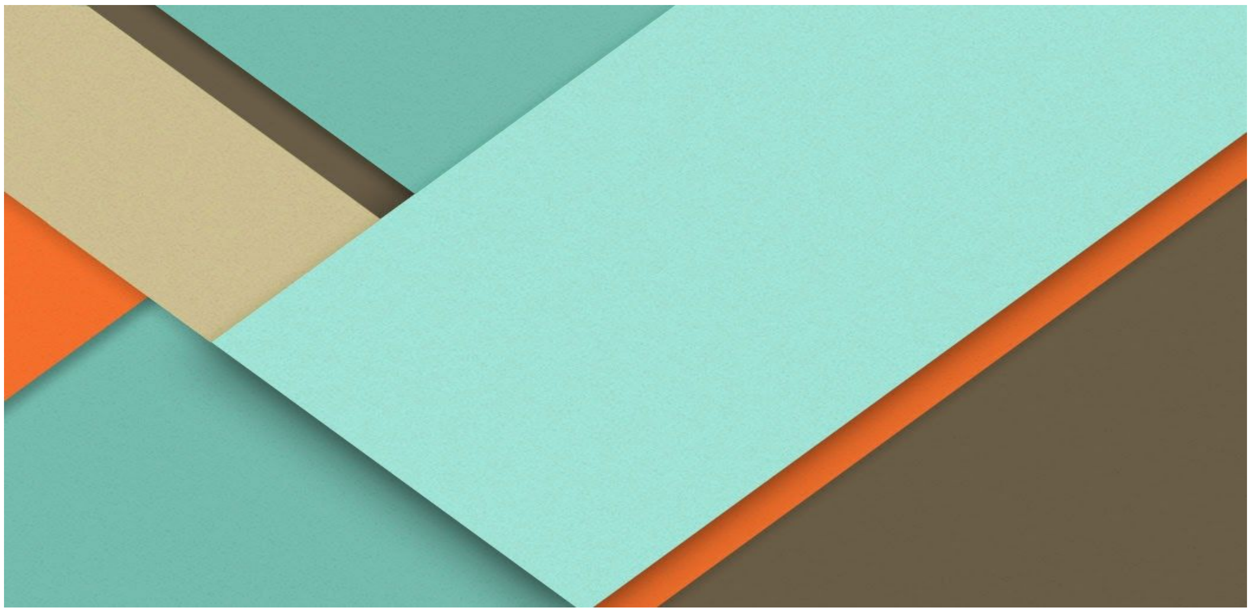


PROJECT REPORT

ON

The Minimum Latency Problem

SPRING 2020

Submitted by:

Krishna Teja

180001026

Jeevan Reddy

180001039

Submitted to:

Dr.Kapil Ahuja

Dept. of CSE

IIT Indore

Index

1.Introduction

We are given a set of points p_l, . . . ,p_n and a symmetric distance matrix (d_{ij}) giving the distance between p_i and pj. We wish to construct a tour that minimizes summation of l(i), i=1 to i=n, where l(i) is the latency of p_i, defined to be the distance traveled before first visiting p_i. The total latency of the tour is the sum of the latencies of all the points. We wish to find the tour which minimizes the total latency.

The problem is a variation of the well-known traveling salesman problem (TSP), is also known as delivery man problem, cumulative TSP and school bus driver problem, traveling repairman problem in which a repairman is supposed to visit the nodes of a network in a way to minimize the overall waiting times of the customers located in the nodes.

Applications

The problem has many real-life applications - one of them is a home delivery of pizzas. In this application, several delivery orders are clustered, and someone hopes to minimize the delivery time. Other applications include in the computer networks where someone wants to find information stored somewhere in the network of computers, single machine scheduling problem with sequence-dependent processing times that aims to minimize the total flow time of the tasks.

2. Difference between MLP and TSP

At first glance the Minimum Latency problem seems to be a simple variant of the travelling salesperson problem. However, closer examination reveals a variety of Aspects in which this problem is very different from Traveling salesman problem.

Small changes in the structure of a metric space can cause highly nonlocal changes in the structure of the MLT . As a result, a simple paradigm which applies to most optimization problems on graphs — namely, “decompose the graph into biconnected components; solve the optimization problem on the various (hopefully) small components and then put them together to obtain a solution for the big problem” — does not work here. The absence of this paradigm rules out simple algorithms even in the case where the underlying graph is a tree (i.e, the metric space is simply the metric closure of a weighted tree.).

Another prominent difference between the MLT and the optimal TSP tour is that the MLT may revisit points an unbounded number of times even when the underlying graph has a bounded degree. Consider the case where the metric space is the real line and the point p_i is located at x=(—3)$^{\wedge}$i . In this case the MLT starting at the origin will visit the points in order (i.e., the ith point to be visited is p_i). Thus the MLT crosses the origin (n – 1) times.

Both the MLP problem and the TSP problems are special cases of a more general problem, the “time-dependent traveling salesman problem”. Assume a graph which is a straight line, Here the distance function on the metric space varies with “time”, i.e., if the ith edge to be traversed has weight e then it incurs a cost c(e, i). The goal is to minimize the total cost of visiting

all vertices. The TSP is the special case where the cost function is just c(e, i) = e (independent of i) and the MLT is the case where the cost function is given by c(e, i) = (n – i)×e.
Situations where the cost function c(e, i) is some convex combination of these two seem to arise naturally too.

For instance, consider the following vehicle routing problem: a delivery truck has to deliver N items to N points in a metric space, and then return to its starting point. If the truck travels a distance D with k items, then the cost of that leg is (k + u)D (the cost is proportional to the load on the truck which is k units for the items + u units for the weight of the truck).

3. Illustration

A brief illustration about the differences between the Traveling Salesman Problem and the Minimum Latency Problem.
a.) For a graph G with six nodes, each edge with a cost b_{ij} we will solve the traveling salesman problem and the Minimum Latency problem.

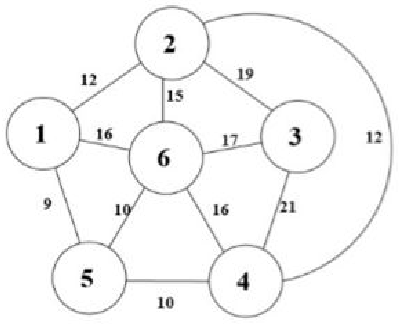


Figure 1: Example.

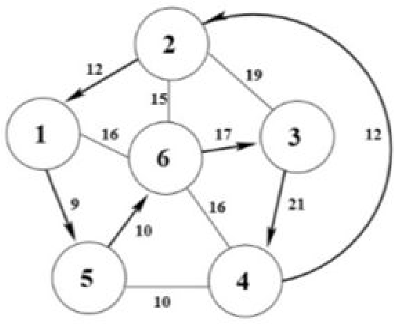


Figure 2: TSP Solution.

