An Efficient Ant Colony Algorithm for The Minimum Latency Problem

Rodrigo Lamblet Mafort

Instituto de Computação
Universidade Federal Fluminense
Niterói, RJ, Brazil
`rmafort@ic.uff.br`

Luiz Satoru Ochi

Instituto de Computação
Universidade Federal Fluminense
Niterói, RJ, Brazil
`satoru@ic.uff.br`

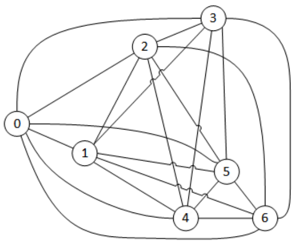
ABSTRACT

This paper presents a new efficient approach for the Minimum Latency Problem (MLP) using an ant colony metaheuristic. The MLP is a variation of the Traveling Salesman Problem which aims to find a Hamiltonian cycle in a given graph with edge weights that minimizes the latency of each vertex. The latency of a vertex corresponds to the time required to reach it from the initial vertex of the cycle, which by definition is v_0. In each iteration of the algorithm, several ants exploit the graph to identify best solutions using instance information and the pheromones associated with each edge. Moreover, the proposed algorithm uses a local search restricted to the best solutions found in each iteration. The computational results show the efficiency of the proposed algorithm when compared with the best results found in literature.

KEYWORDS. Metaheuristics, Minimum Latency Problem, Ant Colony System.

1. Introduction

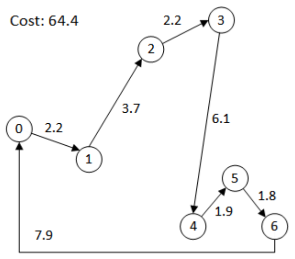
The Minimum Latency Problem (MLP) is a variant of the Traveling Salesman Problem (TSP) whose objective is to find in a given complete graph $G(V, E)$ with edge weights a Hamiltonian Cycle that minimizes the latency of each vertex. The latency of a vertex is defined as the sum of edge weights in the path between this vertex and initial vertex of the cycle, which is always v_0. This problem is also known as the Traveling Repairman Problem, Delivery Man Problem, or School Bus Driver Problem and was first addressed by Fischetti et al. [1993], Tsitsiklis [1992] and Blum et al. [1994]. As the TSP, the MLP is an NP-Hard problem. Figure 1 shows an example of an instance of this problem and a feasible solution.



(a) Graph $G(V, E)$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
v_0	0.0	2.2	4.5	6.7	5.8	6.4	7.9
v_1	2.2	0.0	3.7	5.6	3.6	4.3	5.7
v_2	4.5	3.7	0.0	2.2	5.3	4.5	6.2
v_3	6.7	5.6	2.2	0.0	6.1	4.6	6.2
v_4	5.8	3.6	5.3	6.1	0.0	1.9	2.4
v_5	6.4	4.3	4.5	4.6	1.9	0.0	1.8
v_6	7.9	5.7	6.2	6.2	2.4	1.8	0.0

(b) Matrix of Edges Weights



(c) A Feasible Solution

Figure 1: Example of an instance and a feasible solution for the MLP

The Minimum Latency Problem has real world applications when the costs of the edges correspond to the time needed to attend to each vertex, instead of the travel cost. This is interesting in problems which aim to improve the response time, e.g., support personnel and maintenance of essential services.

This paper presents a new approach for the MLP based on the concepts of an ant colony. Initially, from the moment when the ants leave the nest to locate food, each ant follows a distinct path. During the travel, each animal leaves a chemical mark, denominated pheromone. The goal of this mark is to signal the way back to the nest and guide other ants to a previously found food supply. When more ants follow this trail more pheromone it receives, becoming more attractive to further exploration.

In order to computationally implement the idea of an ant colony some changes are necessary. Beforehand, it is assumed that each ant has a memory capable of storing the traveled path. Furthermore, the pheromone application is done only when the path is complete. The amount of pheromone laid in the path is proportional to its quality. Another implemented change is how the paths are created. Instead of purely random, the computational version of ants analyses the problem data and how much pheromone is deposited in each adjacent edge. The idea of an Ant Colony inspired metaheuristic was first proposed by Dorigo and Di Caro [1999] and Dorigo et al. [1999].

The proposed solution also uses a local search method restricted to the best solutions found by the ants in each iteration. The application of this step has high impact on the final quality of the solutions.

In the literature, there are two distinct approaches for the Minimum Latency Problem. The first is based on exact algorithms and mathematical programming. This approach is interesting for small problems, however, in larger instances the time spend to get a solution is prohibitive. The algorithm proposed in [Abeledo et al., 2010a,b] can be highlighted. This algorithm is based on a branch-and-cut-and-price method and, according to [Silva et al., 2012], can solve instances up to 107 vertices. In [Abeledo et al., 2010a,b] work, the problem definition differs from the classical one. That is, the latency of the initial vertex is considered twice, at the beginning, with cost 0, and in the end of the cycle, with cost equals to the last vertex latency plus the cost of the return edge.

