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

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

Start and end vertex: a t

Shortest path using Dijkstra alg:
Path: a l t
Path distance: 130.0
Number of vertices visited: 16

Second shortest path using Dijkstra alg:
Path: a l q t
Path distance: 131.0
Number of vertices visited: 33

Shortest path using A* alg:
Path: a l t
Path distance: 130.0
Number of vertices visited: 9

Second shortest path using A*   alg:
Path: a l q t
Path distance: 131.0
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 answers to questions 2 and 3 are inadequate (-0.3 marks)


========= 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.

No min-heap used in dijkstra & A* algs (-0.6 marks)
No memoization or pre-initialisation of euclidean distances in the A* alg (-0.5 marks)

-----------------------------------------------------------------------
TOTAL MARKS FOR ASSIGNMENT 3 STEPS 2 to 5:  8.6 MARKS (OUT OF 10)
-----------------------------------------------------------------------

============================ YOUR CODE =======================================*/
/*
    Assignment 3 - Dijkstra's Algorithm
    Name: Thi Thuy Trang Nguyen
    Student login: tttn941
*/

import java.util.Scanner;
import java.io.IOException;
import java.io.FileInputStream;
import java.util.Arrays;

class Vertex
{
    public char label;
    public int x, y;

    public Vertex(char l, int xO, int yO)
    {
        label = l;
        x = xO;
        y = yO;
    }
}

class Matrix
{
    private final int noOfVertices;
    private int noOfEdges;
    private double matrix[][];
    private Vertex vertices[];

    //for dijkstra alg
    private int noVisited = 0;
    private int shortestPath[];
    private int verticesOnShortestPath = 0;


    public Matrix(int v, int e)
    {
        noOfVertices = v;
```

```
134             noOfEdges = e;
135             matrix = new double[noOfVertices+1][noOfVertices+1];
136             vertices = new Vertex[noOfVertices+1];
137         }
138         public static int labelToInt(char c){ //convert char to int with a=1...z=26
139             return (c - 'a' + 1);
140         }
141         public void addEdgeToList(char src, char dest, double cost) //directed
142         {
143             int srcInt = labelToInt(src);
144             int destInt = labelToInt(dest);
145             matrix[srcInt][destInt] = cost;
146         }
147         public void addVertexToList(char label, int x, int y) //directed
148         {
149             int pos = labelToInt(label);
150             vertices[pos] = new Vertex(label,x,y);
151             //initialise the matrix with infinities
152             for(int i= 1; i <= noOfVertices; i++)
153             {
154                 if(i == pos){
155                     matrix[pos][pos] = 0;
156                 }else{
157                     matrix[pos][i] = Double.POSITIVE_INFINITY;
158                 }
159             }
160         }
161 
162         //Step 1 - print first 5 vertecies
163         public void printFive(){
164             for(int i = 1; i <= 5; i++)
165             {
166                 System.out.print(vertices[i].label + ":\t");
167                 for(int j = 1; j <= noOfVertices; j++)
168                 {
169                     if(matrix[i][j] != Double.POSITIVE_INFINITY && i!=j)
170                     {
171                         System.out.print(vertices[j].label + "(" + matrix[i][j] +
                            ")\t");
172                     }
173                 }
174                 System.out.println();
175             }
176         }
177 
178         //Step 2- Dijkstra
179         public double[] Dijkstra(char src, char dest, int P[])
180         {
181             noVisited = 1;
182             int s = labelToInt(src); //source and destination vertices from file
183             int d = labelToInt(dest);
184             double D[] = new double[noOfVertices+1]; //smallest weights
185             boolean S[] = new boolean[noOfVertices+1]; //selected
186             S[1] = true;
187             for(int i = 2; i <= noOfVertices; i++){
188                 D[i] = matrix[s][i]; //initialise from source
189                 P[i] = s;
190             }
191             int v = minNotSelected(S, D);
192 
193             while(v != 0 && !S[d]){
194                 for(int u = 1; u <= noOfVertices; u++){
195                     if(D[u] > D[v] + matrix[v][u] && S[u] == false){
196                         D[u] = D[v] + matrix[v][u];
197                         P[u] = v;
198                     }
199                 }
200                 v = minNotSelected(S, D);
201             }
202 
203             return D;
204         }
```

