An Effective GRASP+VND Metaheuristic for the k-Minimum Latency Problem

Ha Bang Ban
School of Information and Communication Technology
Hanoi University of Science and Technology
Email: BangBH@soict.hust.edu.vn

Duc Nghia Nguyen
School of Information and Communication Technology
Hanoi University of Science and Technology
Email: NghiaND@soict.hust.edu.vn

***Abstract*— Minimum Latency Problem (MLP) is a class of NP-hard combinatorial optimization problems which has many practical applications. In this paper, a general variant of MLP, also known as $k-$MLP is introduced. In $k-$MLP problem, the cost of objective function becomes the sum of waiting times at sites and k vehicles cover one of k routes. The goal is to find the order of customer visits that minimizes the sum of waiting time. The problem is a natural and practical extension of the $k = 1$ case. In our work, we propose the first meta-heuristic algorithm which is mainly based on the principles of Greedy Randomized Adaptive Search Procedure (GRASP) and Variable Neighborhood Descent (VND) to solve the problem. The GRASP is used to build an initial solution which is good enough in construction phase. In a cooperative way, the VND is employed to generate diverse neighborhoods in improvement phase, therefore, it can prevent the search to escape from local optimal. In addition, we also introduce a new novel neighborhoods' structure in VND. In order to evaluate the performance of our algorithm, we also discuss an upper bound and a lower bound of the optimal solution. Extensive numerical experiments on benchmark instances show that our algorithm finds good-quality solutions fast, even for large instances which are up to 1200 customers.**

I. Introduction

The $k-$Minimum Latency Problem ($k - MLP$), in the special case where $k = 1$, has been studied in the number of previous work [1], [2], [3], [5], [6], [7], [10], [11], [13], and is often called the Minimum Latency Problem, the Deliveryman Problem or the Travelling Repairman Problem. It arises many practical applications, e.g., whenever repairmans or servers have to accommodate a set of requests so as to minimize their total (or average) waiting time [1], [2], [10], [11]. In general case, the $k-$ Minimum Latency Problem has, to the best our knowledge, previously not been studied much, even though it is a natural and practical extension of the $k = 1$ case, except F. Jittat et al. [14] gave the definition and approximation algorithm for the problem. Informally, in the $k-$MLP we are given that there are k vehicles at a main depot s an n customers. The goal is to find tours to send the vehicles that minimize the average time a customer has to wait, while making sure that all customers are served. Minimizing average waiting time is a very natural objective. As noted by Blum et al. [10], the $k-$MLP can even be viewed as trying to minimize the expected search time to find an object, given that it is likely to be at any vertex. This variant is at least as hard as the MLP, and it is also NP-hard problem, however, the reduction from the general MLP to the problem in a metric case can be done by a simple transformation as in [19]. The metric case reflects a condition in which a complete graph with distances between vertices satisfying the triangle inequality. In this paper, we consider the problem in the metric case, and formulate the $k-$MLP as follows:

Given a complete graph K_n with the vertex set $V = \{1, 2, \ldots, n\}$ and a symmetric distance matrix $C = \{c(i, j) \mid i, j = 1, 2, \ldots, n\}$, where $c(i, j)$ is the distance between two vertices i and j. Let $R = (1, 2, ..., k)$ be a set of k vehicles which begin at the main deport v_1. Suppose that the tour $T = (R_1, \ldots, R_l, \ldots, R_k)$ is a set of obtained routes from k vehicles. Let $R_l = (v_1, \ldots, v_h, \ldots, v_m)(1 < m \leq n)$ be a route of vehicle $l(l \in k)$. Denote $P(v_1, v_h)$ is the path from v_1 to v_h on the route R_l and $l(P(v_1, v_h))$ is its length. The latency of a vertex $v_h(1 < h \leq m)$ on T_l is the length of the path from starting vertex v_1 to v_h:

$$l(P(v_1, v_h)) = \sum_{i=1}^{h-1} c(v_i, v_{i+1}).$$

The latency of the route R_l is defined as the sum of latencies of all vertices in this route:

$$L(R_l) = \sum_{h=2}^{m} l(P(v_1, v_h)).$$

The total latency of T crossing all vertices is the sum of all the routes' latencies:

$$L(T) = \sum_{l=1}^{k} L(R_l).$$

The $k-$MLP asks for minimum latency tour, which starts at a given vertex v_1 and visits each vertex in the graph once exactly.

