Exact algorithms for the minimum latency problem

Bang Ye Wu*, Zheng-Nan Huang, Fu-Jie Zhan
Dept. of Computer Science and Information Engineering, Shu-Te University,
YenChau, Kaohsiung, Taiwan 824, R.O.C.

Key words: algorithms, minimum latency problem, dynamic programming, branch and bound

1 Introduction

Let $G = (V, E, w)$ be an undirected graph with positive weight $w(e)$ on each edge $e \in E$. Given a starting vertex $s \in V$ and a subset $U \subset V$ as the demand vertex set, the *minimum latency problem* (MLP) asks for a tour P starting at s and visiting each demand vertex at least once such that the total latency of all demand vertices is minimized, in which the latency of a vertex is the length of the path from s to the first visit of the vertex. The MLP is an important problem in computer science and operations research, and is also known as the *delivery man problem* or the *traveling repairman problem*.

Similar to the well-known *traveling salesperson problem* (TSP), in the MLP we are asked to find an "optimal" way for routing a server passing through the demand vertices. The difference is the objective functions. The latency of a vertex can be thought of as the delay of the service. In the MLP we care about the total delay (service quality), while the total length (service cost) is concerned in the TSP. The MLP on a metric space is NP-hard and also MAX-SNP-hard [4]. Polynomial time algorithms are only known for very special graphs, such as paths [1, 6], edge-unweighted trees [9], trees of diameter 3 [4], trees of constant number of leaves [8], or graphs with similar structure [12]. Even for caterpillars (paths with edges sticking out), no polynomial time algorithm has been reported. In a recent work, it is shown that the MLP on edge-weighted trees is NP-hard [11]. Due to the NP-hardness, many works

*corresponding author (bangye@mail.stu.edu.tw)

have been devoted to the approximation algorithms [2, 3, 4, 7, 8], and the current best approximation ratio is 3.59 [5]. More references to exact and approximation algorithms can be found in those papers.

"Dynamic programming" (DP) and "branch-and-bound" (B&B) are two popular strategies used to exactly solve NP-hard problems without exhaustive search. As pointed out in [12], the MLP can be exactly solved by a dynamic programming algorithm. However, the algorithm is still very time-consuming. By designing non-trivial lower bound functions and using a technique combining the advantages of both DP and B&B, we developed a series of exact algorithms for the MLP. Experimental results on both random and real data are also reported in this paper. The results show that our algorithm is much more efficient than the DP algorithm and the B&B algorithm, and we believe that the technique can be also applied to some other problems.

2 Preliminaries

In this paper, a graph is a simple and connected graph with a nonnegative weight on each edge. Throughout this paper, the input graph is G, and n is the number of nodes of graph G. An origin (starting vertex) is a given vertex of G. 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 $d_G(u, v)$ to denote the length of the shortest path between u and v on G. For a subtour P, $d_P(u, v)$ 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 $d_P(s, v)$, which is the distance from the origin to the first visit of v on P. The latency of a tour P is defined by $L(P) = \sum_{v \in U} d_P(s, v)$, 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. Let $G = (V, E, w)$ be the underlying graph and $U \subset V$ be the demand vertex set. We first compute the metric closure $\bar{G} = (U, U \times U, \bar{w})$ of G, in which the weight on each edge is the shortest path length of the two endpoints in G. For any tour $\bar{P}$ on $\bar{G}$, we can construct a corresponding tour P on G by simply replacing each edge in $\bar{P}$ with the corresponding shortest path on G. It is easy to see that $L(P) \leq L(\bar{P})$. Conversely, given any tour P on G, we can obtain a tour $\bar{P}$ on $\bar{G}$ by eliminating all vertices not in U. Since the edge weight is the shortest path length, we have $L(\bar{P}) \leq L(P)$. 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.

3 Algorithms

3.1 Pure dynamic programming

To find the optimal tour of a MLP, a brute force algorithm checking all permutations of the vertices except for the origin will take $\Omega((n-1)!)$ time. In [12], it was pointed out that the MLP can be solved in $O(n^2 2^n)$ time by a dynamic programming algorithm. For the completeness, we briefly explain the algorithm in the following.

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 been visited.

