COMP 6651 Algorithm Design Techniques

Assignment 3 Answers

Name : Parsa Kamalipour , *StudentID* : 40310734

Exercise 1: (20 marks)

Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, \ldots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V : u \rightsquigarrow v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, that is, $\min(u)$ is the vertex v such that

$$L(v) = \min\{L(w) : w \in R(u)\}.$$

(a) Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

(b) Use a proof by contradiction to show that the algorithm is correct.

Answers to Exercise 1

To solve this problem, I need to find the minimum labeled vertex that is reachable from each vertex in the directed graph. I want to do this in $O(V + E)$ time. So my plan is to use Depth-First Search (DFS) or Breadth-First Search (BFS) for each vertex, to traverse the reachable vertices and determine the one with the smallest label. To make sure this approach is efficient, I have to be careful about visiting each vertex and edge only once.

Here is what I do:

Initialization

First, I create an array called `min_label` of size $|V|$ to store the minimum reachable label for each vertex. I set `min_label`[u] to $L(u)$ for every vertex $u \in V$. This way, every vertex starts with its own label as the minimum.

Graph Traversal

For every vertex $u \in V$, I use DFS to explore all vertices reachable from u. During the traversal, I keep track of the minimum label I find among those reachable vertices.

Pseudocode

Here is the pseudocode I wrote for solving this problem:

```
def compute_min_labels(graph, labels):
    # Number of vertices
    V = len(graph)

    # Initialize the result array with the label of each vertex itself
    min_label = labels[:]

    # Function to perform DFS and update min_label
    def dfs(u, visited, current_min):
        visited[u] = True
        current_min[0] = min(current_min[0], labels[u])

        for v in graph[u]:
            if not visited[v]:
                dfs(v, visited, current_min)

    # Iterate over each vertex to calculate min(u) for each u
    for u in range(V):
```

```
        visited = [False] * V
        current_min = [labels[u]]  # Use list to allow updates within dfs
        dfs(u, visited, current_min)
        min_label[u] = current_min[0]

    return min_label
```

Explanation

- **Graph Representation**: I represent the graph as an adjacency list where `graph`[u] is a list of vertices that are directly reachable from vertex u.
- **Labels Array**: The `labels` array stores the label $L(u)$ for each vertex u.
- **DFS Traversal**: For each vertex u, I perform DFS to explore all reachable vertices and find the smallest label among them.
- **Complexity**:
  - The DFS runs in $O(V + E)$ time for each component of the graph.
  - Since I run DFS for every vertex, and every edge and vertex is visited only once in total, the overall complexity is $O(V + E)$.

Proof of Correctness (Part b)

To prove correctness of my algorithm using proof by contradiction:

Suppose there is a vertex $u \in V$ for which the algorithm does not find the correct minimum labeled reachable vertex. This means that there is a vertex $v \in R(u)$ such that $L(v) < \min(u)$, but the algorithm somehow failed to find it.

However, since my algorithm performs a complete DFS from every vertex u and keeps track of the minimum label seen during traversal, it will always find the vertex with the smallest label among the reachable vertices. This contradicts the assumption, which means my algorithm correctly computes $\min(u)$ for all vertices.

Exercise 2: (15 marks)

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$.

1. How fast can you make Prims algorithm run?
2. What if the edge weights are integers in the range from 1 to W, for some constant W?

Answers to Exercise 2

For **Problem 2**, I need to determine how I could make Prim's algorithm more efficient when edge weights are in specific ranges. Here is how I approached this problem.

Part (a): Edge Weights in the Range 1 to $|V|$

If all the edge weights in the graph are between 1 and $|V|$, I can make Prims algorithm more efficient by using an **array of lists** as a priority queue:

1. **Data Structure: Array of Lists**
   I can use an array where each index represents an edge weight. Since weights are between 1 and $|V|$, the array will have $|V| + 1$ slots. Each slot contains a list of edges with that specific weight.
   - **Insert Edge**: I just add the edge to the list at the correct index based on its weight.
   - **Decrease-Key**: If I need to decrease the weight of an edge, I remove it from its current list and add it to the list at the new weight. Since each list is implemented as a doubly linked list, both

removal and insertion take $O(1)$ time.

