u, v) \in E$ and check the relaxation condition:

Is $d[v] > d[u] + w(u, v)$?

Since m passes have already completed, we know that $d[v] = \delta(s, v)$ and $d[u] = \delta(s, u)$ for all u, v.

Because the graph has no negative-weight cycles, the shortest path weights must satisfy the triangle inequality: $\delta(s, v) \leq \delta(s, u) + w(u, v)$ This means $d[v] \leq d[u] + w(u, v)$ for all edges.

Therefore, the relaxation condition $d[v] > d[u] + w(u, v)$ will never be true for any edge during the $(m + 1)$-th pass. No $d[v]$ values will be updated.

The Change:

The change is to stop the algorithm after any pass in which no $d[v]$ value is successfully updated (relaxed).

We can implement this by introducing a boolean flag in the main loop.

Modified Algorithm Pseudocode:

```
MODIFIED-BELLMAN-FORD(G, w, s)
    // Initialization of distance array
    d[s] = 0, all other d[v] = infinity
    // Standard loop runs |V|-1 times
    for i = 1 to |V| - 1:
        // Newly introduced boolean flag
        has_changed = false
        // Inner loop over all edges
        for each edge (u, v) in E:
            if d[v] > d[u] + w(u, v):
                d[v] = d[u] + w(u, v)
                has_changed = true
        // If no d[v] value changed in this full pass, we are done
        if has_changed == false:
            break  // Exit the main loop early
    return d
```

Analysis of Termination:

The algorithm will perform passes $1, 2, \ldots, m$. In each of these passes, at least one $d[v]$ value might be updated, so *has_changed* will be set to true. For $m = 0$, s is an isolated vertex, in which case it terminates after 1 pass.

After pass m, all $d[v]$ values have converged to $\delta(s, v)$. The algorithm then begins pass $m + 1$.

During pass $m+1$, as proven above, no relaxation can occur. The *has_changed* flag will remain false.

At the end of pass $m + 1$, the break statement will be executed.

Thus, the algorithm terminates after exactly $m+1$ passes, as required, without ever needing to know the value of m.

Problem 3 (20 points)**.** Suppose that we change line 6 of Dijkstra's algorithm in our textbook to the following.

6 **while** $|Q| > 1$

This change causes the while loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct? Explain. [Use the version of Dijkstra's algorithm from the textbook]

Solution.
Dijkstra's algorithm works by maintaining a set S of vertices whose final shortest-path weights have been determined. It repeatedly extracts the vertex $u \in Q = V - S$ with the minimum shortest-path estimate $d[u]$. Anothe variant of the algorithm is that when a vertex u is extracted from Q, its path weight is final and correct: $d[u] = \delta(s, u)$.

After extracting u, the algorithm relaxes all edges (u, v). This relaxation step is updates the $d[v]$ values for u's neighbors, ultimately finding a new, shorter path to v that goes through u.

Analysis:

1. The original algorithm's loop while $Q! = \phi$ executes $|V|$ times.
2. The modified algorithm's loop while $|Q| > 1$ executes $|V| - 1$ times.
3. This means the modified algorithm halts just before the last vertex would be extracted from Q. Let's call this vertex k
4. For the first $|V| - 1$ vertices that are extracted, the logic is identical to the standard algorithm. When each of these vertices u is extracted, its $d[u]$ value is final and correct ($d[u] = \delta(s, u)$).
5. We only need to worry about the one vertex k that remains in Q. So the question is if its $d[k]$ value is correct when the loop terminates.
6. **Case 1: k is reachable from s.** Let $s \to x \to k$ be a shortest path to k, where x is the predecessor of k on this path. Because all edge weights are non-negative, the shortest-path distance to x must be less than or equal to the shortest-path distance to k: $\delta(s, x) \leq \delta(s, k)$.
7. Because k is the last vertex left, it must have the largest finite shortest-path distance from s. Its predecessor x (where $x \neq k$) must have $\delta(s, x) \leq \delta(s, k)$. Therefore, x must have been extracted from Q in one of the $|V| - 1$ steps.
8. When x was extracted, $d[x]$ was finalized to $\delta(s, x)$. The algorithm then executed the loop for each vertex v in $Adj[x]$, which includes k. The operation $RELAX(x, k, w(x, k))$ was performed.
9. This $RELAX$ call ensured that $d[k]$ was set to $\min(d[k], d[x] + w(x, k))$. Since $s \to x \to z$ is a shortest path, $d[x] + w(x, k) = \delta(s, x) + w(x, k) = \delta(s, k)$. The $d[k]$ value stored in the priority queue was therefore correctly updated to its true shortest-path value at some point before the loop terminated.
10. **Case 2: k is unreachable from s.** In this case, $d[k]$ was initialized to ∞ and no relaxation operation would ever update it. The algorithm terminates with $d[k] = \infty$, which is correct.

