Algorithm Complexity Analysis

Team Computer Man

May 15, 2024

1 primMST Function

1.1 Code

Listing 1: primMST Function

```
int primMST(const std::vector<Stadium>& graph, std::unordered_map<QString, QString>& parent) {
    int totalDistance = 0;

    std::priority_queue<std::pair<int, QString>, std::vector<std::pair<int, QString>>, std::greater<>> pq;

    std::unordered_set<QString> visited;
    pq.push({0, graph[0].name});

    while (!pq.empty()) {
        auto [weight, currentStadium] = pq.top();
        pq.pop();

        if (visited.find(currentStadium) != visited.end()) {
            continue;
        }

        totalDistance += weight;
        visited.insert(currentStadium);

        for (const auto& [neighbor, neighborWeight] : graph[0].connections) {
            if (visited.find(neighbor) == visited.end()) {
                pq.push({neighborWeight, neighbor});
                parent[neighbor] = currentStadium;
            }
        }
    }
```

```
    return totalDistance;
}
```

1.2 Complexity Analysis

The provided function implements Prim's algorithm for finding the minimum spanning tree (MST). Here's the breakdown of its complexity analysis:

- The priority queue operations (push and pop) have a complexity of $O(\log(n))$ where n is the number of elements in the queue.
- The while loop iterates over each element in the priority queue, resulting in a complexity of $O(n)$.
- Thus, the overall complexity is $O(n \log(n))$ due to the priority queue operations being the dominant factor.

2 dfs Function

2.1 Code

Listing 2: dfs Function

```
int dfs(const QMap<QString, Stadium>& graph, const QString& startStadium, QMap<QString, bool>& visited) {
    visited[startStadium] = true;
    int totalDistance = 0;

    qDebug() << "Visited:" << startStadium;

    auto it = graph.find(startStadium);
    if (it == graph.end()) {
        qWarning() << "Error:␣Starting␣stadium␣not␣found␣in␣the␣graph.";
        return totalDistance;
    }

    for (const auto& adjacentStadium : it.value().connections.keys()) {
        if (!visited[adjacentStadium]) {
            int distance = it.value().connections.value(adjacentStadium);
            qDebug() << "Traversing␣from" << startStadium << " to" << adjacentStadium << "(Distance:" << distance << "miles)";
```

```
            totalDistance += distance + dfs(graph,
                adjacentStadium, visited);
        }
    }

    return totalDistance;
}
```

2.2 Complexity Analysis

The provided function implements depth-first search (DFS) traversal. Here's the complexity analysis:

- The function uses recursion to traverse each vertex once, resulting in a time complexity of $O(n)$, where n is the number of vertices.
- Thus, the overall complexity is $O(n)$.

3 bfs Function

3.1 Code

Listing 3: bfs Function

```
int bfs(const unordered_map<string, Stadium>& graph, const
    string& startStadium) {
    unordered_map<string, int> mileage;
    queue<string> q;
    for (const auto& entry : graph) {
        mileage[entry.first] = numeric_limits<int>::max();
    }
    q.push(startStadium);
    mileage[startStadium] = 0;
    int totalDistance = 0;
    while (!q.empty()) {
        string currentStadium = q.front();
        q.pop();
        for (const auto& [adjacentStadium, distance] : graph.
            at(currentStadium).connections) {
            if (mileage[currentStadium] + distance < mileage[
                adjacentStadium]) {
                mileage[adjacentStadium] = mileage[
                    currentStadium] + distance;
                q.push(adjacentStadium);
                totalDistance += distance;
```

```
            }
        }
    }
    return totalDistance;
}
```

3.2 Complexity Analysis

The provided function implements breadth-first search (BFS) traversal. Here's the complexity analysis:

- The function contains a while loop that iterates through each vertex, resulting in a time complexity of $O(n)$ where n is the number of vertices.
- Inside the while loop, there is another loop that iterates over each neighbor of the current vertex. This loop has a time complexity of $O(m)$ where m is the number of edges.
- Since each vertex and edge is visited once, the overall complexity is $O(n+m)$.