The figure (2) shows the TSP solution for this graph. The integer optimal solution for this TSP is 81 (9 + 10 + 17 + 21 + 12 + 12).

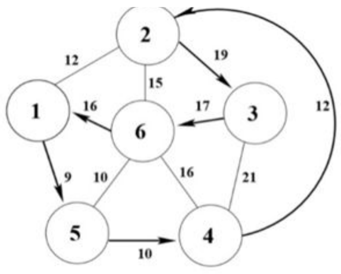


Figure 3: MLP Solution

The figure (3) shows the MLP solution for this graph. The integer optimal solution for this MLP is 259 (9 +(9 + 10)+ (9 + 10 + 12)+ (9 + 10 + 12 + 19)+ (9 + 10 + 12 + 19 + 17) + (9 + 10 + 12 + 19 + 17 + 16)) or 259 (9 ∗ 6) + (10 ∗ 5) + (12 ∗ 4) + (19 ∗ 3) + (17 ∗ 2) + (16 ∗ 1).

b.) For the weighted graph G with six nodes, represented by time matrix T in Table 1,

Table 1: Time matrix T

	1	2	3	4	5	6
1	9999	12	39	42	9	16
2	12	9999	19	12	32	15
3	39	19	9999	21	45	17
4	42	12	21	9999	10	16
5	9	32	45	10	9999	10
6	16	15	17	16	10	9999

The optimal solution for the TSP is 81=(9+10+17+21+12+12) with optimal tour {1→5→6→3→4→2→1}, whereas the MLP solution is 259 = (9 + (9+10) + (9+10+12) +(9+10+12+19) + (9+10+12+19+17) + (9+10+12+19+17+16)) = (9*6 + 10*5 + 12*4 + 19*3 + 17*2 + 16*1) with optimal tour {1→5→4→2→3→6→1}.

4.Algorithm:

4.1. Brute Force Algorithm:

The most basic approach is the brute force approach where you get to check all the permutations of paths from the source. This agorithm has a worst case time complexity of $\Omega((n-1)!)$.

Reason: We start from origin. we could select any one of the remaining (n-1) vertices (in the worst case if all (n-1) vertices are connected with origin) in (n-1) ways, then we are left with (n-2) vertices. Out of these (n-2) vertices we could select any one of the vertices in (n-2) ways, and continued....

4.2. Dynamic Programming:

Preliminaries:

Here we assume that graph is a simple and connected graph with a nonnegative weight on each edge. A tour is a route from the origin and visiting each vertex at least once. A subtour is a partial or a complete tour starting at the origin. Let H be a subgraph or a subtour. The set of vertices of H is denoted by V(H). For u, v $\in$ V(G), we use dG(u, v) to denote the length of the shortest path between u and v on G. For a subtour P, dP(u, v) denotes the distance from the first visit of u to the first visit of v in P, and w(P) denotes the length of P.

Definition 1:

Let P be a subtour starting at “s” on graph G. For a demand vertex v visited by P, the latency of v is defined as dP(s, v), which is the distance from the origin to the first visit of v on P. The latency of a tour

P is defined by $L(P) = \sum_{v \in U} dP(s, v)$, in which U is the demand vertex set.

In general, the input graph of a MLP may be any simple connected graph with nonnegative edge weights and the demand vertex set does not necessarily include all the vertices.

A metric graph is a complete graph with edge weights satisfying the triangle inequality. By a simple reduction, we may assume that the input graph is always a metric graph and all the vertices are the demand vertices.

Computing Metric Closure:

Let G = (V, E, w) be the underlying graph and $U \subset V$ be the demand vertex set. We first compute the metric closure $G^{-} = (U, U \times U, w^{-})$ of G, in which the weight on each edge is the shortest path length of the two endpoints in G. For any tour P^{-} on G^{-}, we can construct a corresponding tour P on G by simply replacing each edge in P^{-} with the corresponding shortest path on G. It is easy to see that $L(P) \leq L(P^{-})$.

Conversely, given any tour P on G, we can obtain a tour P^{-} on G^{-} by eliminating all vertices not in U. Since the edge weight is the shortest path length, we have $L(P^{-}) \leq L(P)$. Consequently the minimum latencies of the two graphs are the same..

Furthermore, if there exists an O(T(n)) time exact or approximation algorithm for the MLP on metric graphs, the MLP on general graphs can be solved in O(T(n) +f(n)) time with the same performance guarantee, in which f(n) is the time complexity for computing the all-pairs shortest path length. In the remaining paragraphs, we assume that the input graph G is a metric graph and each vertex is a demand vertex. It should also be noted that the optimal tour never visits the same vertex twice in a metric graph.

Definition 2:

Let P be a subtour on graph G. Define a cost function c(P) = L(P) + (n − |V (P)|)w(P), i.e., c(P) is the total latency of the visited vertices plus the length of P multiplied by the number of vertices not visited. Let P1 and P0 are two routes such that the last vertex of P1 is the first vertex of P0. We use P1//P0 to denote the route obtained by concatenating P1 and P0. For a subtour P, we say that P has configuration (R, v), in which R = V (P) and v is the last vertex of P. The dynamic programming algorithm is based on the following property which can be easily shown by definition. It also explains the reason why we define the cost function c in such a way.

Claim 1:

Let P1 and P2 be subtours with the same configuration and c(P1) ≤ c(P2). If Y2 = P2//P0 is a complete tour, i.e., P0 is a route starting at the last vertex of P2 and visiting all the remaining vertices, then Y1 = P1//P0 is also a tour and L(Y1) ≤ L(Y2).

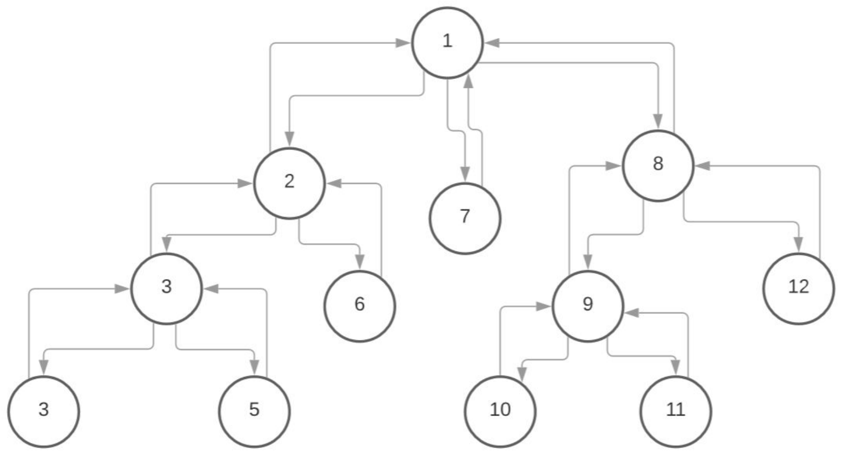
To find the minimum latency, by Claim 1, we only need to keep one subtour for each possible configuration. The dynamic programming algorithm starts at the subtour containing only the origin and computes the best subtour for each configuration in the order that the number of the visited vertex is from small to large. The time complexity then follows that there are O(n*2^n) configurations and we generate O(n) subtours when a subtour is extended by one vertex.