The final, $|V|$-th iteration of the original algorithm is only necessary to finalize $d[k]$ and then relax the edges outgoing from k. This is not necessary for computing the shortest-path distance to k. The value $d[k]$ is already correct after its predecessor x has been extracted and the edge (x, k) has been relaxed. Therefore, stopping after $|V| - 1$ iterations is sufficient.

Problem 4 (40 points)**.** Help Professor Charlie Eppes find the most likely escape routes of thieves that robbed a bookstore on Texas Avenue in College Station. The map will be published on Thursday evening. In preparation, you might want to implement Dijkstra's single-source shortest path algorithm, so that you can join the manhunt on Thursday evening. Include your implementation of Dijkstra's algorithm and explain all details of your choice of the min-priority queue.

[Edge weight 1 means very desirable street, weight 2 means less desirable street]

Solution.
The goal is to assist Professor Charlie Eppes in identifying the most likely escape routes for thieves from a bookstore on Texas Avenue. We are given a map of College Station represented as a graph with $V = 22$ vertices (locations) and $E = 28$ undirected edges (streets). The source of the robbery is vertex 1. Each edge has a weight: weight 1 indicates a "very desirable street," and weight 2 indicates a "less desirable street." The "most likely" route is assumed to be the "shortest path," where the length of a path is the sum of the weights of its edges. We must implement Dijkstra's single-source shortest path algorithm to find the shortest paths from the source to all other vertices.

Dijkstra's Algorithm Overview:

Dijkstra's algorithm is a classic and efficient algorithm for finding the shortest paths from a single source vertex to all other vertices in a weighted graph, provided that all edge weights are non-negative. Our graph fits this requirement, as all weights are 1 or 2. The algorithm works in a greedy manner. It maintains a set of visited vertices, for which the shortest path from the source is known, and a set of unvisited vertices. It also keeps track of the shortest distance found so far from the source to every other vertex, initializing these distances to infinity (∞) and the source's distance to 0.

The core loop of the algorithm is as follows:

1. Select the unvisited vertex u that has the smallest known distance from the source.
2. Mark u as visited. Its shortest path is now considered final.
3. For each neighbor v of u (connected by an edge with weight w):

- This step is called relaxation. We check if the path to v through u is shorter than the currently known shortest distance to v.
- That is, we check if: $distance(u) + w < distance(v)$.
- If it is shorter, we update $distance(v)$ to this new, smaller value: $distance(v) = distance(u) + w$.

4. Repeat steps 1-3 until all reachable vertices have been visited.

Choice of the Min-Priority Queue:

The most computationally expensive part of Dijkstra's algorithm is step 1: "Select the unvisited vertex u that has the smallest known distance."

Naive Approach:

One could maintain an array of distances and, in each of the V iterations, scan through all V vertices to find the minimum distance. This search takes $O(V)$ time. Since this is done V times, the total time for this step alone would be $O(V^2)$. The total runtime for the algorithm would be $O(V^2 + E)$, or simply $O(V^2)$ for dense graphs.

Optimized Approach (Min-Priority Queue):

We use a min-priority queue to speed up the process. A priority queue is an abstract data structure that efficiently supports two key operations:

- **Push:** Add an element with a specific priority.
- **Pop:** Remove and return the element with the *minimum* priority.