As an example of the $k-$MLP, in Figure 1 we show a tour includes four vehicles which start from a main depot for a 2-dimensional Euclidean instance.

For NP-hard problems, such as the MLP or $k-$MLP, there are three common approaches to solve it, namely, 1) exact algorithms, 2) approximation algorithms, 3) heuristic algorithms. Firstly, the exact algorithms guarantees to find the optimal solution and take exponential time in the worst case, but they often run much faster in practice. However, the exact

978-1-5090-4134-3/16/$31.00 ©2016 IEEE

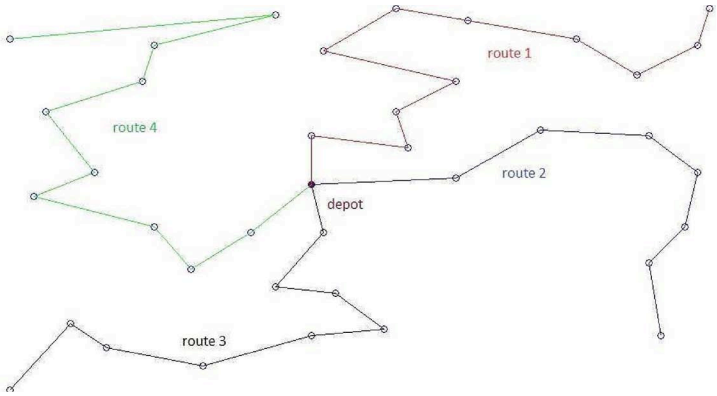

Fig. 1: The $k-$MLP solution with 4 vehicles start at the depot v_1

algorithms only solve small sizes (up to 40 vertices for the MLP in which $k = 1$. It is likely to be even smaller in the general case $k-$MLP) [5], [19]. Secondly, the approximation algorithm produces a solution within some factor alpha of the optimal solution [1], [2], [4], [6], [10], [11], [13]. In this approach, the best theoretical approximation ratio of 3.59 and 16.994 is for the MLP and $k-$MLP [11], [14], respectively, however, it is still far from the optimal solution. Thirdly, heuristic algorithms perform well in practice and validate their empirical performance on an experimental benchmark of interesting instances. The meta-heuristic algorithm depends on this approach. The experimental results for MLP in [3], [7], [8], [9] indicate that meta-heuristic algorithms are a suitable approach since they can find good-quality solutions fast, even for large instances.

Previously, research on the $k-$MLP has not studied much and this work presents the first meta-heuristic approach for this problem. Our meta-heuristic algorithm is mainly based on the principles of Variable neighborhood descent (VND) [15] and Greedy Randomized Adaptive Search Procedure (GRASP) [12] to solve the problem. The GRASP is used to build an initial solution which is good enough in the construction phase. In a cooperative way, the VND is employed to generate diverse neighborhoods in the improvement phase, therefore, it can help prevent the search to escape from local optimal. Moreover, we introduce several new novel neighborhoods' structure in VND, and indicate a sequential order to explore these neighborhoods in which the VND can give the best solutions. In order to evaluate the performance of our algorithm, we also discuss an upper bound and a lower bound of the optimal solution for the problem. Extensive numerical experiments on benchmark instances show that our algorithm finds good-quality solutions fast, even for large instances which are up to 1200 customers.

The rest of this paper is organized as follows. Section 2 presents the proposed algorithm. To evaluate the performance of our approach, we discuss an upper bound and a lower bound in Section 3. Computational evaluations are reported in section 4 and section 5 concludes the paper.