- **Extract-Min**: To extract the edge with minimum weight, I maintain a linked list of non-empty indices. This lets me efficiently find the smallest weight by directly going to the first non-empty index, so this operation takes $O(1)$ time.

2. **Algorithm Complexity**

- **Initialization**: Setting up the array of lists takes $O(V)$ time.
- **Extract-Min and Decrease-Key**: Both of these operations take constant time, $O(1)$, because I am using the linked list to track the non-empty indices.
- Therefore, the overall time complexity of Prim's algorithm in this scenario becomes $O(V+E)$. Since every operation on the priority queue is performed in constant time, this makes it as efficient as possible given the input.

Part (b): Edge Weights in the Range 1 to W

If the edge weights are in range from 1 to W, where W is a constant, I can use a more sophisticated data structure called a **van Emde Boas (vEB) tree**:

1. **Data Structure: Van Emde Boas Tree**
   A **vEB tree** works well when the keys are in a small, bounded universe, which is exactly the case here with edge weights between 1 and W.

- **Insert Edge**: I insert the edge into the vEB tree, which takes $O(\log \log W)$ time.
- **Decrease-Key**: If an edges weight decreases, I can move it in the vEB tree, which also takes $O(\log \log W)$ time.
- **Extract-Min**: Extracting the minimum-weight edge from the vEB tree takes $O(\log \log W)$ time.

2. **Algorithm Complexity**

- **Initialization**: Setting up the vEB tree takes $O(W)$ time, but since W is assumed to be a constant, this is effectively $O(1)$.
- **Extract-Min and Decrease-Key**: These operations are both done in $O(\log \log W)$ time.
- As a result, the total time complexity for Prims algorithm in this scenario becomes $O((V + E) \log \log W)$. This is much more efficient for managing priority queue operations compared to the traditional approach when edge weights are bounded.

Exercise 3: (10 marks)

Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by `INITIALIZE-SINGLE-SOURCE`(G, s). Prove that there exists a sequence of $|V| - 1$ relaxation steps that produces $v.d = \delta(s, v)$ for all $v \in V$.

Answers to Exercise 3

To solve **Problem 3**, I need to prove that for a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, I can find the shortest path distances from a source vertex s to all other vertices by doing a sequence of $|V| - 1$ relaxation steps.

Here is my reasoning.

Concepts I Used

1. **Relaxation**
   In graph algorithms, **relaxation** is a process I use to improve the current estimate of the shortest path

distance to a vertex. Specifically, if there is an edge $(u, v) \in E$ with weight $w(u, v)$, and if the current distance estimate to vertex v is greater than the distance to vertex u plus the weight of edge (u, v), then I update vs distance like this:

$$v.d = \min(v.d, u.d + w(u, v))$$

2. **Initialization with INITIALIZE-SINGLE-SOURCE**
   When I run `INITIALIZE-SINGLE-SOURCE`(G, s), it sets the initial distance estimate $v.d$ to **infinity** (∞) for all vertices $v \in V$, except for the source vertex s, which is set to 0:

$$s.d = 0, \quad v.d = \infty \quad \forall v \neq s$$

3. **Shortest Path Property**
   The **shortest path property** tells me that if a vertex v is reachable from the source s, I will eventually find the shortest path distance from s to v by relaxing the edges repeatedly until I reach the minimum possible distances.

Proof Outline

The **Bellman-Ford algorithm** can be used to find the shortest paths in a graph with negative-weight edges, as long as there are no negative-weight cycles. In Bellman-Ford, I perform a sequence of relaxation steps on the graph.

The main idea is that, for a graph with $|V|$ vertices, if I perform $|V| - 1$ iterations of relaxing all edges, I can make sure that I find the shortest path distance from the source vertex s to all other vertices. Here is why this works:

Path Length

- In the graph, the **longest possible simple path** from the source s to any other vertex has at most $|V| - 1$ edges. Since a simple path cannot visit any vertex more than once, and there are $|V|$ vertices in total, I know the longest possible path is limited to $|V| - 1$ edges.
- This means that any vertex v reachable from s can be reached by a path that has at most $|V| - 1$ edges.

