Dijkstra’s Algorithm

By Adam Mansour, Noah Shehata, Liam Osman, and Reece Williams

Which Graph Type Is Best For Us?

Adjacency List

- Doesn’t fill up memory with empty spaces in the graph
- Linear lookup times: O(V)
- Easy to add a new vertex: O(1)

Adjacency Matrix

- Doesn’t have to store pointers to the next value of the list in memory
- Constant lookup times: O(1)
- Hard to add a new vertex: O (V^2)

By examining these pros/cons, it is clear that the Adjacency List is the better choice here

Our sparsely connected data set will use less memory, be easy to add new vertex data into quickly, and the lookup times will not be too significantly affected.

Edge Class (Graph.hpp)

```
class Edge {
public:

    //incident vertexes
    Vertex* vertex1;
    Vertex* vertex2;

    std::string neighborValue;
    unsigned long weight;

    //constructor
    Edge(Vertex* v1, Vertex* v2, unsigned long wt) : vertex1(v1), vertex2(v2), weight(wt) {}
};
```

Vertex Class

```cpp
class Vertex {
public:

    std::string label;
    std::list<Edge*> adjList;

    //constructor
    Vertex(std::string lbl) : label(std::move(lbl)) {}
};
```

Graph Class

```
class Graph : public GraphBase {
private:
    std::unordered_map<std::string, Vertex*> vertices;
    std::list<Edge*> edges;

public:
    //constructor
    Graph();
    //destructor
    ~Graph();

    //manipulate graph
    void addVertex(std::string label) override;
    void removeVertex(std::string label) override;
    void addEdge(std::string label1, std::string label2, unsigned long weight) override;
    void removeEdge(std::string label1, std::string label2) override;

    //Dijkstra's Algorithm
    unsigned long shortestPath(std::string startLabel, std::string endLabel, std::vector<std::string> &path) override;
};
```

addVertex Method (Graph.cpp)

```cpp
// Adds a new vertex with the specified label to the graph
void Graph::addVertex(std::string label) {
  // Check if a vertex with the given label already exists
  if (vertices.find(label) == vertices.end()) {
    Vertex *newVertex = new Vertex(label); // Create a new vertex
    vertices[label] = newVertex; // Add the new vertex to the map
  }
}
```

removeVertex Method

```
// Removes a vertex with the specified label from the graph
void Graph::removeVertex(std::string label) {
  auto it = vertices.find(label);
  if (it == vertices.end()) {
    return; // Vertex not found, exit the function
  }

  Vertex *vertexToRemove = it->second;

  // Loop through all vertices to remove edges connected to the vertex
being remove
  for (auto &vertexPair : vertices) {
    Vertex *vertex = vertexPair.second;
    auto &adjList = vertex->adjList;
```

removeVertex Method

```
    // Iterate through the adjacency list and remove edges
    for (auto edgeIt = adjList.begin(); edgeIt != adjList.end();) {
      Edge *edge = *edgeIt;
      if (edge->vertex1 == vertexToRemove || edge->vertex2 ==
vertexToRemove) {
        edgeIt = adjList.erase(edgeIt); // Erase and advance the iterator
        delete edge; // Free the memory of the edge
      } else {
        ++edgeIt; // Move to the next edge
      }
    }
  }

  // Finally, delete the vertex and remove it from the vertices map
  delete vertexToRemove;
  vertices.erase(it);
}
```

addEdge Method

```
// Adds an edge between two specified vertices with a given weight
void Graph::addEdge(std::string label1, std::string label2, unsigned long
weight) {
  auto it1 = vertices.find(label1);
  auto it2 = vertices.find(label2);

  // Check if both vertices exist in the graph
  if (it1 == vertices.end() || it2 == vertices.end()) {
    return; // One or both vertices not found
  }

  // Check for self-loop (edge connecting a vertex to itself)
  if (label1 == label2) {
    return; // Invalid operation
  }
```

addEdge Method

```
  // Check if an edge already exists between the two vertices
  for (auto &edge : edges) {
    if ((edge->vertex1 == vertices[label1] && edge->vertex2 ==
vertices[label2]) ||
        (edge->vertex1 == vertices[label2] && edge->vertex2 ==
vertices[label1])) {
      return; // Edge already exists
    }
  }
```