Let P_1 and P_0 be two routes such that the last vertex of P_1 is the first vertex of P_0. We use $P_1//P_0$ to denote the route obtained by concatenating P_1 and P_0. 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 P_1 and P_2 be subtours with the same configuration and $c(P_1) \leq c(P_2)$. If $Y_2 = P_2//P_0$ is a complete tour, i.e., P_0 is a route starting at the last vertex of P_2 and visiting all the remaining vertices, then $Y_1 = P_1//P_0$ is also a tour and $L(Y_1) \leq L(Y_2)$.

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 $O(n2^n)$ configurations and we generate $O(n)$ subtours when a subtour is extended by one vertex.

3.2 Dynamic programming with pruning

To make the program more efficient, we introduce a *pruning* technique in the DP algorithm, which is similar to the one used in a typical branch-and-bound algorithm. While the program is running, we always record an upper bound (UB) of the optimal, which is the latency of some feasible tour. For each generated subtour P, we compute a lower bound of P, which is an under estimate of any complete tour containing P as a prefix. If the lower bound of a subtour is no less than UB, we can prune the subtour without affecting the optimality of the final solution. The key points are how we compute the UB and how we estimate the lower bound of a subtour.

A pure DP algorithm does not generate any complete tour until it reaches the configurations consisting of the set of all vertices. To get an upper bound, we employ a simple greedy algorithm to build a tour. The greedy algorithm uses the "nearest vertex first" strategy. Beginning with a subtour containing only the origin, we repeatedly augment the subtour by one vertex until all vertices are included. At each iteration, we choose the vertex which is

nearest to the stopping vertex of the subtour and has not been visited. Obviously, such a tour can be computed in $O(n^2)$ time. In addition to the initial stage, our algorithm uses the greedy method to build a tour whenever a new subtour is generated, and keep the current best solution.

Algorithm DPwP_MLP
Input: A metric graph $G = (V, E, w)$ and an origin $s \in V$.
Output: The latency of the optimal tour.
// Q_i is a queue for storing the generated subtours consisting of i vertices.
1: Initiate Q_1, and insert subtour (s) into Q_1.
2: Get an upper bound UB of the optimal.
3: For $i \leftarrow 1$ to $n-1$ do
4: For each subtour P in Q_i do
5: compute an upper bound UB$'$ from P;
6: if UB$'$ < UB
7: UB $\leftarrow$ UB$'$;
8: For each vertex v not in $V(P)$ do
9: generate a subtour $P' = P//(v)$;
10: if there exists a subtour with the same configuration in Q_{i+1}
11: keep the one with better $c(\cdot)$ value;
12: else
13: compute a lower bound LB of P';
14: if LB < UB then insert P' into Q_{i+1};
15: Output UB as the minimum latency.

At Step 10, we need to search a configuration in Q_{i+1}. In a typical DP algorithm, such a step can be implemented by employing an array, of which each element is for one configuration. By suitably encoding the configurations, the search can be done in only one memory access. However, such a simple method is not suitable for our algorithm since it requires to check every configuration, and this is what we want to avoid. Because of the large size of the queue, a good data structure should be used. In our program, we use an AVL tree. In the next section, we shall present the experimental results, and it shows that the improvement is very significant, compared to a link list implementation.

As in a typical B&B algorithm, the lower bound function is a key point to the efficiency of the algorithm. The running time depends heavily on two factors: the number of the generated subtours and the time to compute a lower bound of a subtour. A lower bound function eliminating many subtours may be bad if it suffers from a long computation time.

In the following, let $G = (V, E, w)$ be the input metric graph and s be the origin. Let P be a subtour stopping at a vertex r and $Y = P//P_0$ be the best tour consisting of P as its prefix. Let $\bar{V} = V - V(P)$, $\bar{n} = |\bar{V}|$, and $P_0 = (v_0 = r, v_1, v_2 \ldots, v_{\bar{n}})$. Remember that the best tour never visits a vertex twice in a metric graph. A function is a LB function of P if the latency of Y is lower bounded by the value of the function. We begin with a simple observation.