Relaxation in $|V| - 1$ Iterations

- In each iteration, I relax all the edges in the graph.
- In the **first iteration**, I ensure that the shortest path from s to any vertex directly connected to s is correct.
- In the **second iteration**, I propagate the shortest paths further, considering vertices that are two edges away from s, and so on.
- After $|V|-1$ iterations, every vertex v that is reachable from s will have the correct shortest path distance, meaning $v.d = \delta(s, v)$.

No Negative-Weight Cycles

- Since I know that the graph does not have negative-weight cycles, I can be confident that the shortest path estimates will stabilize after $|V| - 1$ iterations.
- If there were negative-weight cycles, further relaxation would keep reducing the distance estimates indefinitely, which isnt the case here.

Detailed Proof

1. **Initialization**
   I start by running `INITIALIZE-SINGLE-SOURCE`(G, s), which sets $s.d = 0$ and $v.d = \infty$ for all $v \neq s$.

2. **Relaxation Steps**
   I perform a sequence of $|V| - 1$ relaxation steps, where in each step, I relax all the edges in the graph. During each relaxation step, for each edge $(u, v) \in E$, I check if the current distance estimate to vertex v can be improved by going through vertex u:

$$\text{if } v.d > u.d + w(u, v) \text{ then } v.d = u.d + w(u, v)$$

3. **Correctness After $|V| - 1$ Iterations**
   After $|V| - 1$ iterations, I can be sure that all vertices reachable from the source have the correct distance estimates. Each iteration guarantees that the shortest path using up to k edges is found, and $|V| - 1$ iterations are enough to cover the longest possible simple path (which has at most $|V| - 1$ edges).

The key idea here is that by doing $|V| - 1$ iterations of relaxing all edges, I can propagate the shortest path distances from the source vertex s to all other vertices. Since the graph contains no negative-weight cycles, the distance estimates will stabilize, and I will get the correct shortest path distances for all vertices reachable from s. Thus, after $|V| - 1$ relaxation steps, $v.d = \delta(s, v)$ for all $v \in V$.

Exercise 4: (20 marks)

The Bellman-Ford algorithm does not specify the order in which to relax edges in each pass. Consider the following method for deciding upon the order. Before the first pass, assign an arbitrary linear order $v_1, v_2, \ldots, v_{|V|}$ to the vertices of the input graph $G = (V, E)$. Then partition the edge set E into $E_f \cup E_b$, where:

$$E_f = \{(v_i, v_j) \in E : i < j\}$$
$$E_b = \{(v_i, v_j) \in E : i > j\}$$

(Assume that G contains no self-loops, so that every edge belongs to either E_f or E_b.) Define $G_f = (V, E_f)$ and $G_b = (V, E_b)$.

(a) Prove that G_f is acyclic with topological sort $\langle v_1, v_2, \ldots, v_{|V|} \rangle$ and that G_b is acyclic with topological sort $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.

Suppose that each pass of the Bellman-Ford algorithm relaxes edges in the following way. First, visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$, relaxing edges of E_f that leave the vertex. Then visit each vertex in the order $v_{|V|}, v_{|V|-1}, \ldots, v_1$, relaxing edges of E_b that leave the vertex.

(b) Prove that with this scheme, if G contains no negative-weight cycles that are reachable from the source vertex s, then after only $\lceil |V|/2 \rceil$ passes over the edges, $v.d = \delta(s, v)$ for all vertices $v \in V$.

(c) Does this scheme improve the asymptotic running time of the Bellman-Ford algorithm?

Answers to Exercise 4

For **Problem 4**, I need to analyze a modified version of the **Bellman-Ford algorithm**, where edges are partitioned and relaxed in a specific order. Here is how I approached each part of the problem.

Part (a): Proving G_f and G_b Are Acyclic

To solve part (a), I need to prove that both G_f and G_b are **acyclic** and provide a **topological sort** for each.

1. **Graph G_f**
   In G_f, I have edges of the form (v_i, v_j) where $i < j$. This means that all edges in G_f point from a vertex earlier in the order to a vertex later in the order. Since all edges point "forward" in the linear order, its impossible for G_f to have a cycle. Any cycle would need an edge that goes "backward" in the order,

which is not allowed by the definition of E_f.

