Algorithm Complexity Analysis

Team Computer Man

May 16, 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 time complexity of this Prim's algorithm implementation is O(V^2), $whereVisthenumberofvertice$

- The space complexity of this implementation is O(V), where V is the number of vertices in the graph. This is because we are using a priority queue and a set to keep track of visited vertices, both of which can have a maximum of V elements in the worst case scenario. Additionally, the parent map will also have V entries.

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. e can be the edge
- Thus, the overall complexity is $O(v + e)$.

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 $O(n+m)$.

4 CSV Parsing Function (csv_to_df)

Listing 4: CSV Parsing Function

```
void MainWindow::csv_to_df(string path, QMap<QString, QMap<
    QString, double>> &dataframe)
{
    ifstream csv(path);

    if (!csv) {
        cout << "Could␣not␣open␣file!␣:(" << endl;
    }
    char ch;
    string buffer;
    unsigned quotes = 0, count = 0;
    string row, col, val;

    csv.ignore(1000, '\n');
    while ((ch = csv.get()) != EOF) {
        if (ch == ',' && quotes % 2 == 0) {
            switch (count % 2) {
            case 0:
                row.assign(buffer);
                break;
            case 1:
                col.assign(buffer);
                break;
            }
            count += 1;
            buffer.assign("");
        }

        else if (ch == '\n') {
            val.assign(buffer);
            buffer.assign("");
            // cout << row << " - " << col << " - " << val <<
                endl;
            dataframe[QString::fromStdString(row)][QString::
                fromStdString(col)] = std::stof(val);
        }

        else if (ch == '\"') {
            quotes += 1;
        }

        else {
            buffer += ch;
        }
    }
}
```

Let n be the number of characters stored in the CSV file. Since the function reads through each character of the file individually, deciding where to assign them based on certain delimiter values, the time complexity of the function is $O(n)$.

5 Adding a stadium to the Trip (on_button_addToTrip_clicked)

Listing 5: Adding a College to the Trip

```
void MainWindow::on_button_addToTrip_clicked(bool checked)
{
    QString text = "";
    currentCollege->toggleInTrip(checked);

    if (checked)
        TripColleges.append(*currentCollege);
    else {
        for (int i = 0; i < TripColleges.length(); i++) {
            if (TripColleges[i].name() == currentCollege->name()) {
                TripColleges.remove(i);
                break;
            }
        }
    }

    TripColleges = *find_shortest_path(TripColleges[0].name(), TripColleges.length());

    ui->label_tripColleges->clear();

    int totalDistance = 0;
    for (int i = 0; i < TripColleges.length() - 1; i++) {
        totalDistance += distanceMap[TripColleges[i].name()][TripColleges[i + 1].name()];
        text += TripColleges[i].name() + "␣>␣"
                + QString::number(distanceMap[TripColleges[i].name()][TripColleges[i + 1].name()])
                + "mi␣>␣";
    }

    if (TripColleges.length() != 0)
        text += TripColleges[TripColleges.length() - 1].name();
    ui->label_tripColleges->setText(text);
    ui->label_totalDistance->setText("Total␣Distance:␣" + QString::number(totalDistance));
}
```

Let n be the number of colleges that the function needs to add to the list. This function runs a for loop across the entire list of colleges, inside of which it retrieves values from a map. Because map retrievals have a time complexity of $\log(n)$, the time complexity of the function is $O(n\log(n))$.