II. The GRASP + VND Algorithm

Our algorithm includes two phases: 1) A greedy randomized construction phase and a variable neighborhood descent improvement phase. In the first phase, GRASP (Greedy Randomized Adaptive Search Procedure) [12] allows a controlled amount of randomness to overcome the behaviour of a purely greedy heuristic. It is used to build an initial solution which is good enough. In the second, VND [15] is based on the principle of systematically exploring several different neighborhoods, combined with a shaking move to escape from local optima. Our algorithm is repeated a number of times, and the best solution found is reported. An outline of the algorithm is shown in Algorithm 1. In step 1, the algorithm starts with an initial solution obtained from the GRASP [12]. In Step 2 phase, we investigate a novel neighborhoods' structure in VND and explore systematically switches between different neighborhoods. Therefore, VND provides our algorithm diverse neighborhoods. In order to explore the entire solution space, a diversification step is added in step 3. In the remaining of this section, more details about the three steps of our algorithm are given.

Step 1: We use the GRASP for finding an initial solution [12]. In this step, a feasible solution is built, one vertex at a time for each route. At each constructive step, a random route is selected from k routes in Tour T. Then, a restricted candidate list (RCL) is determined by ordering all non selected vertices with respect to a greedy function that measures the benefit of including them in the route. After that, one element is randomly chosen from the restricted candidates list and added into the route. Since all vertices are visited, the algorithm stops and the initial solution is returned. The size of RCL is a parameter of the GRASP that controls the balance between greediness and randomness. The GRASP for our algorithm is described in Algorithm 2.

Step 2: In this step, five neighborhoods investigated are divided two categories: 1-route, and 2-route. In 1-route is used as a post-optimizer on single vehicle routes. It consists of applying remove-insert, swap-adjacent, swap, 2-opt [18]. Meanwhile, solution improvements can often be obtained by

Algorithm 1 The Proposed Algorithm

Input: $v_1, V, N_i(T)(i = 1, ..., 5), pos$ are a starting vertex, the set of vertices in K_n, the set of neighborhoods and the number of swap, respectively.
Output: The best solution T^*.
Step 1 GRASP(Generate an initial solution):
$T = \phi$; //T is the Tour
for $(l = 1; l < k; l++)$ **do**
$R_l = R_l \cup v_1$; //The $l-th$ route of the tour T starts at the depot v_1
end for
$L = \phi$; //L is the list of visited vertices
while $|L| < n$ **do**
$l =$ random(k);//Choose a route randomly in k routes.
//v_t is the last vertex in route R_l;
Create RCL with α vertices $v_i \in V$ closest to v_t;
Select randomly vertex $v_t = \{v_i | v_i \in RCL$ and $v_i \notin R_l\}$ to add to R_l;
$L = L \cup v_t$;
end while
for $(l = 1; l < k; l++)$ **do**
$T = T \cup R_l$;//Update $l-th$ route in the tour T
end for
while stop criteria not met **do**
Step 2 (VND):
for $i : 1 \rightarrow 5$ **do**
$T' \leftarrow argmin_{T'' \in N_i(T)} L(T'')$
if $((L(T') < L(T))$ or $(L(T') < L(T^*)))$ **then**
$T \leftarrow T'$
end if
if $(L(T') < L(T^*))$ **then**
$T^* \leftarrow T'$
else
$i++$
end if
end for
step 3 (Implement Diversification):
type = rand(2);//Select randomly a number from 1 to 2.
if type==1 **then**
R_l = Select randomly the $l-th$ route of T;
$R_l \longleftarrow$ Shaking-1-route(R_l, pos);
else
R_l and R_h = Select randomly two routes of T;
R_l and $R_h \longleftarrow$ Shaking-2-routes(R_l, R_h, pos) ;
end if
end while
return T^*;

Algorithm 2 Shaking-1-route(R_l, pos)

Input: R_l, k, pos are the $l-th$ route, the number of vehicles and the number of swap, respectively.
Output: a new solution R_l.
while $(pos > 0)$ **do**
select i, j positions from R_l at random
if $(i \neq j)$ **then**
Insert $R_l[i]$ between $R_l[j]$ and $R_l[j+1]$;
$pos \longleftarrow pos - 1$;
end if
end while
return R_l;

Algorithm 3 Shaking-2-routes(R_l, R_h, pos)

Input: R_l, R_h, k, pos are the $l-th$, $h-th$ route, the number of vehicles and the number of swap, respectively.
Output: a new solution R_l and R_h.
while $(pos > 0)$ **do**
select $i-th$ and $j-th$ positions from R_l and R_h at random, respectively;
swap $R_l[i]$ between $R_h[j]$;
$pos \longleftarrow pos - 1$;
end while
return R_l and R_h;