5. Exact Solutions:

5.1. Unweighted trees and depth-first search:

Consider the case when the points are vertices of a tree all of whose edges have unit length.In this case a tour is optimal if and only if it is a depth-first search.

For a vertex v , let depth(v) denote its distance in the tree from the starting point. If we fix a particular depth-first search for a given graph, and let v be the ith distinct vertex to be visited. Then v has latency 2i – depth(v).



Note:

Latency of node 12 = 2×11 - 2 (since node 12 is the 11th distinct vertex to be visited and depth of node 12 i.e., distance from node 1 is 2 .)

=22-2

=20.

The path used to reach node 12 is 1 -> 2 -> 3 -> 4 -> 3 -> 5 -> 3 -> 2 -> 6 -> 2 -> 1 -> 7 -> 1 -> 8 -> 9 -> 10 -> 9 -> 11 -> 9 -> 8 -> 12. (latency is 20 because 20 edges are involved in this path).

Total latency is the sum of latencies of all the vertices.

So, Total latency = Latency of vertex 1 + Latency of vertex 2 + Latency of vertex 3 +.....+ Latency of vertex 11 + latency of vertex 12.

Total latency = 0 + 1 + 2 + 3 + 5 + 8 + 11 + 13 + 14 + 15 + 17 + 20

= 109

- Code for this is presented at the end of the document.

5.2. Travelling repairman problem in one-dimension:

Formally, an instance of the TRP can be described as follows:

Given a distance matrix t [ij] between n locations, and a distinguished starting locations, find a permutation π (0)=s, π (1), . . ., π (n— 1) such that the following cost function is minimized:

$$c = \sum_{i=1}^{n-1} \sum_{j=1}^{i} t[\pi(j-1), \pi(j)] = \sum_{i=1}^{n-1} (n-i)\, t[\pi(i-1), \pi(i)].$$

Since this problem is so closely related with the travelling salesman problem, it is not surprising that the general version of the TRP is NP-complete. Unlike the TSP, however, it is non-trivial to solve the one-dimensional further restriction of the problem, that is, the case in which the locations are ail on a straight line. This is called the line-TRP.

5.2.1. THE LINE-TRP:

The instance of the line-TRP is best described as in figure 2, where s = $x_o = y_o$ is the starting location, $x_1, \ldots, x_m$ (resp. $y_1, \ldots, y_n$) are the machines to the left (resp. right) of the origin. Let us start with an elementary, although crucial, observation.

Lemma 1:

If in the optimum route the repairman has visited machine x_u, then it has also visited machine x_v , $v<u$. Similarly for the machines to the right of s.

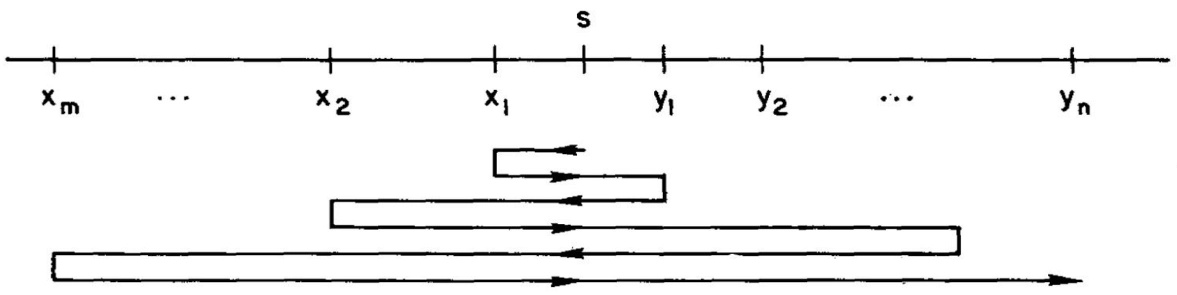


Figure 2

Proof: Trivial

In other words, it is suboptimal for the repairman to "pass by" a machine without repairing it. Therefore, the optimum route looks like the one shown in figure 2.

Let us represent by $[x_i , y_j]$ the fact that the repairman is currently at x_i, which is also the leftmost location visited, while y_j is the rightmost location the repairman has visited. Similarly for $[y_j , x_i]$ (repairman is currently at y_j, which is also the rightmost location visited, while x_i is the leftmost location the repairman has visited). By lemma 1, all locations between x_i and y_j are already visited.Thus, each such pair of locations represents a complete state of the route. The initial state is $[x_o , y_o]$, and the final is one of $[x_m , y_n]$,

$[y_n, x_m]$. Finally, it follows from the lemma that states $[x_i, y_j]$ can be reached in an optimal route only from states $[x_{i-1}, y_j]$ and $[y_j, x_{i-1}]$ (since they are the last two preceding states) .

We let c $[x_i, y_j]$ denote the minimum possible total delay accumulated by all locations during the time it took to visit all locations between x_i and y_j ending up in x_i. The above observations immediately lead to the following equations for Computing this cost:

$$c[x_o, y_o] = c[y_o, x_o] = 0,$$

$$c[x_i, y_j] = \min \{ c[x_{i-1}, y_j] + (m+n+1-i-j)t[x_{i-1}, x_i], \quad c[y_j, x_{i-1}] + (m+n+1-i-j)t[y_j, x_i] \}, \qquad (2.1)$$

$$c[y_i, x_j] = \min \{ c[y_{i-1}, x\, j] + (m+n+1-i-j)t[y_{i-1}, y_i], \quad c[x_j, y_{i-1}] + (m+n+1-i-j)t[x_j, y_i] \},$$

we let $c[x_i, y_{-1}] = c[y_j, x_{-1}] = \infty$.

To justify (2.1), the second and third equations state that the minimum total delay in order to reach a particular state is the minimum total delay for reaching one of the two preceding states ($[x_{i-1}, y_j]$ and $[y_j, x_{i-1}]$), plus the distance of the last locations in the two states multiplied by the number of unvisited locations at the previous state.

Finally, the minimum cost is

$$(2.2) \qquad C = \min \{ c[x_m, y_n], c[y_n, x_m] \}.$$

Complexity analysis:

Theorem 1: The line-TRP can be solved in O (mn) time.

Proof: We can compute all c[x, y] using equations (2.1) in constant time per value of c [x, y] computed, and thus in O (mn) time in total (since we have m points on left of s and n points on right of s, we have total of m+n points and m×n combinations of points such that one point is on the left of s and the other is on right of s) . In this estimate, our units of computation are arithmetic operations on integers of precision comparable to that of the largest integer in the cost matrix t.

In equation (2.1) , at each step we have two different candidates to choose minimum among them. So, at each computation we record which of the candidate values was the optimum, and we can reconstruct the optimum path.