Therefore, I can say that G_f is **acyclic**. The **topological sort** for G_f is simply the given vertex order: $\langle v_1, v_2, \ldots, v_{|V|} \rangle$.

2. **Graph G_b**
In G_b, I have edges of the form (v_i, v_j) where $i > j$. This means that all edges in G_b point from a vertex later in the order to a vertex earlier in the order. Since all edges point "backward" in the linear order, it is also impossible for G_b to have a cycle. Any cycle would need an edge going "forward" in the order, which is not allowed by the definition of E_b.

   Therefore, I conclude that G_b is **acyclic**. The **topological sort** for G_b is the reverse of the original order: $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.

Part (b): Proving Correctness of the Relaxation Scheme

In part (b), I need to prove that if I use the given relaxation scheme, and if graph G contains no negative-weight cycles reachable from the source vertex s, then after $\lceil |V|/2 \rceil$ passes, all vertices $v \in V$ will have the correct shortest path distance from s.

1. **Relaxation Scheme**
The relaxation scheme has two parts in each pass:

   (a) First, I visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$ and relax the edges in E_f that leave the vertex.

   (b) Then, I visit each vertex in the reverse order $v_{|V|}, v_{|V|-1}, \ldots, v_1$ and relax the edges in E_b that leave the vertex.

   This way, I make sure that all edges in both E_f and E_b are relaxed during each pass.

2. **Number of Passes**
In the standard **Bellman-Ford algorithm**, I usually need $|V| - 1$ passes over all edges to guarantee correctness. In the given relaxation scheme, each pass consists of two phases: one for relaxing edges in E_f and one for relaxing edges in E_b. By visiting vertices in both forward and backward orders, I propagate shortest path information more effectively.

   Since relaxation happens in both directions, it effectively doubles the progress compared to a single relaxation. So, the shortest paths can be found in only $\lceil |V|/2 \rceil$ passes instead of $|V| - 1$ passes.

3. **Conclusion**
After only $\lceil |V|/2 \rceil$ passes, I can be confident that the distance estimates $v.d$ are equal to the shortest path distances $\delta(s, v)$ for all vertices $v \in V$.

Part (c): Asymptotic Running Time

In part (c), I need to determine whether this scheme **improves the asymptotic running time** of the Bellman-Ford algorithm.

1. **Standard Bellman-Ford Complexity**
The standard Bellman-Ford algorithm runs in $O(V \cdot E)$ time, because I need $|V| - 1$ passes over all edges.

2. **Modified Scheme Complexity**
In the modified scheme, I need only $\lceil |V|/2 \rceil$ passes over all edges. Each pass still involves relaxing all edges, which takes $O(E)$ time. Therefore, the total running time of the modified scheme is $O((|V|/2) \cdot E) = O(V \cdot E)$.

3. **Conclusion**
The **asymptotic running time** remains $O(V \cdot E)$, which means that there is no improvement in the worst-case complexity. However, in practice, the number of passes required is reduced, which can improve the **practical performance** of the algorithm, making it converge faster.

Exercise 5: (17 marks)

Suppose that it takes $f(|V|, |E|)$ time to compute the transitive closure of a directed acyclic graph, where f is a monotonically increasing function of both $|V|$ and $|E|$. Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is then $f(|V|, |E|) + O(V + E^*)$.

Answers to Exercise 5

To solve Question 5, I decide to break the problem into easier steps.

Step 1: Compute the Strongly Connected Components (SCCs)

First, I need to compute the **strongly connected components (SCCs)** of the graph G. To do this, I used **Tarjan's Algorithm** or **Kosaraju's Algorithm**, both of which have time complexity $O(V + E)$. I denote SCCs as $S_1, S_2, \ldots, S_k$, where each S_i represent one SCC of G.

Once I have SCCs, I create a **condensed graph** G', where:

- Each SCC is represented as one "supervertex".
- There is edge between two supervertices in G' if there was edge between any vertices in their respective SCCs in G.