Algorithm 4 The nearest heuristic

Input: v_1, V, k are a starting vertex, the set of vertices in K_n, the number of vehicles, respectively.
Output: the initial solution T.
$T = \phi$; //T is the Tour T
for $(l = 1; l < k; l++)$ **do**
$R_l = R_l \cup v_1$; //start at the depot v_1
end for
$L = \phi$; //L is the list of visited vertices
while $|L| < n$ **do**
$l =$ random(k);//Choose a route randomly in k routes.
Select the nearest vertex $v_t = \{v_i | v_i \notin L$ and $v_i \notin R_l\}$ to add to R_l;
$L = L \cup v_t$;
end while
for $(l = 1; l < k; l++)$ **do**
$T = T \cup R_l$
end for
return T;

moving vertices belonging to two or more different routes in 2-route. In this work, we introduce new neighborhoods in 2-route such as swap-2-route, and insert-2-route. For a given current solution T, neighborhood explores the neighboring solution space set $N(T)$ of T iteratively and tries to replace T by the best solution $T' \in N(T)$. The main operation in exploring the neighborhood is the calculation of a neighboring solutions latency. In straightforward implementation, this operation requires $Tsol = O(k \times max(|R_1|, |R_2|, ..., |R_L|, ..., |R_k|))$ ($|R_l|$ is the size of $l-th$ route in T). Now, let $T = (R_1, R_2, ..., R_l, ..., R_k)(l = 1, ..., k)$ be a tour, we introduce a novel neighborhoods structure and complexity of its exploration.

For 1-route: 1-route is used only to optimizer on single route.

We investigate four neighborhoods' structure in turn.

- **The swap-adjacent** (see in Fig. 1 in [23]) attempts to swap each pair of adjacent vertices in R_l. The complexity of exploring the neighborhood is $O(Tsol \times |R_l|)$.
- **The remove-insert** (see in Fig. 2 in [23]) considers each vertex v_i in R_l and to places the vertex furthest away from this vertex v_i at the end of swap-adjacent the route R_l. The complexity of exploring the neighborhood is $O(Tsol \times |R_l|)$.
- **The swap** (see in Fig. 3 in [23]) tries to swap the positions of each pair of vertices in the single route T_l. The complexity of exploring the neighborhood is $O(Tsol \times |R_l|^2)$.
- **The 2-opt** (see in Fig. 4 in [23]) removes each pair of edges from the route R_l and reconnects the vertices. The complexity of exploring the neighborhood is $O(Tsol \times |R_l|^2)$.

For 2-route: Let R_l, $|R_l|$ and R_h, $|R_h|$ be two different routes and their sizes in T, respectively. 2-route is used to moving vertices in two different routes as followings:

- **The swap-2-route** (see in Fig. 5 in [23]) tries to exchange the positions of each pair of vertices in R_l and R_h. The complexity of exploring the neighborhood is $O(Tsol \times |R_l| \times |R_h|)$.
- **The insert-2-route** (see in Fig. 6 in [23]) considers each vertex v_i in R_l and insert it into each position in R_h. The complexity of exploring the neighborhood is $O(Tsol \times |R_l| \times |R_h|)$.

In preliminary study, we realize that the efficiency of VND algorithm relatively depends on the order in which the neighborhoods are used. Therefore, the neiborhoods are explored in a specific order based on the size of their structrure, namely, from small to large, such as swap-adjacent, remove-insert, swap, 2-opt, swap-2-route, insert-2-route.

Step 3: Shaking procedure allows to guide the search towards an unexplored part of the solution space. In this work, two types of shaking are used to give a new solution such as shaking in a single route (Shaking-1-route) and shaking in two routes (Shaking-2-routes). In shaking procedure in a single route, it selects the l-th route R_l of T and then swaps randomly several vertices for each other. In the rest, it picks two routes R_l and R_h in random manner and after that, exchanges randomly some several vertices in them. We finally return to step 2 with the new solution. The Shaking procedure is described in Algorithm 3.