```
205
206         public void printPath(char s, char d, int P[])
207         {
208             int src = labelToInt(s); //source and destination vertices
209             int dest = labelToInt(d);
210             int count = 1;
211             int current = dest;
212
213             shortestPath = new int[noOfVertices+1];
214             shortestPath[count] = current;
215             while(current != src && count <= noOfVertices){
216                 count++;
217                 current = P[current];
218                 shortestPath[count] = current;
219             }
220             System.out.print("Path: ");
221             for(int i=count; i >= 1; i--){
222                 System.out.print(vertices[shortestPath[i]].label + " ");
223             }
224             verticesOnShortestPath = count;
225         }
226         public void getPathDistance(double distance)
227         {
228             System.out.println("\nPath distance: " + distance);
229         }
230         public void getNoVerticesVisited()
231         {
232             System.out.println("Number of vertices visited: " + noVisited);
233         }
234         public int minNotSelected(boolean S[], double D[])
235         {
236             int index = 0;
237             double min = Double.POSITIVE_INFINITY;
238             for(int i = 1; i <= noOfVertices; i++){
239                 if(S[i] == false && D[i] < min){
240                     min = D[i];
241                     index = i;
242                 }
243             }
244             noVisited++;
245             S[index] = true;
246             return index;
247         }
248
249         //Step 3
250         public double[] secondShortestPath(char src, char dest, int P[])
251         {
252             double otherPaths[][] = new double[noOfVertices+1][noOfVertices+1];
253             int index = 0, visited = 0;
254             int tempPath[] = P;
255             double shortest = Double.POSITIVE_INFINITY;
256             for(int i=1; i < verticesOnShortestPath; i++){
257                 double temp = matrix[shortestPath[i+1]][shortestPath[i]] ;
258                 matrix[shortestPath[i+1]][shortestPath[i]] = Double.POSITIVE_INFINITY;
259                 otherPaths[i] = Dijkstra(src, dest, P);
260                 matrix[shortestPath[i+1]][shortestPath[i]] = temp;
261                 if(otherPaths[i][labelToInt(dest)] < shortest){
262                     shortest = otherPaths[i][labelToInt(dest)];
263                     index = i;
264                     tempPath = Arrays.copyOf(P, P.length);
265                 }
266                 visited += noVisited;
267             }
268             P = Arrays.copyOf(tempPath, tempPath.length);
269             noVisited = visited;
270             printPath(src,dest,P);
271
272             return otherPaths[index];
273         }
274
275         //Step 4
276         public double calcDistance(Vertex v, Vertex dest)
```

```
277         {
278             return Math.sqrt(Math.pow(v.x - dest.x,2) + Math.pow(v.y - dest.y,2));
279         }
280         public int minNotSelectedAStar(boolean S[], double D[], int d)
281         {
282             int index = 0;
283             double min = Double.POSITIVE_INFINITY;
284             for(int i = 1; i <= noOfVertices; i++){
285                 if(S[i] == false && D[i] + calcDistance(vertices[i], vertices[d]) < min){
286                     min = D[i] + calcDistance(vertices[i], vertices[d]);
287                     index = i;
288                 }
289             }
290             S[index] = true;
291             noVisited++;
292             return index;
293         }
294         public double[] AStar(char src, char dest, int P[])
295         {
296             noVisited = 1;
297             int s = labelToInt(src); //source and destination vertices from file
298             int d = labelToInt(dest);
299             double D[] = new double[noOfVertices+1]; //smallest weights
300             boolean S[] = new boolean[noOfVertices+1]; //selected
301             S[1] = true;
302             for(int i = 2; i <= noOfVertices; i++){
303                 D[i] = matrix[s][i]; //initialise from source
304                 P[i] = s;
305             }
306             int v = minNotSelectedAStar(S, D, d);
307 
308             while(v != 0 && !S[d]){
309                 for(int u = 1; u <= noOfVertices; u++){
310                     if(D[u] > D[v] + matrix[v][u] && S[u] == false){
311                         D[u] = D[v] + matrix[v][u];
312                         P[u] = v;
313                     }
314                 }
315                 v = minNotSelectedAStar(S, D, d);
316             }
317             return D;
318         }
319         public double[] secondShortestPathAStar(char src, char dest, int P[])
320         {
321             double otherPaths[][] = new double[noOfVertices+1][noOfVertices+1];
322             int index = 0, visited = 0;
323             int tempPath[] = P;
324             double shortest = Double.POSITIVE_INFINITY;
325             for(int i=1; i < verticesOnShortestPath; i++){
326                 double temp = matrix[shortestPath[i+1]][shortestPath[i]] ;
327                 matrix[shortestPath[i+1]][shortestPath[i]] = Double.POSITIVE_INFINITY;
328                 otherPaths[i] = AStar(src, dest, P);
329                 matrix[shortestPath[i+1]][shortestPath[i]] = temp;
330                 if(otherPaths[i][labelToInt(dest)] < shortest){
331                     shortest = otherPaths[i][labelToInt(dest)];
332                     index = i;
333                     tempPath = Arrays.copyOf(P, P.length);
334                 }
335                 visited += noVisited;
336             }
337             P = Arrays.copyOf(tempPath, tempPath.length);
338             noVisited = visited;
339             printPath(src,dest,P);
340 
341             return otherPaths[index];
342         }
343     }
344 
345     public class ass3
346     {
347         public static void main(String[] args) throws CloneNotSupportedException
348         {
```