5.2.2. THE LINE-TRP WITH DEADLINES :

Suppose now that each machine t has a deadline d_i and we impose the additional constraint that each machine must be repaired before its deadline expires. We show below that this problem is NP-complete. We redefine the problem for this purpose as usual, by also introducing another bound M, and asking whether there is a route such that all deadlines are met, while the sum of delays is at most M.

THEOREM 2: The line-TRP with deadlines is NP-complete.

Proof:

Theorem: If X is NP-complete , Y is NP and X is polynomial time reducible to Y then we can say that Y is NP-complete.

From above theorem first we show that Y is NP, then choose X such that its NP-complete and polynomial time is reducible to Y.

From THEOREM 1 stated above line-TRP can be solved in polynomial time. Here we have line-TRP with deadlines. So at each computation we need to check if the deadline condition is met or not (which would take additional constant time). For a given instance of a solution we can apply a checking algorithm which runs in polynomial time(refer Theorem 3). Hence, line-TRP with deadlines is NP.

We now choose a proper X such that it is NP-complete. Let us choose 0-1 KNAPSACK problem as our X. (KNAPSACK problem is NP-complete according to Karp's 21 NP-complete problem. KNAPSACK was shown to be NP-complete by reducing Exact Cover to KNAPSACK).So, We show that 0-1 KNAPSACK is polynomial time reducible to line-TRP with deadlines.

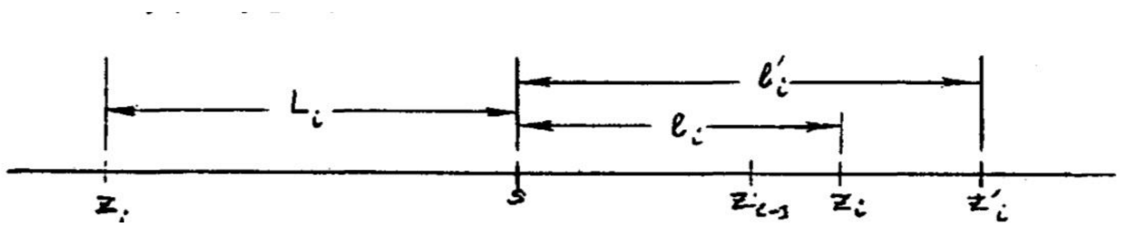
In this problem we are given a set [I] of n objects. For each i ∈ I we are given its size s_i , and its value v_i , both integers. Finally, we are given two positive integers B and K We are asked if there exist x_1 . . . , x_n belongs to { 0 , 1} such that:

$$\sum_{i=1}^{n} v_i x_i \geqq K, \tag{3.1}$$

$$\sum_{i=1}^{n} s_i x_i \leqq B. \tag{3.2}$$

In fact, we shall need a slightly modified version of the 0-1 KNAPSACK in which no two v_i are equal. It is trivial to prove that this version of the problem still remains NP-complete. We can also assume, without loss of generality, that $v_i<v_{i+1}$ for all i.

Given an instance of 0-1 KNAPSACK, we construct an instance of the line-TRP with deadlines as follows: For each element in I belongs to l we add to our instance three locations. Two of these, z_i and z_i', are to the right of s at a distance of l_i and l_i' from s, respectively, and the third, Z_i, to the left at a distance of L_i (see fig. 4).



Also, we add a point z_{n+1} to the right of s, at a distance l_{n+1}. All these lengths are defined below in terms of the parameters of the instance of 0-1 Knapsack:

(3.3) $$a_i = \frac{1}{2} s_i,$$

(3.4) $$a = \sum_{i=1}^{n} s_i,$$

(3.5) $$l_1 = a,$$

(3.6) $$l'_i = l_i + a_i,$$

(3.7) $$l_{i+1} = l'_i + a,$$

(3.8) $$L_i = Gv_i + 2a_i + 3(n-i)a_i - l_i.$$

Here G is a very large constant, whose role will be clear later. As for the deadlines:

(3.9) $$d_i = a_i,$$

(3.10) $$d'_i = d_i + a_i + 2l_i + 2L_i,$$

(3.11) $$D_i = d_i + 2a_i + l_i + L_i,$$

(3.12) $$d_{i+1} = d_i + 2a_i + l_i + 2L_i + l_{i+1},$$

(3.13) $$d_{n+1} = B + l_{n+1} + 2\sum_{i=1}^{n} (l_i + L_i),$$

and finally:

(3.14) $$M = N - 2GK,$$

where N stands for:

(3.15) $$N = 2\sum_{i=1}^{n} (n-i+1)(l_i + L_i) + 2a + 8\sum_{i=1}^{n} l_i + 5\sum_{i=1}^{n} L_i + l_{n+1}.$$

Claim: There exists a legal route with cost $C \leq M$ which achieves all deadlines if and only if there is a solution to the given instance of 0-1 KNAPSACK.

Part 1 of the claim($\leftarrow$, backward):

If there is a solution to the given instance of 0-1 KNAPSACK, then there exists a legal route with cost $C \leq M$ which achieves all deadlines.

Suppose that $x_1 \ldots, x_n$ is the solution to the knapsack. We construct a route as follows: We first visit location z_1 then, if $x_1 = 1$ we repair machine z_1', change direction, visit Z_1 , change again, and visit z_2 . On the other hand, if $x_1 = 0$, after z_1 we go to Z_1 first, and then to z_1' and z_2 . In general, we proceed in the same way, always changing direction at the locations on the left, while we change direction at the machine z_i (resp. z_i') if $x_i = 0$ (resp. $x_i =1$). Thus the route ends at z_{n+1} .

We now have to prove two things:

(a) That the route is legal, in that it makes all the deadlines.

(b) That the corresponding cost is less than or equal to M.

To prove (a), we have to show that the deadline is not violated at any point. For machines z_i , z_i' and Z_i , $i = 1, \ldots, n$, this will be proved by induction on i. Denote by t_i , t_i' and T_i the time the repairman arrives at machine z_i , z_i' and Z_i respectively.

For the basis step, it is obvious that $t_1 \leq d_1$.

Assume $t_i \leq d_i$.

Since z_i' will be visited first iff x_i=1:

$$t_i' \leq d_i + a_i + (2l_i+2L_i)(1-x_i)$$

$$=(d_i + a_i + (2l_i+2L_i)) - x_i(2l_i+2L_i)$$

$$=d_i' - x_i(2l_i+2L_i) \qquad \text{(From (3.10))}$$

$$\leq d_i'$$

For Z_i : $T_i = t_i + 2a_ix_i + l_i + L_i$

But $t_i \leq d_i$ and $x_i \leq 1$

So, $T_i = t_i + 2a_ix_i + l_i + L_i \leq d_i + 2a_i + l_i + L_i$

By (3.11), $T_i \leq D_i$

For z_{i+1} : $t_{i+1} = T_i + L_i + l_{i+1}$

But $T_i \leq D_i$

So, $t_{i+1} = T_i + L_i + l_{i+1} \leq D_i + L_i + l_{i+1}$

By (3.11) and (3.12), $t_{i+1} \leq d_{i+1}$

For machine Z_{n+1} , time t_{n+1} can be recursively computed to:

(3.16) $$t_{n+1} = 2\sum_{i=1}^{n} (l_i + L_i) + l_{n+1} + 2 \sum_{i=1}^{n} a_ix_i$$

By (3.3), (3.2) and (3.13) we obtain,

$$t_{n+1} \leq d_{n+1}$$

To prove (b), we first compute the cost C.