For any $1 \leq i \leq \bar{n}$, by the triangle inequality, we have

$$d_Y(s, v_i) = w(P) + d_Y(r, v_i) \geq w(P) + w(r, v_i).$$

Therefore,

$$\begin{aligned} L(Y) &\geq L(P) + \sum_{i=1}^{\bar{n}} (w(P) + w(r, v_i)) \\ &= L(P) + \bar{n}w(P) + \sum_{i=1}^{\bar{n}} w(r, v_i) \\ &= c(P) + \sum_{v \in \bar{V}} w(r, v). \end{aligned}$$

The following property is obvious, and we omit the proof.

Claim 2: The function $B_1(P) = c(P) + \sum_{v \in \bar{V}} w(r, v)$ is a LB function of P and can be computed in $O(n)$ time.

Next, we generalize the simple idea. Let $l_i(r, v)$ be the length of the shortest i-edges path between vertices r and v. Thereby an i-edges path is a path consisting of exactly i different edges. We first show the following property.

Lemma 3: For any vertices r and v, $l_i(r, v) \leq l_j(r, v)$ if $i < j$.

Proof: It is sufficient to show that $l_i(r, v) \leq l_{i+1}(r, v)$. Let $Q = (r, u_1, u_2, \ldots, u_{i+1} = v)$ be the shortest $(i + 1)$-edges path. Then $Q' = (r, u_2, \ldots, u_{i+1})$ is an i-edges path, and $w(Q') \leq w(Q)$ since $w(r, u_2) \leq w(r, u_1) + w(u_1, u_2)$ by the triangle inequality. By the definition of l_i, we have $l_i(r, v) \leq w(Q')$, and this completes the proof. □

Note that $l_1(r, v)$ is exactly $w(r, v)$ by definition. By the monotonic property of l_i, it is natural to use a more general l_i as the lower bound function. In the next theorem, we establish a family of lower bound functions. Note that the function B_1 coincides with the one in Claim 2.

Theorem 4: Let $k \geq 1$. The function

$$B_k(P) = c(P) + \sum_{v \in \bar{V}} l_k(r, v) - \sum_{i=1}^{k-1} \max_{v \in \bar{V}} \{l_k(r, v) - l_i(r, v)\}$$

is a LB function of P and can be computed in $O(kn)$ time if the value $l_i(r, v)$ is available for any $1 \leq i \leq k$ and any $v \in \bar{V}$.

Proof: Clearly $l_i(r, v_i) \leq d_Y(r, v_i)$ since $d_Y(r, v_i)$ is the length of an i-edges path while $l_i(r, v_i)$ is the minimum among all possible such paths. Furthermore, by Lemma 3, we have $l_i(r, v) \leq d_Y(r, v_j)$ for any $j \geq i$, and therefore, for $k \geq 1$,

$$\begin{aligned} L(Y) &= c(P) + \sum_{i=1}^{\bar{n}} d_Y(r, v_i) \\ &\geq c(P) + \sum_{i=1}^{\bar{n}} l_i(r, v_i) \\ &\geq c(P) + \sum_{i=1}^{k-1} l_i(r, v_i) + \sum_{i=k}^{\bar{n}} l_k(r, v_i) \end{aligned} \tag{1}$$

For $i < k$, we rewrite

$$l_i(r, v_i) = l_k(r, v_i) - (l_k(r, v_i) - l_i(r, v_i))$$

in Eq. (1), and obtain

$$\begin{aligned} L(Y) &\geq c(P) + \sum_{i=1}^{\bar{n}} l_k(r, v_i) - \sum_{i=1}^{k-1} (l_k(r, v_i) - l_i(r, v_i)) \\ &\geq c(P) + \sum_{v \in \bar{V}} l_k(r, v) - \sum_{i=1}^{k-1} \max_{v \in \bar{V}} \{l_k(r, v) - l_i(r, v)\} \end{aligned}$$

Finally the time complexity is obviously $O(kn)$. □

Although it is very time-consuming to compute l_k even for small k, we compute the values only once in a preprocessing stage. As a subtour is generated, we need only $O(kn)$ time to obtain a lower bound. We summarize the time complexity of the algorithm in the next theorem.

Theorem 5: The algorithm DPwP_MLP with lower bound function B_k runs in $O(n^{k+1} + n^2T)$ time, in which T is the number of generated subtours.

