```
/*======= CSCI203/803 ASSIGNMENT-3 MARKING (out of 10 Marks) =============

========================== YOUR OUTPUT ===================================

Shortest path using Dijkstra alg:
Path: a, l, t
Path distance: 130
Number of vertices visited: 16

Second shortest path using Dijkstra alg:
Path: a, l, q, t
Path distance: 131
Number of vertices visited: 33

Shortest path using A* alg:
Path: a, l, t
Path distance: 130
Number of vertices visited: 9

Second shortest path using A* alg:
Path: a, l, q, t
Path distance: 131
Number of vertices visited: 19

======== MARKING & FEEDBACK ON YOUR OUTPUT (2 marks) ===================================

Your output is correct.

--------- The correct output is shown below ----------

Start and end vertex: a  t

Shortest path using Dijkstra alg:
Path: a l t
Path distance: 130
Number of vertices visited: 16

Second shortest path using Dijkstra alg:
Path: a l q t
Path distance: 131
Number of vertices visited: 33

Shortest path using A* alg:
Path: a l t
Path distance: 130
Number of vertices visited: 9

Second shortest path using A* alg:
Path: a l q t
Path distance: 131
Number of vertices visited: 20

========= MARKING & FEEDBACK ON YOUR REPORT (2 marks) ===================================

// Note: to get full marks the report should list all the algs and data structures
// used in your code and explain any optimisations you did to improve the speed. (1 mark)
// You should also give sensible answers to the questions (1 mark). (see below)

------------------ Example Answers to Questions --------------------------------------
Q1. What if we require that the second shortest path be longer than the shortest path?
Answer: If the second shortest path is required to be shorter than the shortest path, the
proposed solution may not find it. To fulfill this requirement any second shortest path
with the same length should be skipped.

Q2. What if the graph contains cycles?
Answer: If the graph contains cycles the proposed alg may not find the second shortest path
```

```
if it happens to be comprised of a loop in the shortest path. A different alg that test for
loops would be required.

Q3. What if the graph is undirected?
Answer: If the graph is undirected the proposed alg may not find the sceond shortest path
if it happens to be comprised of a backtracked edge in the shortest path.  A different alg
that tests any backtracked edges would be required.
--------------------------------------------------------------------------------------

Your report and your answers to the questions are adiquate.

========= MARKING & FEEDBACK ON YOUR CODE (6 marks) ===================================

// Note: to get full marks the code should be correct and have three optimisations:
// 1. Break loop when end vertex reached.  2. Min-heap used in dijkstra & A* algs.
// 3. Memoization of euclidean distances in the A* alg.

Code looks good

---------------------------------------------------------------------
TOTAL MARKS FOR ASSIGNMENT 3 STEPS 2 to 5:  10 MARKS (OUT OF 10)
---------------------------------------------------------------------

=========================== YOUR CODE =======================================*/
/*
Assignment 3
Ben Malen
bm365

This assignment involves an extension to the single source/single destination
shortest path problem.
*/
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip> // setw
#include <math.h> // sqrt, pow
using namespace std;

const char FILE_NAME[9] = "ass3.txt";
const unsigned int MAX = 50;

/*---------------------------------------------------------------------------*/
// Some helper functions
char indexToLabel(int i) { return (char)i + 'a'; }
unsigned int labelToIndex(char c) { return c - 'a'; }

/*---------------------------------------------------------------------------*/
// Declarations

struct Coord {
    Coord() {}
    int x_;
    int y_;
};

struct WeightedVertex {
    WeightedVertex(int vertex, double weight) : vertex_(vertex), weight_(weight) {}
    int vertex_;
    double weight_;
};
inline bool operator<(const WeightedVertex &lhs, const WeightedVertex &rhs) { return lhs.weight_ < rhs.weight_; }