In my C++ implementation, I have used `std::priority_queue`, which is typically implemented using a binary heap. A binary heap provides $O(\log N)$ time complexity for both push and pop operations, where N is the number of elements in the queue.

Implementation Details in C++:

- We store pairs in the priority queue: `pair<int, int>`, which represents {`distance, vertex_id`}. The queue automatically sorts elements based on the first item in the pair, the distance.
- `std::priority_queue` in C++ is a *max-heap* by default (it pops the *largest* element). To make it a *min-heap* (to pop the *smallest* distance), we must provide a custom comparator.
- The declaration: `priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;`
  - `pair<int, int>`: The type of element stored.

  - `vector<pair<int, int>>`: The underlying container.
  - `greater<pair<int, int>>`: The comparator. This tells the queue to use the `>` operator, effectively reversing the default (which uses `<`) and creating a min-priority queue.

- **Handling Updates:** A standard binary heap data structure for Dijkstra's requires a `decrease-key` operation (finding an element in the heap and reducing its priority). The C++ `std::priority_queue` does not support this operation efficiently.
- **The Workaround:** As seen in the code, instead of `decrease-key`, we simply push a new pair `{new_distance, vertex}` onto the queue whenever we find a shorter path (the relaxation step).
- This means the queue might contain multiple "stale" entries for the same vertex (e.g., `{5, v}`, `{8, v}`, `{12, v}`). However, because we are using a min-priority queue, we are *guaranteed* to process the entry with the actual shortest distance (e.g., `{5, v}`) first.
- When the stale, more-expensive entries (like `{8, v}`) are eventually popped, the check `if(d+w < shortestDistances[adjNode])` will fail, as `d` (the stale, larger distance) will not be able to produce a shorter path than the one already found. The algorithm's correctness is preserved.

Time Complexity with Priority Queue:

- V vertices, E edges.
- Initialization: $O(V)$ to set all distances to `INT_MAX`.
- In the worst case, every edge relaxation (of which there are $O(E)$) could result in a `push` operation. The queue size can grow up to $O(E)$.
- Each `push` takes $O(\log E)$ time. Total push time: $O(E \log E)$.
- Each `pop` (extract-min) also takes $O(\log E)$ time. Total pop time: $O(E \log E)$.
- Since E can be at most $O(V^2)$, $\log E$ is $O(\log(V^2)) = O(2 \log V) = O(\log V)$.
- Therefore, the total time complexity of this implementation is $O(E \log V)$, which is significantly faster than the $O(V^2)$ naive approach for sparse graphs (where E is much smaller than V^2).

Graph Adjacency List:

The graph consists of $V = 22$ vertices and $E = 28$ undirected edges. The adjacency list, mapping each vertex u to its neighbors v with the corresponding weight w in the format `u: (v, w), ...`, is as follows:

- `1: (2, 1), (11, 1)`

- 2: (1, 1), (3, 1), (21, 2)
- 3: (2, 1), (4, 1), (8, 2)
- 4: (3, 1), (5, 1)
- 5: (4, 1), (6, 2), (7, 1), (22, 1)
- 6: (5, 2), (7, 1)
- 7: (5, 1), (6, 1), (8, 1)
- 8: (3, 2), (7, 1), (9, 1)
- 9: (8, 1), (10, 1), (19, 2)
- 10: (9, 1), (11, 1), (18, 2)
- 11: (1, 1), (10, 1), (17, 1), (12, 2)
- 12: (11, 2), (13, 2)
- 13: (12, 2), (14, 2), (21, 1)
- 14: (13, 2), (15, 1), (16, 1), (20, 1)
- 15: (14, 1)
- 16: (14, 1), (17, 2)
- 17: (11, 1), (16, 2), (18, 2)
- 18: (10, 2), (17, 2)
- 19: (9, 2)
- 20: (14, 1), (21, 2), (22, 1)
- 21: (2, 2), (13, 1), (20, 2), (22, 2)
- 22: (5, 1), (20, 1), (21, 2)

