This paper examines an algorithm that optimizes a spanning tree based on a given objective function. We analyze its correctness and show that the algorithm has a runtime complexity of $\Theta(n^2)$ and space complexity of $\Theta(n)$. 
Table of Contents
Introduction........................................................................................................................................... 1
Problem.................................................................................................................................................. 1
Algorithm............................................................................................................................................. 1
Proof of correctness ............................................................................................................................. 3
Complexity analysis............................................................................................................................... 5
  Time complexity .................................................................................................................................. 5
  Space complexity .............................................................................................................................. 5
Conclusion ............................................................................................................................................ 5
Reference .............................................................................................................................................. 6
Introduction

In this paper we design and analyze an algorithm to optimize a spanning tree of a weighted graph. In the first section we discuss this optimization problem and introduce an objective function that dictates how the optimization will be done. In the next section we describe the algorithm used to solve this problem. Following that, we look at a proof to show the algorithms correctness and then complexity analysis of its runtime and space requirements.

Problem

The optimization problem as stated is a simple problem to understand. First let us state with a few assumptions and definitions.

Assume that a given undirected graph \( G = (V, E) \) is simple and connected. Each vertex \( v \in V \) is associated with a weight \( q(v) \in \mathbb{N} \) that represents a frequency of visiting \( v \), where \( \mathbb{N} \) denotes the set of natural numbers.

Assume that a tree \( T = (V'E') \) is a subgraph of \( G \), i.e., \( V' \subseteq V \) and \( E' \subseteq E \). Let \( r \) be the root of \( T \) and \( p(r, v) \) denote the path from the root \( r \) to vertex \( v \in V' \) in \( T \). Let \( |p(r, v)| \) denote the length of the path \( p(r, v) \) from \( r \) to \( v \) in \( T \).

Using these definitions and assumptions, we can describe the problem as follows.

**Input:** A simple, connected, weighted, undirected graph \( G = (V, E) \) with weight \( q: V \to \mathbb{N} \)

**Output:** A spanning tree \( T = (V, E) \) of \( G \) rooted at \( r \in V \)

**Objective Function:** \( f(T, r) = \sum_{v \in V} q(v)|p(r, v)| \to \min \)

Algorithm

In this algorithm there are a few data structures that must first be defined so that there is an understanding of what is being done. There are adjacency lists, queues, and arrays used in this algorithm.

**Adjacency List:** An adjacency list is graph representation consisting of an array \( \text{Adj} \) of \( |V| \) lists, one for each vertex in \( V \). For each \( u \in V \), \( \text{Adj}[u] \) contains all the vertices \( v \) such that there is an edge \( (u, v) \in E \). \( \text{Adj}[u] \) consists of all the vertices adjacent to \( u \) in the graph. [1]
**Operations**

`addEdge(int u, int v)`  
{Adds an edge to this graph.}

`addVertex(int index, String name)`  
{Adds a vertex to this graph.}

`getVertex(int index)`  
{Returns the vertex with a given index.}

`makeEmptyGraph(int v, boolean directed)`  
{Creates and returns an empty AdjacencyListGraph with no edges, given the number of vertices and a boolean indicating whether the graph is directed.}

**Queue**: A queue is a dynamic set in which the element removed from the set is always the one that has been in the set for the longest time: the queue implements a first-in, first-out policy. [2]

**Operations**

`dequeue()`  
{Returns and removes the object at the head of the queue.}

`enqueue(Object x)`  
{Adds an object to the tail of the queue.}

**Array**: An array is a data structure consisting of a group of elements that are accessed by indexing. Each element has the same data type and the array occupies a contiguous area of storage.

The algorithm is designed using a breadth first search in order to find the shortest path to each vertex and the minimal value of $f(T, r)$. For each vertex in the graph, we do a breadth first search to create the minimal spanning tree and get the minimal weight. Then we store these values. On the next iteration, we compare the value of the current weight to the previous weight. If the current weight is less than the stored one, we swap the old weight with the new one and save both the current tree and current weight as the minimal values. Below is a pseudo code representation of the algorithm. The breadth first search code is referenced from “Introduction To Algorithms: Second Edition”. [3]

```
1 Algorithm(G, n) { //G is the graph given and n is the number of vertices in the graph.
2    min ← NULL; //Is the smallest value of f(T, r).
3    minTree ← NULL; //Adjacency list representation of the tree that has the smallest f(T, r).
4    s ← 0; //s is the starting root.
5    graph ← G;
6    while s < n {
7        tempTree ← NULL; // Adjacency list representation of the tree with min value so far.
```
BFS(graph, s) //Breadth first search
for each vertex u ∈ V – {s} { //u is a vertex of the graph G.
     color [u] ← white; //An array that stores the color of u.
     d[u] ← ∞; //An array that stores the distance to u.
     π[u] ← NULL; //An array that stores the predecessor of u.
}
color[s] ← gray;
d[s] ← 0;
π[s] ← NULL;
w ← f[s] * d[s]; //Integer that stores the min so far. f[s] is the weight of the vertex
     //and d[s] is the distance from the root.
Q ← 0; //Queue that is used to sort vertices.
Enqueue(Q, s); //Adds to queue.
while Q ≠ 0 {
    u ← Dequeue(Q); //removes from the queue.
    for each v ∈ Adj[u] { //v is a vertex that is in the adjacency list at index u.
        if color[v] = white then {
            color[v] ← gray;
            d[v] ← d[u] + 1;
            w ← w + (f[v] * d[v]);
            π[v] ← u;
            Add v to tempTree(u); //tempTree is an adjacency list and v is added at u.
            Enqueue(Q, v);
        }
    }
    Color[u] ← black;
}
s++;
if w < min or w = NULL then {
    min ← w;
    minTree = tempTree; //minTree points to the same tree as tempTree.
}
return minTree;

Proof of correctness

First let us prove that the breadth first search portion of our code finds the shortest path between the root of the tree and any other vertex in tree. This is important because finding the shortest path means that for each vertex, q(v)|p(r, v)| is the smallest value for the tree. Therefore f(T, r) is the smallest value for the tree. Again using “Introduction To Algorithms: Second Edition” as reference, we can use their proof to show the correctness of this portion of the algorithm. [4]
Lemma 22.1

Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.