```
349
350             Scanner sc = new Scanner(System.in);
351
352             try{
353                 Scanner input = new Scanner(new FileInputStream("ass3.txt"));
354                 final int noOfVertices = input.nextInt();
355                 final int noOfEdges = input.nextInt();
356                 Matrix matrix = new Matrix(noOfVertices, noOfEdges);
357                 for(int i = 1; i <= noOfVertices; i++)
358                 {
359                     char src = input.next().charAt(0);
360                     int x = input.nextInt();
361                     int y = input.nextInt();
362                     matrix.addVertexToList(src, x, y);
363                 }
364                 for(int i = 1; i <= noOfEdges; i++)
365                 {
366                     char src = input.next().charAt(0);
367                     char dest = input.next().charAt(0);
368                     double cost = input.nextInt();
369                     matrix.addEdgeToList(src, dest, cost);
370                 }
371                 char src = input.next().charAt(0);
372                 char dest = input.next().charAt(0);
373                 input.close();  //close file
374
375                 //matrix.printFive(); //step 1
376                 int P[] = new int[noOfVertices+1];
377                 System.out.println("Start and end vertex: " + src + " " + dest);
378
379                 //Step 2
380                 System.out.println("\nShortest path using Dijkstra alg:");
381                 double D1[] = matrix.Dijkstra(src,dest,P);
382                 matrix.printPath(src, dest, P);
383                 matrix.getPathDistance(D1[matrix.labelToInt(dest)]);
384                 matrix.getNoVerticesVisited();
385
386                 //Step 3
387                 System.out.println("\nSecond shortest path using Dijkstra alg:");
388                 double D2[] = matrix.secondShortestPath(src,dest,P);
389                 matrix.getPathDistance(D2[matrix.labelToInt(dest)]);
390                 matrix.getNoVerticesVisited();
391
392                 //Step 4
393                 System.out.println("\nShortest path using A* alg:");
394                 double D3[] = matrix.AStar(src,dest,P);
395                 matrix.printPath(src, dest, P);
396                 matrix.getPathDistance(D3[matrix.labelToInt(dest)]);
397                 matrix.getNoVerticesVisited();
398
399                 System.out.println("\nSecond shortest path using A* alg:");
400                 double D4[] = matrix.secondShortestPathAStar(src,dest,P);
401                 matrix.getPathDistance(D4[matrix.labelToInt(dest)]);
402                 matrix.getNoVerticesVisited();
403
404             }catch(IOException e){
405                 System.err.println("File fails to open. Terminating...");
406                 System.exit(1);
407             }
408         }
409     }
410
411     /* Step 5
412     -To represent the graph, I used an adjacency matrix implemented using a
        direct-access 2 dimentional array.
413     -Instead of using the ASCII code of the vertex's lebel as an index to directly
        access the matrix, which is huge,
414      I used a function that change the value of the label to a smaller integer from 1 to
        26, with a=1 and z=26.
415     -I used a boolean array to record the visited nodes. If node i is visited, then S[i]
        = true
416     -A double array was used to record the distance from the start vertex to every other
```

```
    vertices in the graph.
417  If there's no path from the start vertex to vertex i, then D[i] = infinity
418
419 I used the proposed solution to the second shortest path problem:
420     find the shortest path and store it in an array
421     minCost = infinity
422     for each edge ei on the shortest path
423         store the cost of edge ei in the graph
424         remove the edge by set the cost of edge ei in the graph to infinity
425         find the shortest path from the start to the goal without edge ei using
            Dijkstra's algorithm
426         if(minCost > the cost of the shortest path without edge ei)
427             minCost = currentCost
428             record the number of visited nodes and the path
429         end if
430     end for
431
432 The algorithm goes through all but one edge. If the second shortest path must be
    longer than the shortest patha and there's another path/edge with the same cost, the
    algorithm will fail.
433 We can repeat the same algorithm and remove each edge in the path that we have just
    found together with each edge of the shortest path,
434  compare the cost of the new second shortest path to the paths found before and
     return the one with the lowest cost.
435  We will need to keep track of the paths we visited, which can get very huge. The
     worst case is when all paths have the same cost.
436
437 Since the algorithm uses Dijkstra's algorithm, it will fail if the graph has cycles.
    But we can replace the Dijktra's algorithm with Belman-Ford algorithm to find the
    second shortest path.
438
439 The algorithm will still work with an undirected graph. Since an undirected graph is
    simply just a normal graph with bidirectional connections and the algorithm relies
    on Dijkstra's performance,
440 it still can work with an undirected graph.
441 */
442
```