C++ Code Implementation:

The following is the complete C++ program used to solve the problem. A link to the code repository can be found here: Link

```cpp
/* Dijkstra's single source shortest path algorithm */
#include<iostream>
#include<vector>
#include<queue>
#include<climits>
#include<unordered_map>
using namespace std;

// Time: E.log(V)
vector<int> dijkstra(int src, int n,
        vector<vector<pair<int, int>>>& adjList) {
    vector<int> shortestDistances(n, INT_MAX);
    shortestDistances[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>,
            greater<pair<int, int>>> pq;
    pq.push({0, src});
    while(!pq.empty()) {
        int curr = pq.top().second;
        int d = pq.top().first;
        pq.pop();
        for(auto adjNodePair : adjList[curr]) {
            int adjNode = adjNodePair.first;
            int w = adjNodePair.second;
            if(d+w < shortestDistances[adjNode]) {
                shortestDistances[adjNode] = d+w;
                pq.push({shortestDistances[adjNode],
                        adjNode});
            }
        }
    }
    return shortestDistances;
}

int main()
{
    int nodes, edges;
    cin >> nodes >> edges;
    vector<vector<pair<int, int>>> adjList(nodes+1);
    for(int i=0; i<edges; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adjList[u].push_back({v, w});
        adjList[v].push_back({u, w});
    }
    vector<int> shortestDistances = dijkstra(1, nodes+1,
                                            adjList);
    vector<int> destinations = {6, 8, 9, 15, 16, 22};
    for(int d : destinations) {
        cout << "Distance of " << d << " from 1 is " <<
```

```
                shortestDistances[d] << endl;
    }
    return 0;
}

/*
Input:
22 28
 1  2 1
 1 11 1
 2  3 1
 2 21 2
 3  4 1
 3  8 2
 4  5 1
 5  6 2
 5  7 1
 5 22 1
 6  7 1
 7  8 1
 8  9 1
 9 10 1
 9 19 2
10 11 1
10 18 2
11 17 1
11 12 2
12 13 2
13 14 2
13 21 1
14 15 1
14 16 1
14 20 1
16 17 2
17 18 2
20 21 2
20 22 1
21 22 2

Output:
Distance of 6 from 1 is 6
Distance of 8 from 1 is 4
Distance of 9 from 1 is 3
Distance of 15 from 1 is 6
Distance of 16 from 1 is 4
Distance of 22 from 1 is 5
*/
```

Listing 1: Dijkstra's Algorithm in C++ for Finding Escape Routes

Results and Conclusion:

The code was executed using the provided graph data. The source vertex was set to 1 (the bookstore). The algorithm calculated the shortest path (minimum total weight) from vertex 1 to all other 21 vertices. The output for the specified destinations is:

- Distance of 6 from 1 is 6
- Distance of 8 from 1 is 4
- Distance of 9 from 1 is 3
- Distance of 15 from 1 is 6
- Distance of 16 from 1 is 4
- Distance of 22 from 1 is 5

These distances represent the "cost" of the most likely (lowest cost) escape route. A lower number indicates a more "desirable" path. Based on this, the route to vertex 9 (cost 3) is the most likely, while the routes to vertices 6 and 15 (cost 6) are the least likely among this list.

By implementing Dijkstra's algorithm with a min-priority queue, we have efficiently solved the single-source shortest path problem.

Make sure that you derive the solutions to this homework by yourself without any outside help. Searching for solutions on the internet or asking any form of AI is not allowed. Write the solutions in your own words. Use version control for your program development and be prepared to show and explain any version of your code.

Checklist:

✓ Did you add your name?
✓ Did you disclose all resources that you have used?
(This includes all people, books, websites, etc. that you have consulted)
✓ Did you sign that you followed the Aggie honor code?
✓ Did you solve all problems?
✓ Did you typeset your answers entirely in LaTeX?
✓ Did you submit the pdf file of your homework?