class WeightedVertexQueue {
```

```cpp
    public:
        WeightedVertexQueue() { nElements_ = 0; }
        bool isFull() { return nElements_ == MAX; }
        bool isEmpty() { return nElements_ == 0; }
        void enqueue(WeightedVertex *elements_);
        unsigned int dequeue();
        void swapVertex(WeightedVertex *&a, WeightedVertex *&b);
        void siftUp(unsigned int i);
        void siftDown(unsigned int i);
        void clear();
    private:
        WeightedVertex * elements_[MAX];
        unsigned int nElements_;
};

struct Path {
    Path(unsigned int start, unsigned int goal, unsigned int nVertices);
    static void printPath(const Path &path, int vertex);
    static void print(const Path &path, unsigned int total = 0);
    int dist_[MAX], parent_[MAX];
    bool selected_[MAX];
    unsigned int nVisited_, start_, goal_;
};

struct Graph {
    Graph(unsigned int nVertices, unsigned int nEdges);
    void addEdge(unsigned int v1, unsigned int v2, unsigned int weight) { matrix_[v1][v2] = weight; }
    void deleteEdge(unsigned int v1, unsigned int v2) { matrix_[v1][v2] = -1; }
    double euclideanDist(unsigned int v);
    void dijkstra(Path &path, bool aStar = false);
    void addCoord(unsigned int vertex, int x, int y) { coords_[vertex].x_ = x; coords_[vertex].y_ = y; };
    unsigned int secondShortest(Path &path, const Path &best, bool aStar = false);
    int matrix_[MAX][MAX];
    Coord coords_[MAX];
    double eDist_[MAX];
    unsigned int nVertices_, nEdges_, start_, goal_;
};

/*---------------------------------------------------------------------------*/
// Driver

int main() {
    ifstream fin;
    fin.open(FILE_NAME);
    if (!fin) {
        cerr << "Could not open " << FILE_NAME << endl;
        return 1;
    }
    // Read in number of vertices and edges
    unsigned int nVertices, nEdges;
    fin >> nVertices >> nEdges;
    if (nVertices > MAX) {
        cerr << "Number of vertices exceeds MAX constant." << endl;
        return 1;
    }
    Graph graph(nVertices, nEdges);
    // Read in coordinates of vertices
    char v;
    int x, y;
    for (unsigned int i = 0; i < nVertices; ++i) {
        fin >> v >> x >> y;
        graph.addCoord(labelToIndex(v), x, y);
    }
    // Read in edges and weights
    char v1Label, v2Label;
```

```
    unsigned int v1, v2, weight;
    for (unsigned int i = 0; i < nEdges; ++i) {
        fin >> v1Label >> v2Label >> weight;
        v1 = labelToIndex(v1Label);
        v2 = labelToIndex(v2Label);
        graph.addEdge(v1, v2, weight);
    }
    char startLabel, goalLabel;
    fin >> startLabel >> goalLabel;
    graph.start_ = labelToIndex(startLabel);
    graph.goal_ = labelToIndex(goalLabel);
    // Run the shortest path algorithms
    unsigned int total;
    cout << "\nShortest path using Dijkstra alg:" << endl;
    Path a1(graph.start_, graph.goal_, nVertices);
    graph.dijkstra(a1);
    Path::print(a1);
    cout << "\nSecond shortest path using Dijkstra alg:" << endl;
    Path a2(graph.start_, graph.goal_, nVertices);
    total = graph.secondShortest(a2, a1);
    Path::print(a2, total);
    cout << "\nShortest path using A* alg:" << endl;
    Path b1(graph.start_, graph.goal_, nVertices);
    graph.dijkstra(b1, true);
    Path::print(b1);
    cout << "\nSecond shortest path using A* alg:" << endl;
    Path b2(graph.start_, graph.goal_, nVertices);
    total = graph.secondShortest(b2, b1, true);
    Path::print(b2, total);
    return 0;
}

/*---------------------------------------------------------------------------*/
// WeightedVertexQueue implementations

// Inserts a new element into the heap.
// The new element is placed at the end of the heap and siftUp moves it up into the correct position.
void WeightedVertexQueue::enqueue(WeightedVertex *vertex) {
    if (isFull()) {
        cerr << "Enqueue overflow: MAX constant exceeded." << endl;
        exit(1);
    }
    elements_[nElements_++] = vertex;
    siftUp(nElements_ - 1);
}

// Removes the top element in the heap
// Returns index of Vertex with lowest weight.