The last aspect to discuss is the stop criterium of our algorithm. A balance must be made between computation time and efficiency. Here, the algorithm stops if no improvement is found after the number of loop (*NL*).

The running time of our algorithm mainly spends for exploring in VND. In VND, swap-2-route and insert-2-route, in which their time complexity is not less than those of any neighborhoods, run in $O(Tsol \times |R_l| \times |R_h|)$ time. Assume that k_1 is the number of runs of them, therefore our algorithm requires $O(k_1 \times Tsol \times |R_l| \times |R_h|)$ time.

III. BOUNDS FOR THE $k - MLP$

In order to evaluate the efficiency of our algorithm, our algorithm can be compared to 1) the optimal solutions or their lower bound; 2) the initial solution of the construction phase (GRASP); and 3) a good enough upper bound (UB). Due to the properties of the problem, giving good bounds is not easy.

In this work, we use the nearest neighbourhood heuristic to obtain an upper bound. The basic idea of this heuristic is to construct an initial solution by starting with the depot and adding repeatedly the nearest vertex to the last-one until all vertices are visited. Clearly, it is a special of GRASP with RCL only contains the nearest vertex, namely $RCL = 1$. The nearest neighbourhood heuristic is described detail in Algorithm 4.

Our lower bound for the $k-$MLP is computed by sorting the edges of the graph in order of increasing weight and multiplying each edge with a factor. The k smallest edges are multiplied with $n-1$ (each smallest edge for the one of k vehicles), the k second-smallest with $n-2$, etc, the k largest edges are multiplied with a factor 1. For example, assume that $E^{'} = (e_1, e_2, ..., e_k, e_{k+1}, ..., e_n)$ is a set of the cheapest costs in the matrix cost C such that $e_1 < e_2 < ... < e_n$ and there are k vehicles. The value of the lower bound is calculated as follows:

$$\begin{aligned} LB &= (n-1) \times (e_1 + e_2 + ... + e_k) \\ &\quad +(n-2) \times (e_{k+1} + e_{k+2} + ... + e_{2k}) \\ &\quad +... + 1 \times (e_{n-k} + e_{n-k+1} + ... + e_n) \end{aligned}$$

Now we define the improvement of our algorithm in comparison with the lower bound of the optimal solution ($gap_1[\%]$), the initial solution ($improv[\%]$) and the upper bound ($gap_2[\%]$) in percent respectively as follows:

$$gap_1[\%] = \frac{Best.Sol - LB}{LB} \times 100\% \quad (1)$$

$$improv[\%] = \frac{Best.Sol - Init.Sol}{Init.Sol} \times 100\% \quad (2)$$

$$gap_2[\%] = \frac{Best.Sol - UB}{UB} \times 100\% \quad (3)$$

It is showed that the lower bound is crude in general, however, it is also an important measurement of the improvement of our algorithm.

IV. COMPUTATIONAL EVALUATION

The experiments are conducted on a personal computer, which is equipped with an Intel Pentium core i7 duo 2.10 Ghz CPU and 4 GB bytes RAM memory.

A. Datasets

The numerical analysis was performed on a set of benchmark problems for capacitated VRP described in [20]. As testing our algorithm on all instances would have been computationally too expensive we implemented our numerical analysis on some selected instances. Firstly, in order to eliminate

size effects problems with approximately from 25 up to 1200 customers are chosen. Also, in order not to bias the results by taking "easy" or "hard" instances we randomly choose one hundred instances. These are:

- Christofides et al. [20]: This dataset includes 14 instances (CMT1, CMT2, ..., CMT14), which varies the number of vertices from 50 to 200 and vehicles from 5 to 18.
- Converted TSPLIB [20]: Some instances translated from TSPLIB to generate several instances for k-MLP such as att-n48-k4, bayg-n29-k5, dantzig-n42-k4, gr-n21-k3, gr48-k3, swiss-n42-k5.
- Taillard et al. [20]: Ten instances from 75 to 150 vertices are picked randomly from Taillard et al., specifically, tai75a, tai75b, tai75c, tai100a, tai100b, tai100c, tai150a, tai150b, tai150c.
- Augerat et al. [20]: Thirty instances of Dataset A, B, and P are selected from Augerat et al., which varies the number of vertices from 32 to 101 and vehicles from 4 to 10.
- Li et al. [20]: For very large instances, we choose randomly twenty instances from Li et al. with the number of vertices from 241 to 1200 and vehicles from 9 to 11.
- BangBH et al. [21]: Twenty small instances ($n \leq 30$) in this dataset are generated in a special way in which the costs of edges in the matrix cost are close to each other. In general, it is shown that our lower bound is crude. However, since the edges' costs are distributed closely to each other, the quality of the lower bound can be acceptable.