This new condensed graph G' is a **DAG** because there is no cycles between SCCs.

Step 2: Compute Transitive Closure of the Condensed Graph

After that, I compute **transitive closure** of the condensed graph G'. Since G' is a **DAG**, I can use function $f(|V'|, |E'|)$ for this.

Here, $|V'|$ is number of SCCs and $|E'|$ is number of edges between those SCCs. Since G' has fewer vertices and edges compared to original graph, $f(|V'|, |E'|) \leq f(|V|, |E|)$. So its still within the bounds of the given function.

Step 3: Add Intra-SCC Edges

After computing transitive closure of G', I need to **handle intra-SCC reachability**. For each SCC, every vertex is reachable from every other vertex inside the SCC. So, I need to add edges to make every SCC a **complete subgraph**.

Adding these edges takes $O(V+E^*)$ time, where E^* represent total number of edges in transitive closure.

Final Time Complexity

- **Step 1** (computing SCCs) takes $O(V + E)$ time.
- **Step 2** (computing transitive closure of DAG G') takes $f(|V|, |E|)$ time.
- **Step 3** (adding intra-SCC edges) takes $O(V + E^*)$ time.

So, total time complexity to compute transitive closure of the general directed graph G is:

$$f(|V|, |E|) + O(V + E^*)$$

This match with the required time complexity in the question.

Conclusion

To summarize, I break down the solution by:

1. Decomposing the graph into SCCs to form a condensed DAG.
2. Computing transitive closure of that condensed DAG.

3. Adding edges within each SCC to make sure it fully connect.

The final result is time complexity of $f(|V|, |E|) + O(V + E^*)$, which is what the problem is asking for.

Exercise 6: (18 marks)

The Edmonds-Karp algorithm implements the Ford-Fulkerson algorithm by always choosing a shortest augmenting path in the residual network. Suppose instead that the Ford-Fulkerson algorithm chooses a widest augmenting path: an augmenting path with the greatest residual capacity. Assume that $G = (V, E)$ is a flow network with source s and sink t, that all capacities are integer, and that the largest capacity is C. In this problem, you will show that choosing a widest augmenting path results in at most $|E| \ln |f^*|$ augmentations to find a maximum flow f^*.

(a) Show how to adjust Dijkstras algorithm to find the widest augmenting path in the residual network.

(b) Show that a maximum flow in G can be formed by successive flow augmentations along at most $|E|$ paths from s to t.

(c) Given a flow f, argue that the residual network G_f has an augmenting path p with residual capacity $c_f(p) \geq \frac{|f^*|-|f|}{|E|}$.

(d) Assuming that each augmenting path is a widest augmenting path, let f_i be the flow after augmenting the flow by the i-th augmenting path, where f_0 has $f(u, v) = 0$ for all edges (u, v). Show that $|f^*| - |f_i| \leq |f^*|(1 - \frac{1}{|E|})^i$.

(e) Show that $|f^*| - |f_i| < |f^*|e^{-i/|E|}$. You may want to use inequality (3.14) from the text,

$$1 + x < e^x \quad \text{for all } x \neq 0$$

(f) Conclude that after the flow is augmented at most $|E| \ln |f^*|$ times, the flow is a maximum flow.

Answers to Exercise 6

To solve Question 6, I will go step by step. This problem involves modifying Ford-Fulkerson algorithm to find a maximum flow by always choosing the widest augmenting path instead of shortest one, like Edmonds-Karp algorithm does.

Problem Overview

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method that always chooses shortest augmenting path, ensuring that the number of augmentations is polynomial. In this problem, I need to see what happens if I always choose **widest augmenting path** meaning, the path with the greatest residual capacity. I will show how this change affects the number of augmentations required to find maximum flow.

Let's solve step-by-step.

Part (a): Adjust Dijkstras Algorithm to Find Widest Augmenting Path

To find the widest augmenting path, I need to modify **Dijkstras algorithm** to use a **maximum capacity approach** instead of shortest path approach. Heres how I do it:

1. **Data Structure**:
   - Instead of using a priority queue that orders by distance, I use a **maximum heap** (or a priority queue that orders by maximum capacity).
2. **Initialization**:
   - Set the capacity of source s to be infinity: capacity$[s] = \infty$.