Lemma 22.2

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on $G$ from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $d[v]$ computed by BFS satisfies $d[v] \geq \delta(s, v)$.

Corollary 22.4

Suppose that vertices $v_i$ and $v_j$ are enqueued during the execution of BFS, and that $v_i$ is enqueued before $v_j$. Then $d[v_i] \leq d[v_j]$ at the time that $v_j$ is enqueued.

Theorem 22.5 (Correctness of breadth first search)

Let $G = (V,E)$ be a directed or undirected graph, and suppose that BFS is run on $G$ from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source $s$, and upon termination, $d[v] = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from $s$, one of the shortest paths from $s$ to $v$ is a shortest path from $s$ to $\pi[v]$ followed by the edge $(\pi[v], v)$.

Proof: Assume, for the purpose of contradiction, that some vertex receives a $d$ value not equal to its shortest path distance. Let $v$ be the vertex with minimum $\delta(s, v)$ that receives such an incorrect $d$ value; clearly $v \neq s$. By Lemma 22.2, $d[v] \geq \delta(s, v)$, and thus we have that $d[v] > \delta(s, v)$. Vertex $v$ must be reachable from $s$, so if it is not, then $\delta(s, v) = \infty \geq d[v]$. Let $u$ be the vertex immediately preceding $v$ on a shortest path from $s$ to $v$, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$, and because of how we chose $v$, we have $d[u] = \delta(s, u)$. Putting these properties together, we have $d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$.

Now consider the time when BFS chooses to dequeue vertex $u$ from $Q$ in line 22. At this time, vertex $v$ is either white, gray, or black. We shall show that in each of these cases, we derive a contradiction to inequality (22.1). If $v$ is white, then line 26 sets $d[v] = d[u] + 1$, contradicting inequality (22.1). If $v$ is black, then it was already removed from the queue, and by Corollary 22.4, we have $d[v] \leq d[u]$, again contradicting inequality (22.1). If $v$ is gray, then it was painted gray upon dequeuing some vertex $w$, which was removed from $Q$ earlier than $u$ and for which $d[v] = d[w] + 1$. By Corollary 22.4, however, $d[w] \leq d[u]$, and so we have $d[v] \leq d[u] + 1$, once again contradicting inequality (22.1).

Thus we conclude that $d[v] = \delta(s, v)$ for all $v \in V$. All vertices reachable from $s$ must be discovered, for if they were not, they would have infinite $d$ values. To conclude the proof
of the theorem, observe that if \( \pi[v] = u \), then \( d[v] = d[u] + 1 \). Thus, we can obtain a shortest path from \( s \) to \( v \) by taking a shortest path from \( s \) to \( \pi[v] \) and then traversing the edge \((\pi[v], v)\).

Now that we have shown that breath first search finds the shortest path and thereby generates the smallest weight; let us show that out of all the weights generated the one stored in the end by the algorithm is the smallest.

**Proof:** Assume that the stored minimum weight \( m \) at the termination of the algorithm is not the smallest weight and there exists another weight \( w \) such that \( w < m \). If \( w \) does exist then it would contradict with line 37 in our code. If conditions are met, line 38 will always store the smallest weight that the algorithm generates. Thus, the algorithm will always terminate with the smallest weight value.

**Complexity analysis**

**Time complexity**

To start, we first look at the main portion of the algorithm, the breadth first search. The operations to enqueue, dequeue and add to an adjacency list are constant time operations \( \Theta(1) \). However, since we must enqueue all the vertices, the time it takes in total for enqueues is \( \Theta(V) \). Each time a vertex is dequeued its adjacency list is scanned. In total all adjacency lists are scanned once and the sum of those lists is equal to \( \Theta(E) \). Overall, breadth first search has a linear runtime of \( \Theta(V + E) \). [5]

Outside of the breadth first search we have the compare operation on line 35 and the outer while loop. The comparison of weights is constant \( \Theta(1) \). The while loop is linear to the number of vertices so \( \Theta(V) \). So in total the runtime for this algorithm is \( \Theta(V(V + E)) \). Simplified we get a runtime of \( \Theta(n^2) \).

**Space complexity**

In this algorithm we have many data structures. We have the adjacency list, the queue, and the array. Adjacency lists take up space proportional to the number of vertices and the number of edges \( \Theta(V + E) \). Queues can take up as much space as \( \Theta(V) \). Similarly, arrays have a space complexity of \( \Theta(V) \). Overall, 3 adjacency lists are used to store the graph, minTree, and tempTree. We also have several arrays and the queue. In total we would have a complexity of \( \Theta(7V + 4E) \). Simplified we get a space complexity of \( \Theta(V + E) \).

**Conclusion**

In this paper, we have seen the design and analysis of an algorithm that optimizes a spanning tree of a weighted graph, give a specific objective function. We have shown that the algorithm has a
runtime complexity of $\Theta(n^2)$ and a space complexity of $\Theta(V + E)$. Also we have proven its correctness to return the minimal spanning tree and minimal weight value.

**Reference**