For all instances, the optimal solutions of the problem have been unknown, therefore, the efficiency of our algorithm only evaluate relatively according to formula from (1) to (3).

B. Results and Discussion

The results for those dataset are reported for thirty independent runs, and in each run our algorithm was executed with parameters as follows: $NL = 100, pos = 10$, and $RCL = 15$.

In the tables presented in [22], *Best.Sol*, *Aver.Sol*, *Aver.Time* and $STDV$ correspond to the best solution, the average solution, the average time in seconds and the standard deviation of ten executions obtained by our algorithm, respectively. Tables 1, ..., 7 in [22] compare the results of our algorithm with the upper bounds obtained by using the nearest neighborhood heuristic and the initial solution using GRASP. The experimental results are illustrated in Table 1 and 2, which are the average values calculated from Table 1 to 7 in [22]. Table 1 and 2 present a summarized comparison between the average gaps, in respect to $UB, Init.Sol, LB$ in small and large instances, respectively.

Experimental results for small instances

The experimental results, which are averagely calculated from Table 1 and 2 in [22], are illustrated in Table 1 in this paper. In Table 1, we denote $\overline{Gap_1}, \overline{Gap_2}$ and $\overline{T}$ by the average values of Gap_1, Gap_2 and T for the dataset. From Table 1, it is obvious that in average, the improvement of our algorithm upon these construction phase solutions is quite good (2.36% for test-20-x, 6.82% for test-30-x). Although the difference in the gap between the construction phase solution and our solutions is not too large, but significant, since they are within 10.5% - 15.7% of crude lower bound in a reasonable amount of time.

TABLE I: $\overline{Gap_1}, \overline{Gap_2}$ for small dataset

Instances	$\overline{Gap_1}$	$\overline{Gap_2}$	$\overline{T}$
test-20-x	10.5	2.36	0.11
test-30-x	15.6	6.82	0.17

TABLE II: $\overline{Gap_1}, \overline{Gap_2}$ for large dataset

Instances	$\overline{Gap_2}$	$\overline{improv}$	$\overline{T}$
Christofides et al.	42.74	40.46	5.85
Taillard et al.	54.80	52.60	5.29
Converted TSPLIB	26.82	24.70	0.37
Augerat et al.	27.83	23.66	0.87
Li et al.	63.00	60.00	1847

Experimental results for large instances

The experimental results, which are average values calculated from Table 3 to 7, are illustrated in Table 2 in this paper. In Table 2, we denote $\overline{Gap_1}, \overline{Gap_2}$ and $\overline{T}$ by the average values of Gap_1, Gap_2 and T for these datasets. In the fig. 2 and 3, the horizontal axes present the number calls to VND and vertical axes show $\overline{improv}$ and $\overline{T}$ by seconds in Table 3, respectively.

In Table 2, for all instances, it can be observed that our algorithm is capable of improving the solutions in comparison with UB and $Init.Sol$. The average improvement of our algorithm with average Gap_2 between 26.82% and 63%, and $improv$ between 23.66% and 60%. Obviously, our algorithm can obtain a significant improvement for almost instances and required smaller scaled running time. Moreover, the average value of $STDV$ is quite small. That means the quality of our algorithm is very stable.