- Set capacity of all other vertices to be 0.

3. **Relaxation**:
   - During relaxation, update the capacity of each neighbor v of the current vertex u like this:

     $$\text{capacity}[v] = \max(\text{capacity}[v], \min(\text{capacity}[u], c_{uv}))$$

     if the minimum capacity along path from s to v through u is greater than current capacity of v.
   - Where c_{uv} is capacity of edge from u to v.

This modified Dijkstra's algorithm will always select the augmenting path with the **maximum bottleneck capacity**.

Part (b): Maximum Flow with At Most $|E|$ Paths

In this part, I need to show that maximum flow in G can be formed by augmenting flow along at most $|E|$ paths from source s to sink t.

- The property of the **widest augmenting path** is that every time I add flow along this path, the maximum residual capacity of at least one edge along that path is reduced.
- Since I am choosing paths with the **widest possible capacity** each time, after at most $|E|$ paths, all edges with the largest capacity will be reduced, and I will either reach maximum flow or have reduced maximum capacity in path.
- Hence, by augmenting along at most $|E|$ paths, I can find the maximum flow.

Part (c): Residual Network and Augmenting Path Capacity

Let f be the current flow, and let G_f be the corresponding **residual network**. I need to argue that there exists an augmenting path p in G_f with **residual capacity** $c_f(p) \geq \frac{|f^*|-|f|}{|E|}$.

- Here, f^* is the maximum flow, and f is current flow. The difference $|f^*| - |f|$ shows how much flow still needs to be added to reach maximum flow.
- Since I am always choosing the widest augmenting path, and there are at most $|E|$ edges involved, by the **pigeonhole principle**, there must be at least one path p whose residual capacity satisfies:

$$c_f(p) \geq \frac{|f^*| - |f|}{|E|}$$

Part (d): Flow After i-th Augmentation

Let f_i be the flow after the i-th augmentation. I need to show:

$$|f^*| - |f_i| \leq |f^*| \left(1 - \frac{1}{|E|}\right)^i$$

- Each time I augment along the **widest path**, the residual capacity of at least one edge is reduced.
- Since I am choosing the widest path, the flow added in each step is at least a fraction $\frac{1}{|E|}$ of the remaining difference $|f^*| - |f|$.
- Therefore, after i augmentations, the remaining difference $|f^*| - |f_i|$ is at most:

$$|f^*| \left(1 - \frac{1}{|E|}\right)^i$$

Part (e): Inequality Using Exponential Bound

Now, I need to show:

$$|f^*| - |f_i| < |f^*|e^{-i/|E|}$$

For this, I use the inequality:

$$1 + x < e^x \quad \text{for all } x \neq 0$$

Here, $x = -\frac{1}{|E|}$. Applying this inequality, I get:

$$\left(1 - \frac{1}{|E|}\right)^i < e^{-i/|E|}$$

Multiplying both sides by $|f^*|$, I have:

$$|f^*| \left(1 - \frac{1}{|E|}\right)^i < |f^*|e^{-i/|E|}$$

Thus:

$$|f^*| - |f_i| < |f^*|e^{-i/|E|}$$

Part (f): Conclude Maximum Flow After $|E| \ln |f^*|$ Augmentations

Finally, I need to conclude that after at most $|E| \ln |f^*|$ augmentations, the flow is maximum flow.

- From part (e):
$$|f^*| - |f_i| < |f^*|e^{-i/|E|}$$
- To make $|f^*| - |f_i| < 1$ (to reach maximum flow), I need:
$$e^{-i/|E|} < \frac{1}{|f^*|}$$
- Taking natural logarithm on both sides:
$$-\frac{i}{|E|} < \ln\left(\frac{1}{|f^*|}\right)$$
$$i > |E| \ln |f^*|$$
- Therefore, after at most $|E| \ln |f^*|$ augmentations, I will reach maximum flow.

Acknowledgement

I would like to acknowledge that I used ChatGPT to help refine my grammar and sentence structure in this document. However, the solutions and ideas presented here are entirely my own.