Notice that z_i is reached at time

$$t_i = l_i + 2\sum_{j=1}^{i-1} (l_i + L_i + a_j x_j) ,$$

z_i' is reached at time $t_i' = t_i + a_i + 2(l_i + L_i)(1-x_i)$ and

$$Z_i \text{ at } T_i = t_i + l_i + L_i + 2a_i x_i$$

Total contribution of i^{th} round of visits to cost is,

$$c_i = t_i + t_i' + T_i$$

$$c_i = a_i + 3t_i + 3(l_i+L_i) - 2(Gv_i + 3(n-i+1)a_i)x_i$$

In adding all c_i's, $a_i x_i$ term gets cancelled and thus:

(3.17) $$C = N - 2G \sum_{i=1}^{n} v_i x_i$$

Now, by (3.1) and (3.14),

$$C \leq N\text{-}2GK = M$$

Hence Part 1 of the claim is proved.

Part 2 of the claim(→, forward):

If there exists a legal route with cost $C \leq M$ which achieves all deadlines, then there is a solution to the given instance of 0-1 KNAPSACK. Suppose now that we have a legal route with cost $C \leq M$. First, we claim that the route has the form described above, that is , after z_i , we visit either z_i' , Z_i , z_{i+1} in this order, or Z_i , z_i' , z_{i+1} , in this order. Suppose not.

Take the first time this form is violated. This can be done in four different ways:

(a) $\quad z_i \to z_i' \to z_{i+1} \to Z_i;$

(b) $\quad z_i \to z_i' \to Z_{i+1} \to z_{i+1};$

(c) $\quad z_i \to Z_i \to Z_{i+1} \to z_i';$

(d) $\quad z_i \to Z_i \to z_i' \to Z_{i+1}.$

Each one of the preceding four cases leads to a contradiction, because the deadlines are defined in such a way, that they forbid exactly these kinds of routes. We shall prove only case (a) as an example:

(a) $\mathbf{z_i \to z_i' \to z_{i+1} \to Z_i}$

We have, $D_i = d_i + 2a_i + l_i + L_i$

Now, to reach Z_i from source,

Total time taken $= t_i + a_i + (l_{i+1} - l_i') + l_{i+1} + L_i$

$T_i = t_i + a_i + 2l_{i+1} - (l_i + a_i) + L_i$ (By (3.6))

$T_i = t_i + 2(l_i+a_i+a) - l_i + L_i$ (By (3.6), (3.7))

$T_i = t_i + 2a + (l_i+2a_i+L_i)$

We have, $a>a_i$

Because, $a = \sum_{i=1}^{n} (s_i) = \sum_{j=1 \text{ and } j \neq i}^{n} (s_j) + s_i$

$$= \sum_{j=1 \text{ and } j \neq i}^{n} (s_j) + 2a_i$$

$$a - 2a_i = \sum_{j=1 \text{ and } j \neq i}^{n} (s_j) > 0 \Rightarrow a > 2a_i$$

$$\Rightarrow a > a_i$$

$$\therefore \quad t_i + 2a > d_i = a_i$$

$$t_i + 2a + (l_i + 2a_i + L_i) > d_i + (l_i + 2a_i + L_i)$$

$$\therefore \quad T_i > D_i$$

Thus, the deadline is violated.

Once we have established this "normal form" of the route, we can construct a solution of the 0-1 KNAPSACK by taking $x_i = 1$ whenever the route goes from z_i to z_i' and $x_i = 0$ otherwise. It remains to be proved that such an assignment satisfies (3.1) and (3.2).

Since the route is a legal one, $t_{n+1} \leq d_{n+1}$. However, by (3.16) and

(3.13) we conclude that:

$$\sum_{i=1}^{n} s_i x_i \leq B.$$

Finally, we know that C ≤ M, and by (3.17) and (3.14), we immediately Obtain:

$$\sum_{i=1}^{n} v_i x_i \geqq K.$$

Thus the x_i's constructed constitute a solution to the knapsack problem.

Hence Part 2 of the claim is proved.

Now if we wish to determine whether the given deadlines can be satisfied at all, no matter with how much total delay (above we restricted the total delay or cost to be ≤ M).

THEOREM 3: Given an instance of the line-TRP with deadlines, we can tell whether there is some route that satisfies all the deadlines in O(mn) time.

Sketch: We can do this by a dynamic programming recurrence, as in theorem 1. One key observation is that a lemma analogous to lemma 1 holds here as well, in that it is suboptimal for the repairman to "pass by" machines, and thus pairs [x, y] are adequate states. The recurrence now computes, for each pair [x, y] as in theorem 1, the shortest time within which the repairman can visit all locations between x and y, ending up at y, meeting all deadlines so far. If no such route exists, this value is ∞. Our task is over once this time for [x_m , y_n] or [y_n , x_m] is finite.

5.2.3. A PSEUDO-POLYNOMIAL ALGORITHM :

We now attack the question of finding the optimum route, satisfying the deadlines (despite the fact that we proved it NP-complete above). Let us denote by $[x_i , y_j , t]$ the state at which the repairman at time t is currently at machine x_i , and has travelled in the opposite direction till machine y_j . Now we assign to each state the cost:

$$c[x_i, y_j , t] = \infty \quad \text{if } t > dx_i,$$

$$= \min \{c[x_{i-1} , y_j , t - t[x_{i-1} , x_i]] + (m+n+1-i-j)t[x_{i-1} , x_i], \; c[y_j , x_{i-1}, t - t[y_j , x_i]] + (m+n+1-i-j)t[y_j , x_i]\} \quad \text{otherwise}$$

The optimum total cost is given by:

$C = \min\{c[x_m , y_n , t], c[y_n , x_m , t]:t = 0, 1, . . ., D\}$,

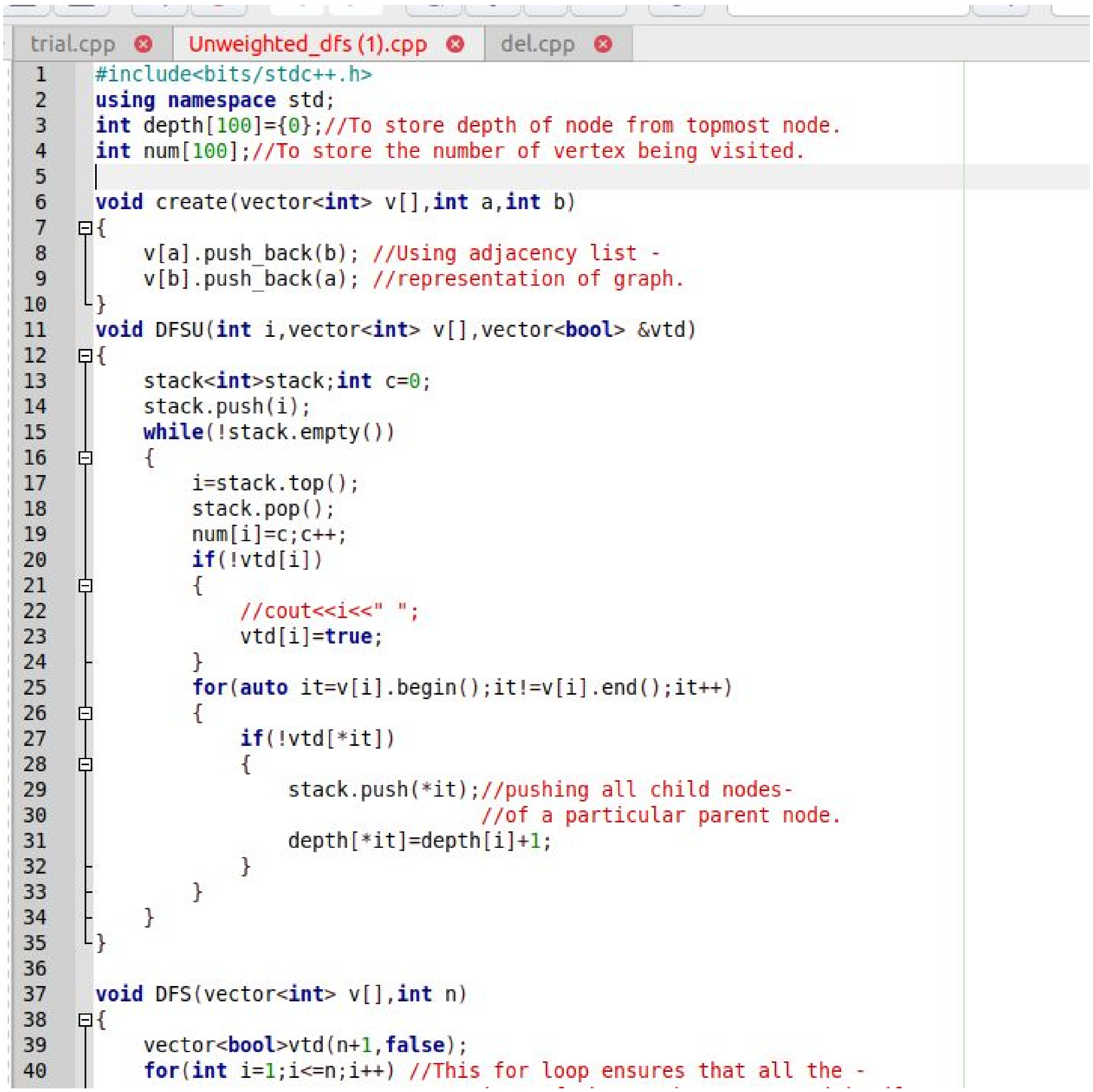
where D is the longest deadline.

THEOREM 4: The line-TRP with deadlines can be solved in O (mnD).

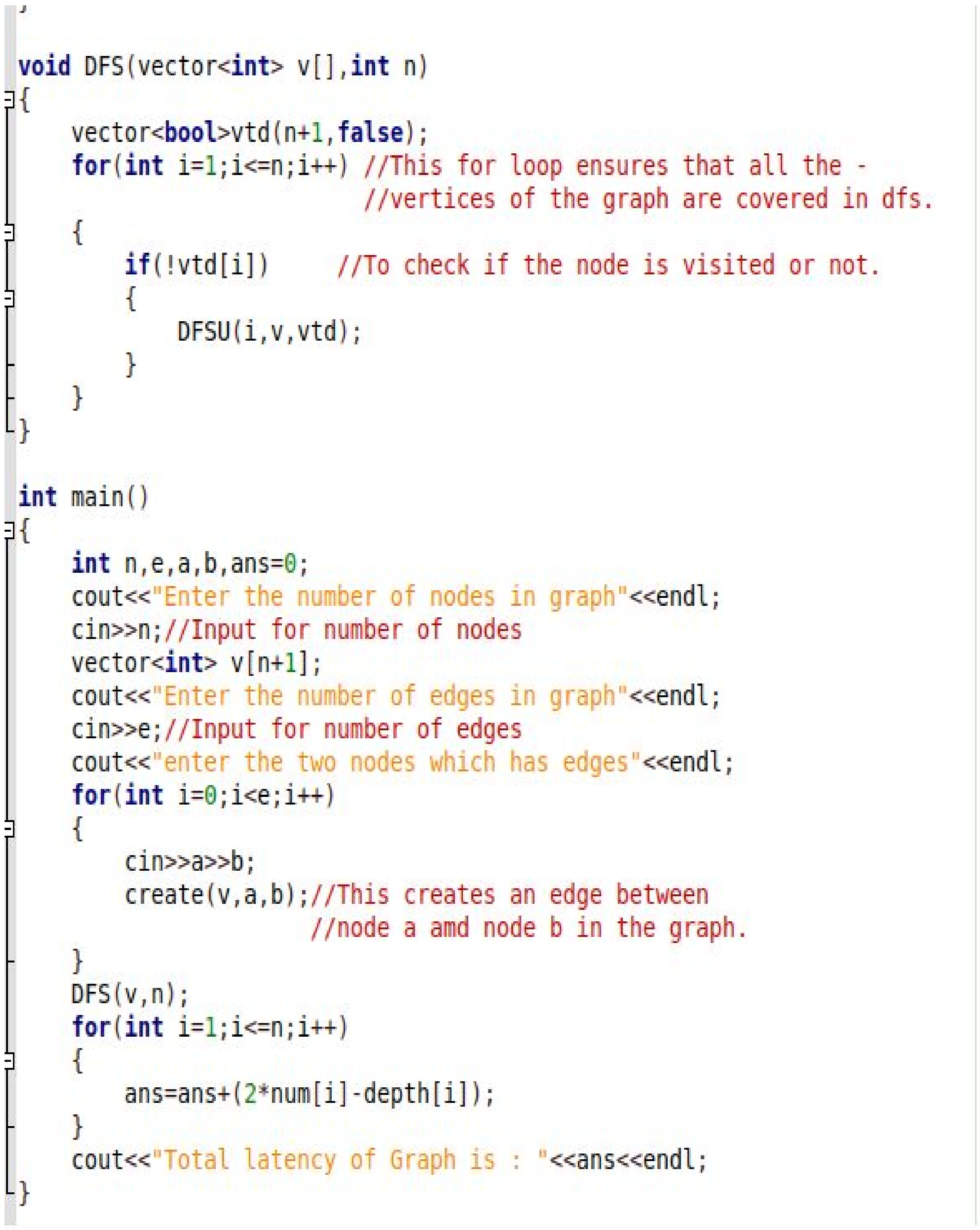
Sketch: By the straightforward implementation of the equations above, we can see that the number of steps involved in the implementation is $m \times n \times D$.
Hence it can be solved in O (mnD).