Figure 2 and 3 shows the evolution of the average deviation to the initial solutions with respect to $\overline{improv}$ and $\overline{T}$ during the iterations on some instances. The deviations are 36.6%, 38.3%, 39.1%, 39.7%, 39.9%, and 40.3% for the first local optimum, obtained by one, ten, twenty, thirty, fifty and one-hundred calls VND, respectively. A major part of the descent obtained is about 0.4% by from fifty to three-hundred calls VND. As can be observed, additional iterations give a minor improvement with the large running time. Hence, the first way to reduce the large running time is to use no more than fifty calls to VND and the improvement of our algorithm is about 39.9%. A much faster option is to run the initial construction phase then improve it by using a single call to VND, which obtains an average deviation of 36.6% and average time of 105 seconds, even for the instances which are up to 1200 customers.

V. CONCLUSIONS

In this paper, we propose the first meta-heuristic algorithm which is mainly based on the principles of VND and GRASP to solve the problem. Moreover, we introduce a new novel

TABLE III: Evolution of average deviation to $Init.Sol$

Instances	Our algorithm															
	1 call to VND		10 calls to VND		20 calls to VND		30 calls to VND		50 calls to VND		100 calls to VND		200 calls to VND		300 calls to VND	
	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$	$\overline{impro}$	$\overline{T}$
Christofides	33.3	0.40	37.1	0.76	38.7	1.16	39.4	1.70	40.1	2.81	40.5	5.85	40.5	10.87	40.5	16.12
Taillard et al.	48.69	0.41	50.93	0.79	51.79	1.11	52.15	1.58	52.45	2.62	52.6	5.29	52.61	11.01	52.61	16.59
Converted TSPLIB	22.87	0.003	23.36	0.03	23.94	0.06	24.31	0.10	24.53	0.17	24.7	0.37	24.71	0.80	24.71	1.13
Augerat et al.	22.29	0.04	23.27	0.11	23.27	0.20	23.47	0.27	23.56	0.42	23.66	0.87	23.663	1.75	23.663	2.60
Li et al.	56	528	57	628	58	724	59	820	59	1061	60	1847	60.01	4284	60.01	6426
Aver	36.6	105	38.3	126	39.1	145	39.7	164	39.9	213	40.3	371	40.3	861	40.3	1292

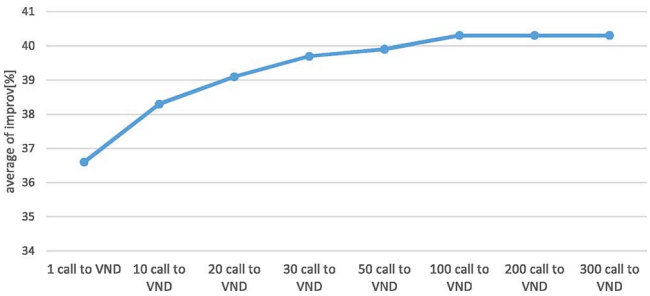

Fig. 2: Evolution of average deviation with respect to $\overline{improv}$

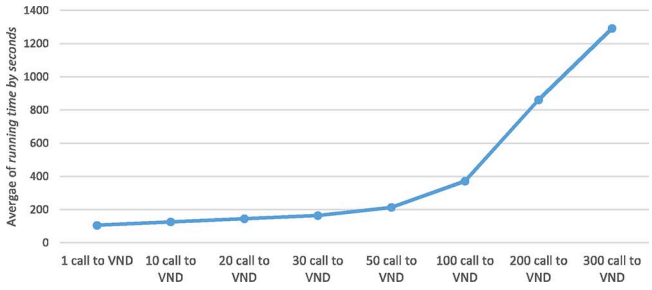

Fig. 3: Evolution of average deviation with respect to $\overline{T}$

neighborhoods' structure in VND for the problem. Extensive numerical experiments on benchmark instances containing up to 1200 customers show that our solutions for small instances fall into the range of 10.5 - 15.7% of the lower bound of the optimal solution in a reasonable amount of time. It is an important improvement for such crude lower bound. For large instances, our algorithm led to significant improvement between 23.66% and 60% for most instances and also required small computational time, even between 22.29% and 56% and an average time of 105 seconds which is only used for a single call to VND. In the future, we intend to extend our algorithm by including more neighborhoods and careful study of the effectiveness of each neighbourhood on the problem. Increasing efficiency of our algorithm to allow even larger problems to be solved, is another future research topic.