Proof: To employ B_k as the lower bound function, we compute $l_i(u, v)$ for any $1 \leq i \leq k$ and each vertex pair (u, v) in a preprocessing stage. Since $l_i(u, v)$ is the length of the shortest i-edges path and an i-edges path containing exactly $i-1$ intermediate vertices, all these values can be computed in $O(n^{k+1})$ time by exhaustively checking all possible permutations.

For each generated subtour, at Step 5–7, we compute a feasible tour and update the upper bound if necessary, and it takes $O(n^2)$ time. For searching the configuration in Q_{i+1} at Step 10, by employing an AVL tree, we perform $O(\log |Q_{i+1}|)$ comparisons of configurations. Since there are at most $n2^n$ configurations, the number of comparisons is $O(n)$. A configuration consists of a vertex and a set of up to n vertices. Comparing two configurations takes $O(n)$ time. Therefore, the total time for searching the AVL trees is $O(n^2T)$, in which T is the total number of generated subtours.

For Step 13, by Theorem 4, the time for computing the lower bounds of all subtours is $O(knT)$. For Step 14, since inserting an element into the AVL tree has the same time complexity as the searching, the total time for all the insertions is also $O(n^2T)$. In summary, the time complexity of the algorithm is therefore $O(n^{k+1} + n^2T)$. □

4 The experimental results

We implemented the algorithms in C language and investigated their practical performances. All the tests were performed on personal computers, each of which is equipped with an Intel Pentium IV 2.4 GHz CPU and 256M bytes memory. Two types of test data were used: random data and real data. For each test case, the running time includes all the steps except for generating or calculating the input distances.

4.1 Random data

The random data were generated artificially with edge weights drawn from uniform distribution. All the edge weights are integers between 1 and 1024. In Table 1, we summarize the maximum running time for each program in the tests on random data. Algorithm DPP(i) denote the algorithm DPwP_MLP with lower bound function B_i. For the sake of compari-

Table 1: The maximum running time in the random data tests (seconds, K=1000)

n	15	16	17	18	19	20	21	22	23
BF	10.5K	165K	-	-	-	-	-	-	-
DP	1.45	3.27	8.38	17.7	40.0	96.8	12.5K*	-	-
DPP_L	1.07	2.77	25.2	99.4	367	4.07K	8.41K	-	-
DPP(1)	0.30	0.50	2.03	4.19	11.3	43.7	81.0	180	11.7K*
DPP(2)	0.22	0.38	1.44	2.94	7.66	29.1	54.6	166	302
DPP(3)	0.17	0.28	0.91	2.03	5.27	17.7	37.2	128	247
DPP(4)	0.25	0.47	1.00	2.06	4.91	11.1	25.8	96.6	176
DPP(5)	1.45	2.56	4.92	8.03	13.7	22.7	40.5	105	165
B&B(1)	1.80	3.91	15.0	55.7	161	1.77K	2.97K	6.50K	-

Table 2: The maximum number of generated subtours in random data tests (M=10^6)

	DP	DPP(1)	DPP(2)	DPP(3)	DPP(4)	DPP(5)	B&B(1)
$n = 15$	114689	25825	17737	11835	7624	5147	0.54M
$n = 18$	1.1M	0.23M	0.12M	0.11M	78989	69855	14.8M
$n = 21$	10.5M	2.96M	1.99M	1.32M	0.83M	0.51M	593M
$n = 23$	-	9.17M	7.35M	6.51M	4.51M	3.26M	-

son, we also implemented the brute-force method (labeled by BF) and the branch-and-bound method (labeled by B&B(1), using the lower bound function B_1). The BF computes the optimal solution by simply checking all the possible permutations. The B&B(1) program is similar to DPP(1) except that it does not merge the subtours with the same configuration. It uses the depth-first strategy to choose the subtour to be extended, and the chosen subtour is augmented by each of the vertices not been visited yet. In fact, we also implemented the branch-and-bound method with B_i, $i > 1$. But their behaviors are similar, and we only list B&B(1) for comparison. Algorithm DPP_L is the same as DPP(1) but using a link list instead of an AVL tree as the data structure for storing the configurations.

Basically at least one hundred data instances were used for each problem size. But, for BF and DP, only few instances are tested because their performances almost do not vary with the input data of the same number of vertices. Some cells in the table are marked with "-" to indicate that we did not complete the tests on these cases because some data instances took too long to complete. A "*" in a cell indicates that the long running time is caused by "disk swap" in the virtual memory system. In Table 2, we list the maximum number of subtours generated by each program for some typical values of n.