addEdge Method

```
  Vertex *v1 = it1->second;
  Vertex *v2 = it2->second;

  // Create a new edge and add it to the adjacency lists of both vertices
  Edge *newEdge = new Edge(v1, v2, weight);
  v1->adjList.push_back(newEdge);
  v2->adjList.push_back(newEdge);
}
```

removeEdge Method

```
// Removes an edge between two specified vertices
void Graph::removeEdge(std::string label1, std::string label2) {
  // Check if both vertices exist in the graph
  if (vertices.find(label1) == vertices.end() || vertices.find(label2) ==
vertices.end()) {
    return; // One or both vertices not found
  }
```

removeEdge Method

```
  // Loop through the global edge list to find and remove the edge
  for (auto it = edges.begin(); it != edges.end(); ++it) {
    Edge *edge = *it;
    // Check if this is the edge to remove
    if ((edge->vertex1 == vertices[label1] && edge->vertex2 ==
vertices[label2]) ||
        (edge->vertex1 == vertices[label2] && edge->vertex2 ==
vertices[label1])) {
      edges.erase(it); // Remove edge from global list
      delete edge; // Free memory
      return;
    }
  }
  // If the edge was not found, it means there was no such edge
}
```

shortestPath Method

```
// Calculates the shortest path between two vertices using Dijkstra's
algorithm
unsigned long Graph::shortestPath(std::string startLabel, std::string
endLabel,
                                  std::vector<std::string> &path) {
  // Check if the start vertex exists in the graph
  if (this->vertices.find(startLabel) == this->vertices.end()) {
    throw std::runtime_error("Start vertex does not exist in the graph");
  }
  // Check if the end vertex exists in the graph
  if (this->vertices.find(endLabel) == this->vertices.end()) {
    throw std::runtime_error("End vertex does not exist in the graph");
  }
```

shortestPath Method

```
// Initialize all distances as "infinity" and previous vertices as empty
std::unordered_map<std::string, unsigned long> distances;
std::unordered_map<std::string, std::string> previous;
for (const auto &vertex : this->vertices) {
  distances[vertex.first] = std::numeric_limits<unsigned long>::max();
  previous[vertex.first] = "";
}

// Set the distance to the start vertex as 0
distances[startLabel] = 0;
// Priority queue to hold vertices to be processed
std::set<std::pair<unsigned long, std::string>> vertexQueue;
vertexQueue.insert(std::make_pair(0, startLabel));
```

shortestPath Method

```
  // Process the graph
  while (!vertexQueue.empty()) {
    std::string currentVertex = vertexQueue.begin()->second;
    vertexQueue.erase(vertexQueue.begin());

    // Explore the neighbors of the current vertex
    for (const auto &edge : this->vertices[currentVertex]->adjList) {
      std::string neighbor = (edge->vertex1->label == currentVertex) ?
edge->vertex2->label : edge->vertex1->label;
      unsigned long weight = edge->weight;
```

shortestPath Method

```
        // Calculate the distance through the current vertex
        unsigned long distanceThroughU = distances[currentVertex] + weight;
        if (distanceThroughU < distances[neighbor]) {
          // Update distance and previous vertex if a shorter path is found
          vertexQueue.erase(std::make_pair(distances[neighbor], neighbor));
          distances[neighbor] = distanceThroughU;
          previous[neighbor] = currentVertex;
          vertexQueue.insert(std::make_pair(distances[neighbor], neighbor));
        }
      }
    }

    // Reconstruct the shortest path
    std::vector<std::string> tempPath;
    for (std::string at = endLabel; at != ""; at = previous[at]) {
      tempPath.push_back(at);
      if (previous[at] == "") {
        break; // Start vertex reached
      }
    }
```

shortestPath Method

```
  // Check if a path was found
  if (tempPath.back() != startLabel) {
    throw std::runtime_error("No path exists between '" + startLabel + "'
and '" + endLabel + "'");
  }

  // Reverse the path to get the correct order from start to end
  path.clear();
  for (auto it = tempPath.rbegin(); it != tempPath.rend(); ++it) {
    path.push_back(*it);
  }

  return distances[endLabel]; // Return the shortest path distance
}
```

Difficulties Encountered

- Team Miscoordination
- Disagreements on Design Choices
- Scheduling Conflicts

Conclusion

The trials and tribulations of teamwork!