VI. Acknowledgement

This work was supported by the project "An efficiency Meta-heuristic Algorithm for the $k-$MLP" funded by Hanoi University of Science and Technology under grant number T2016-PC-037.

References

[1] A. Archer, A. Levin, and D. Williamson, "A Faster, Better Approximation Algorithm for the Minimum Latency Problem", J. SIAM, Vol. 37, No. 1, 2007, pp. 1472-1498.
[2] S. Arora, and G. Karakostas, "Approximation schemes for minimum latency problems", Proc. STOC, 1999, pp. 688-693.
[3] H.B. Ban, and D.N. Nguyen, "Improved genetic algorithm for minimum latency problem", Proc. SOICT, 2010, pp. 9-15.
[4] H.B. Ban, and D.N. Nguyen, "An Experimental Study about Efficiency of the Approximation Algorithms for Minimum Latency Problem", Proc. RIVF, 2012, pp. 230-234.
[5] H.B. Ban, K. Nguyen, M.C. Ngo, and D.N. Nguyen, "An efficient exact algorithm for Minimum Latency Problem", J. PI, No.10, 2013, pp. 1-8.
[6] H.B. Ban, K. Nguyen, M.C. Ngo, and D.N. Nguyen, "A Subgradient Method to Improve Approximation Ratio in the Minimum Latency Problem", Proc. KSE, 2013, pp. 339-350.
[7] H.B. Ban, N.D. Nghia, A hybrid algorithm combining genetic algorithm with ant colony algorithm for the Minimum Latency Problem, J. Computer Science and Cybernetics, T.29, S. 3, 2013, pp. 287-298 (in Vietnamese).
[8] H.B. Ban, N.D. Nghia, A Meta-Heuristic Algorithm Combining between Tabu and Variable Neighborhood Search for the Minimum Latency Problem, Proc. RIVF, pp.192-197, 2013.
[9] H.B. Ban, N.D. Nghia, A parallel algorithm combines genetic algorithm and ant colony algorithm for the minimum latency problem, Proc. SOICT, pp. 39-48, 2014
[10] A. Blum, P. Chalasani, D. Coppersmith, W. Pulleyblank, P. Raghavan, and M. Sudan, "The minimum latency problem", Proc. STOC, 1994, pp.163-171.
[11] K. Chaudhuri, B. Goldfrey, S. Rao, and K. Talwar, "Path, Tree and minimum latency tour", Proc. FOCS, 2003, pp. 36-45.
[12] T.A. Feo and M.G.C. Resende, "Greedy randomized adaptive search procedures", J. Global Opt., 1995, pp. 109-133,
[13] M. Goemans, and J. Kleinberg, "An improved approximation ratio for the minimum latency problem", Proc. SIAM SODA, 1996, pp. 152-158.
[14] F. Jittat, Chris Harrelson, and Satish Rao, "The k-traveling repairman problem", Proc. ACM-SIAM, pp. 655-664, 2003.
[15] N. Mladenovic, P. Hansen, "Variable neighborhood search", J. Operations Research, vol.24, No. 11 24, 1997, pp.1097-1100.
[16] A. Salehipour, K. Sorensen, P. Goos, and O.Braysy, "Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem", J. Operations Research, Vol. 9, No. 2, 2011, pp.189-209.
[17] M. Silva, A. Subramanian, T. Vidal, and L. Ochi, "A simple and effective metaheuristic for the Minimum Latency Problem", J. Operations Research, Vol 221, No. 3, 2012, pp.513-520.
[18] D. S. Johnson, and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization in Local Search in Combinatorial Optimization", E. Aarts and J. K. Lenstra, eds., pp. 215-310.
[19] B.Y. Wu, Z.-N. Huang and F.-J. Zhan, "Exact algorithms for the minimum latency problem", Inform. Proc. Letters, Vol. 92, No. 6, 2004, pp. 303-309.
[20] http://neo.lcc.uma.es/vrp/vrp-instances/
[21] https://sites.google.com/site/kminimumlatencyprolem/dataset
[22] https://sites.google.com/site/kminimumlatencyprolem/our-results
[23] https://sites.google.com/site/kminimumlatencyprolem/neighborhoods