Table 3: The running time in the real data tests (seconds)

Data name	DPP(1)	DPP(2)	DPP(3)	DPP(4)	DPP(5)	B&B(1)
Ulysses16	0.09	0.08	0.09	0.13	0.33	0.45
Ulysses22	3.40	3.53	3.50	3.42	5.55	54.47
Gr24	54.47	51.54	43.64	34.41	30.23	285.17
Fri26	39.61	37.64	32.75	26.09	27.41	257.60

4.2 Real data

In addition to the random data, we also used real data to test the performances of the algorithms. The data instances are chosen from TSPLIB [10] for the sake of their problem sizes. The results are shown in Table 3. Note that the number appeared in the name indicates the number of vertices for each instance. In fact, we have also performed some other tests on partial data drawn from larger instances in TSPLIB. The results are similar. Roughly speaking, problems with 25–26 vertices can be solved in about 100 seconds. Comparing with the results of random data, the performances are much better. The reason may be that the real data are more structured and therefore the bad cases rarely happen.

5 Discussion and concluding remarks

By the experimental results and some other observations in our development, we make the following conclusions.

- The algorithm DPwP_MLP takes the advantages of both the dynamic programming and the branch-and-bound strategies, and significantly improves the performance.
- Using a good data structure such as the AVL tree in our program is very important. The reason is obvious by knowing the numbers of the generated subtours (Table 2).
- For small integers $j > i$, DPP(j) is better than DPP(i) when n exceeds some value.
- Theoretically, we can improve the lower bound by restricting that the i-edge path can only visit the vertices in $\bar{V}$. But it suffers from a long computation time and therefore has a worse performance. In fact, we have tried several other lower bound functions. Some of them eliminate much more subtours than B_1 but has a worse performance.

References

[1] F. Afrati, S. Cosmadakis, C. Papadimitriou, G. Papageorgiou, and N. Papakostantinou, The complexity of the traveling repairman problem, *Theoretical Informatics and Applications*, 20(1)(1986) 79–87.

[2] A. Archer and D.P. Williamson, Faster approximation algorithms for the minimum latency problem, in *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms* (SODA 2003), 2003 pp. 88–96.

[3] S. Arora and G. Karakostas, Approximation schemes for minimum latency problems, *SIAM J. Comput.*, 32(5)(2003) 1317-1337.

[4] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan, The minimum latency problem, in *Proc. 26th ACM Symposium on the Theory of Computing* (STOC'94), 1994 pp. 163–171.

[5] K. Chaudhuri, B. Godfrey, S. Rao, and K. Talwar, Paths, Trees, and Minimum Latency Tours, in *Proc. 44th Symposium on Foundations of Computer Science* (FOCS 2003), 2003 pp. 36–45.

[6] A. Garcia, P. Jodrá, and J. Tejel, A note on the traveling repairmen problem, *Networks*, 40(1)(2002) 27–31.

[7] M. Goemans and J. Kleinberg, An improved approximation ratio for the minimum latency problem, *Math. Program.*, 82(1998) 114–124.

[8] E. Koutsoupias, C. Papadimitriou and M. Yannakakis, Searching a fixed graph, in *Proc. 23nd Colloquium on Automata, Languages and Programming*, Lecture Notes in Comput. Sci., Vol. 1099, 1996, pp. 280–289.

[9] E. Minieka, The delivery man problem on a tree network, *Ann. Oper. Res.*, 18(1989) 261–266.

[10] G. Reinelt, TSPLIB — a traveling salesman problem library, *ORSA J. Computing*, 3(1991) 376–384. See also http://www.iwr.uni-heidelberg.de/groups/comopt/software-/tsplib95/.

[11] R. Sitters, The minimum latency problem is NP-hard for weighted trees, in *Proc. 9th International IPCO Conference*, Lecture Notes in Comput. Sci., Vol. 2337, 2002, pp. 230–239.

[12] B.Y. Wu, Polynomial time algorithms for some minimum latency problems, *Inf. Process. Lett.*, 75(5)(2000) 225–229.