Implementation

5.1 Unweighted dfs

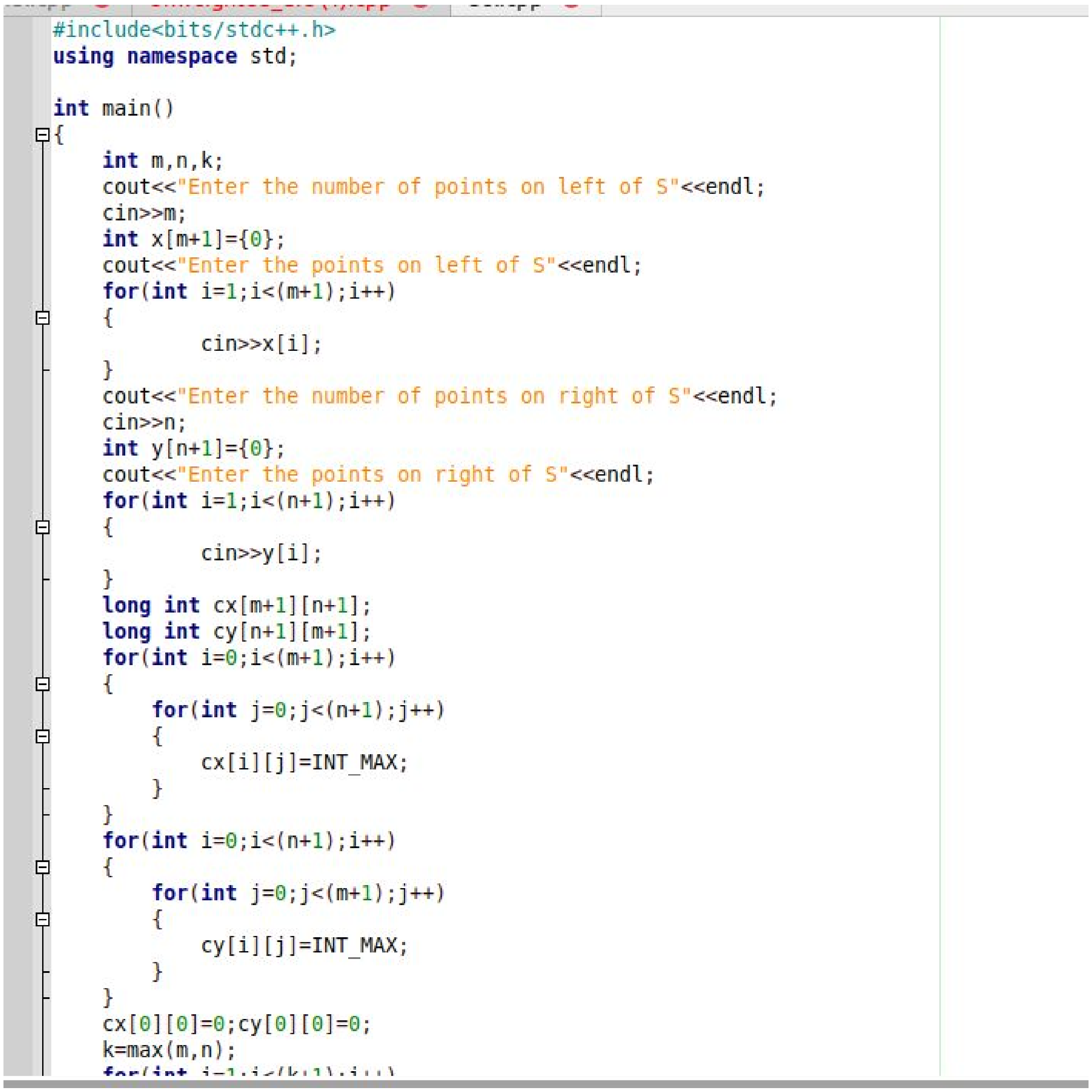


```cpp
#include<bits/stdc++.h>
using namespace std;
int depth[100]={0};//To store depth of node from topmost node.
int num[100];//To store the number of vertex being visited.

void create(vector<int> v[],int a,int b)
{
    v[a].push_back(b); //Using adjacency list -
    v[b].push_back(a); //representation of graph.
}
void DFSU(int i,vector<int> v[],vector<bool> &vtd)
{
    stack<int>stack;int c=0;
    stack.push(i);
    while(!stack.empty())
    {
        i=stack.top();
        stack.pop();
        num[i]=c;c++;
        if(!vtd[i])
        {
            //cout<<i<<" ";
            vtd[i]=true;
        }
        for(auto it=v[i].begin();it!=v[i].end();it++)
        {
            if(!vtd[*it])
            {
                stack.push(*it);//pushing all child nodes-
                                //of a particular parent node.
                depth[*it]=depth[i]+1;
            }
        }
    }
}

void DFS(vector<int> v[],int n)
{
    vector<bool>vtd(n+1,false);
    for(int i=1;i<=n;i++) //This for loop ensures that all the -
```