// The top element is replaced with the last element in the heap,
// and siftDown then moves that element back to the bottom of the heap.
unsigned int WeightedVertexQueue::dequeue() {
    if (isEmpty()) {
        cerr << "Cannot dequeue. Queue is empty." << endl;
        exit(1);
    }
    WeightedVertex *pt = elements_[0];
    elements_[0] = elements_[nElements_ - 1];
    --nElements_;
    siftDown(0);
    unsigned int index = pt->vertex_;
    delete pt;
    return index;
}

// Swap the addresses that the pointers are pointing to using reference-to-pointer.
void WeightedVertexQueue::swapVertex(WeightedVertex *&a, WeightedVertex *&b) {
```

```
        WeightedVertex *temp = a;
        a = b;
        b = temp;
    }

    // Min heap
    // Moves element up to its correct position.
    void WeightedVertexQueue::siftUp(unsigned int i) {
        if (i == 0) // then the element is the root
            return;
        unsigned int p = (i - 1) / 2; // integer division to find the parent
        if (*elements_[p] < *elements_[i]) // parent is smaller, so we will leave it as is
            return;
        else {
            swapVertex(elements_[i], elements_[p]); // put smallest in parent
            siftUp(p); // and siftUp parent
        }
    }

    // Min heap
    // Moves element down to its correct position.
    void WeightedVertexQueue::siftDown(unsigned int i) {
        unsigned int left = i * 2 + 1; // index of the left child
        if (left >= nElements_)
            return; // left child does not exist
        unsigned int smallest = left;
        unsigned int right = left + 1; // index of the right child
        if (right < nElements_) // right child exists
            if (*elements_[right] < *elements_[smallest]) // right child is smallest child
                smallest = right;
        if (*elements_[smallest] < *elements_[i]) {
            swapVertex(elements_[i], elements_[smallest]);
            siftDown(smallest);
        }
    }

    // Cleans up memory when we are done with the queue.
    void WeightedVertexQueue::clear() {
        for (unsigned int i = 0; i < nElements_; ++i)
            delete elements_[i];
        nElements_ = 0;
    }

    /*---------------------------------------------------------------------------*/
    // Path implementations

    // Constructor
    Path::Path(unsigned int start, unsigned int goal, unsigned int nVertices) :
    start_(start), goal_(goal) {
        // Initialise arrays
        for (unsigned int i = 0; i < nVertices; ++i) {
            dist_[i] = INT_MAX; // Infinity
            parent_[i] = -1;
            selected_[i] = false;
        }
        nVisited_ = 1;
    }

    // Static function that prints the vertices within a path
    void Path::printPath(const Path &path, int vertex) {
        if (path.parent_[vertex] == -1)
            return;
        printPath(path, path.parent_[vertex]);
        cout << ", " << indexToLabel(vertex);
    }

    // Static function that prints information about the path
```

```cpp
void Path::print(const Path &path, unsigned int total) {
    if (path.dist_[path.goal_] == INT_MAX) {
        cout << "No path found." << endl;
        return;
    }
    cout << "Path: " << indexToLabel(path.start_);
    printPath(path, path.goal_);
    cout << "\nPath distance: " << path.dist_[path.goal_]
         << "\nNumber of vertices visited: " << (total ? total : path.nVisited_) << endl;
}

/*--------------------------------------------------------------------------------*/
// Graph implementations

// Constructor
Graph::Graph(unsigned int nVertices, unsigned int nEdges) :
    nVertices_(nVertices), nEdges_(nEdges) {
    // Initialise all weights and Euclidean distances to -1
    for (unsigned int i = 0; i < nVertices; ++i) {
        for (unsigned int j = 0; j < nVertices; ++j)
            matrix_[i][j] = -1;
        eDist_[i] = -1;
    }
}

// Returns the Euclidean distance from vertex, v, to the goal vertex.
// If the Euclidean distance has been previously calculated, it is fetched from an array.
double Graph::euclideanDist(unsigned int v) {
    if (eDist_[v] == -1)
        eDist_[v] = sqrt(pow(coords_[goal_].x_ - coords_[v].x_, 2) +
        pow(coords_[goal_].y_ - coords_[v].y_, 2));
    return eDist_[v];
}

// Dijkstra's algorithm
void Graph::dijkstra(Path &path, bool aStar) {
    WeightedVertexQueue pq; // Priority queue
    // We can set some values directly from the adjacency matrix (i.e. neighbouring
    vertices of starting vertex).
    for (unsigned int i = 0; i < nVertices_; ++i) {
        if (matrix_[start_][i] != -1) { // There is an edge
            path.dist_[i] = matrix_[start_][i];
            path.parent_[i] = start_;
            pq.enqueue(new WeightedVertex(i, path.dist_[i] + (aStar ? euclideanDist(i)
            : 0)));
        }
    }
    // Set the distance to 0 for the starting vertex and add it to the selected set
    path.dist_[start_] = 0;
    path.selected_[start_] = true;
    while (!pq.isEmpty()) {
        unsigned int u = pq.dequeue(); // Extract vertex with smallest weight
        if (path.selected_[u]) // The shortest path has already been found for this
        vertex
            continue;
        path.selected_[u] = true; // Final shortest distance from starting vertex has
        been determined
        ++path.nVisited_;
        // We can exit as soon as the shortest path to the goal has been found.
        if (path.selected_[path.goal_])
            break;
        // For each neighbour vertex, v, of u
        for (unsigned int v = 0; v < nVertices_; ++v) {
            if (matrix_[u][v] != -1 // if there is a connecting edge
                    && !path.selected_[v] // and the vertex is unvisited
                    && (path.dist_[u] + matrix_[u][v]) < path.dist_[v]) { // and there
                    is a shorter path to v from the starting vertex, through u
```

```
                path.dist_[v] = path.dist_[u] + matrix_[u][v];
                path.parent_[v] = u;
                pq.enqueue(new WeightedVertex(v, path.dist_[v] + (aStar ? euclideanDist(v) : 0)));
            }
        }
    }
    pq.clear(); // Clean up memory
}

// Find the second shortest path
unsigned int Graph::secondShortest(Path &path, const Path &best, bool aStar) {
    int total = 0;
    path.dist_[goal_] = INT_MAX; // Initialise as infinity
    unsigned int current = goal_;
    // Loop through vertices, starting at the goal vertex, until we reach the starting vertex
    while (current != start_) {
        // Store the edge info so we can restore it later
        unsigned int v1 = best.parent_[current], v2 = current, weight = matrix_[v1][v2];
        deleteEdge(v1, v2); // Delete the edge
        Path candidate(start_, goal_, nVertices_);
        dijkstra(candidate, aStar); // Run Dijkstra's algorithm, without the edge
        // Add to the total number of vertices visited
        total += candidate.nVisited_;
        if (candidate.dist_[goal_] != INT_MAX) { // If the goal is reachable
            // Update the 2nd shortest path with the candidate if it is shorter
            if (candidate.dist_[goal_] < path.dist_[goal_])
                path = candidate;
        }
        current = best.parent_[current];
        // Restore the deleted edge
        addEdge(v1, v2, weight);
    }
    return total;
}

/*--------------------------------------------------------------------------------

The graph is read into an array representing an adjacency matrix. Coordinates of
vertices are read into an array of Coord objects. Path objects are used to store
the resulting path information from the shortest path algorithms.

At the start of Dijkstra's algorithm, some values are set directly from the
adjacency matrix (i.e. neighbouring vertices of starting vertex). The algorithm
will stop as soon as the shortest path to the goal has been found. The algorithm
uses a priority queue (min-heap) of pointers to WeightedVertexQueue objects.
Dequeue will always pick the vertex with the minimum weight. Vertices are
weighted by their distance from the starting vertex. When A* is used, the weight
is this distance plus the Euclidean distance between the current vertex and
goal. If the Euclidean distance has been previously calculated, it is fetched
from an array.

The proposed solution from the assignment specification is used to find the
second shortest path.

Notes:
The "second shortest path" algorithm uses the path found by the shortest path
algorithm. The "Number of vertices visited" does not include those counted to
find this path.

What if we require that the second shortest path be longer than the shortest
path?

Since the proposed solution may find an alternate route with the same distance,
Dijkstra's algorithm would need to be modified to find k-shortest paths, then
k+1 shortest paths may be found until the path's distance is greater than the
```

```
shortest. Eppstein's algorithm or Yen's algorithm can also be used to find the
k-shortest paths.

What if the graph contains cycles?

If the graph contains cycles, the second shortest path may contain the shortest
path. If an edge is removed, the algorithm would fail to find this path.
A different approach is needed, such as Eppstein's algorithm. Cycles can be
detected if a depth-first search finds an edge that points back to an ancestor
of the current vertex (back edge). Eppstein's algorithm finds all possible
deviations from the shortest path, from which k-shortest paths are obtained.

What if the graph is undirected?

Similarly, if it's an undirected graph, the second shorted path may contain
the shortest path, and if an edge is removed, the algorithm would fail to find
this path. The same approach mentioned for cycles can be applied.

-------------------------------------------------------------------------------*/
```