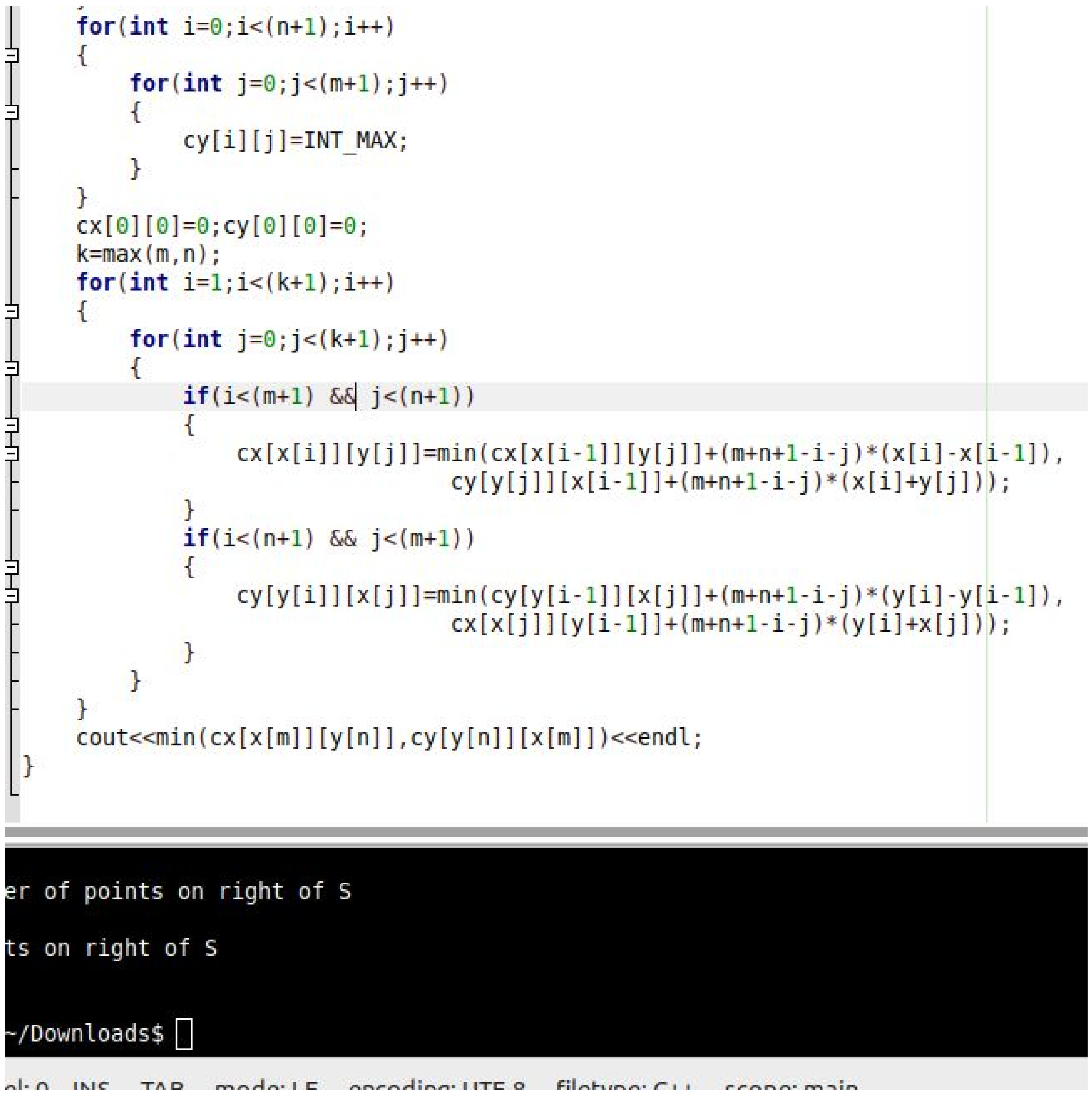


```
}

void DFS(vector<int> v[],int n)
{
    vector<bool>vtd(n+1,false);
    for(int i=1;i<=n;i++) //This for loop ensures that all the -
                          //vertices of the graph are covered in dfs.
    {
        if(!vtd[i])     //To check if the node is visited or not.
        {
            DFSU(i,v,vtd);
        }
    }
}

int main()
{
    int n,e,a,b,ans=0;
    cout<<"Enter the number of nodes in graph"<<endl;
    cin>>n;//Input for number of nodes
    vector<int> v[n+1];
    cout<<"Enter the number of edges in graph"<<endl;
    cin>>e;//Input for number of edges
    cout<<"enter the two nodes which has edges"<<endl;
    for(int i=0;i<e;i++)
    {
        cin>>a>>b;
        create(v,a,b);//This creates an edge between
                      //node a amd node b in the graph.
    }
    DFS(v,n);
    for(int i=1;i<=n;i++)
    {
        ans=ans+(2*num[i]-depth[i]);
    }
    cout<<"Total latency of Graph is : "<<ans<<endl;
}
```

5.2 Traveling repairman problem



```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    int m,n,k;
    cout<<"Enter the number of points on left of S"<<endl;
    cin>>m;
    int x[m+1]={0};
    cout<<"Enter the points on left of S"<<endl;
    for(int i=1;i<(m+1);i++)
    {
            cin>>x[i];
    }
    cout<<"Enter the number of points on right of S"<<endl;
    cin>>n;
    int y[n+1]={0};
    cout<<"Enter the points on right of S"<<endl;
    for(int i=1;i<(n+1);i++)
    {
            cin>>y[i];
    }
    long int cx[m+1][n+1];
    long int cy[n+1][m+1];
    for(int i=0;i<(m+1);i++)
    {
        for(int j=0;j<(n+1);j++)
        {
            cx[i][j]=INT_MAX;
        }
    }
    for(int i=0;i<(n+1);i++)
    {
        for(int j=0;j<(m+1);j++)
        {
            cy[i][j]=INT_MAX;
        }
    }
    cx[0][0]=0;cy[0][0]=0;
    k=max(m,n);
```



```
    for(int i=0;i<(n+1);i++)
    {
        for(int j=0;j<(m+1);j++)
        {
            cy[i][j]=INT_MAX;
        }
    }
    cx[0][0]=0;cy[0][0]=0;
    k=max(m,n);
    for(int i=1;i<(k+1);i++)
    {
        for(int j=0;j<(k+1);j++)
        {
            if(i<(m+1) && j<(n+1))
            {
                cx[x[i]][y[j]]=min(cx[x[i-1]][y[j]]+(m+n+1-i-j)*(x[i]-x[i-1]),
                                cy[y[j]][x[i-1]]+(m+n+1-i-j)*(x[i]+y[j]));
            }
            if(i<(n+1) && j<(m+1))
            {
                cy[y[i]][x[j]]=min(cy[y[i-1]][x[j]]+(m+n+1-i-j)*(y[i]-y[i-1]),
                                cx[x[j]][y[i-1]]+(m+n+1-i-j)*(y[i]+x[j]));
            }
        }
    }
    cout<<min(cx[x[m]][y[n]],cy[y[n]][x[m]])<<endl;
}
```

```
er of points on right of S

ts on right of S

~/Downloads$
```

```
ol: 0   INS   TAB   mode: LF   encoding: UTF-8   filetype: C++   scope: main
```

6. References

• Minimizing the average wait time of requests served by high traffic global servers. Avrim Blum, Prasad Chalasani, Don Coppersmith from School of Computer Science, CMU.

• A New Flo Formulation For The Minimum Latency Problem by Jao Sarubbi, Henrique Pacca Loureiro Luna.

• An experimental Study About Efficiency of the Approximation Algorithms for Minimum Latency Problem by Ha Bang Ban and Duc Nghia Nguyen.

• Computing Sharp Lower And Upper Bound for the Minimum Latency Problem by Joao Sarubbi, Henrique Pacca Loureiro Luna, Gilberto de Miranda Jr, Richardo Saraiva de Camargo.

• The complexity of the travelling repairman problem by Foto A Frati, Stavros Cosmadakis, Stavros Cosmandakis, Christos H, Papadimitriou, George Papageorgiou,Nadia Papakostantinou.