The second approach aims to obtain good solutions in a viable amount of time. However, no guarantees can be given about the optimality of the results. Generally, this approach consists of meta-heuristics. The first work, presented by [Salehipour et al., 2011] consists of an algorithm with two phases. The first phase corresponds to the construction of a feasible solution. The second is an improvement phase, in which a VND/VNS method is applied. The second work that needs to be highlighted was proposed by [Silva et al., 2012] and consists of an ILS algorithm, with a RVND local search method. This work comprehends the state of art for MLP.

Furthermore, this work uses an optimization technique that reduces the complexity to evaluate solutions in the local search routine. This optimization method was initially presented in [Kindervater and Savelsbergh, 1997]. Next, it was extended by [Vidal et al., 2011; Vidal, 2013] to another neighborhood structures. This technique was applied to the Minimum Latency Problem in [Silva et al., 2012].

This paper will be split in four sections. In Section 2, the proposed algorithm will be detailed. Furthermore, Section 3 presents the obtained results and some comparisons with other proposal for the MLP. Finally, Section 4 contains the conclusions of this work, with some interesting future work proposals.

2. The Proposed Solution

The proposed approach can be divided in two parts. Initially, a set of valid solutions is generated using an ant colony based algorithm. Next, the best solutions among all constructed in the previous phase are improved by an RVND method. When a better solution is found, it is stored as a solution for that instance, overwriting the previous best. This process ends when no improvement is achieved in a predeterminated number of iterations.

A solution for the MLP is a permutation of vertices of the current instance and it is stored in a n-dimensional array, where n corresponds to the size of the instance (i.e. number of vertices). The first position of this vector is always the vertex v_0, which corresponds to the begin and the end of the cycle. As mentioned before, each solution corresponds to a Hamiltonian cycle in the input graph. So, to be considered feasible, a solution must contain all vertices exactly once.

The objective of the MLP is to find among all the cycles the one that minimizes the latency of each vertex. The latency of a vertex corresponds to the sum of all edges weights in the cycle. So, the cost of a solution can be evaluated by this equation:

$$f(x) = \sum_{i=1}^{n} (n - i + 1) \times C[x_{i-1}, x_i]. \tag{1}$$

Where $C[x_{i-1}, x_i]$ corresponds to the cost of the edge between the vertices sitting in the x_{i-1} and x_i positions of the current solution. This method leads to $O(n)$ complexity for each solution evaluation.

It should be highlighted that the last edge of the cycle is not included in the latency of any vertex, so it can be omitted. However, to respect the classical definition of a Hamiltonian cycle this edge will be included in the examples. Furthermore, the restriction about the initial vertex is not common to all variations of this problem, but all of them can be converted to respect this restriction adding a vertex v_0 and n edges between v_0 and the other vertices, with null cost.

As said before, the proposed approach consists of two distinct components, merged in a single algorithm. The first phase corresponds to the creation of a set of feasible solutions by the ants. The second phase selects the best solutions from the constructed set and tries to improve each one. The best solution among all is compared with the previous best. If a new best is found, this solution is saved, overwriting the previous best, otherwise it is discarded. Considering the elevated computational cost of the improvement phase, only a limited number of solutions is improved. The parameter γ specifies how many solutions are selected for improvement, establishing a balance between the quality of final results and the spent time.

The construction and improvement phases are repeated for a predetermined number of iterations without improvements of the best solution.

Algorithm 1: The Proposed Algorithm

```
Input: $n$, $C$, $E_{rate}$, $I_{max}$, $N_{ants}$, $\alpha$, $\beta$, $\gamma$, and $\varphi$
Output: The best solution found for the current instance
1  $F$ = InitializePheromone($C$,$n$);
2  $bestCost = +\infty$;  $best = \emptyset$;  $I_{best} = 0$;
3  while $I_{best} \leq I_{max}$ do
4      $improved = False$;
5      $S = \emptyset$;
6      for $i = 1$ to $N_{ants}$ do  $S = S \bigcup$ doAntSolution($C$,$F$,$n$,$\alpha$,$\beta$, $\varphi$) ;
7      evaporatePheromone($F$,$E_{rate}$);
8      $bests$ = selectBest($S$,$\gamma$);
9      foreach $s \in bests$ do
10         $s'$ = RVND($C$,$n$,$s$);
11         if $f(s') < bestCost$ then
12             $best = s'$;
13             $bestCost = f(s')$;
14             $improved = True$;
15         end
16     end
17     if $improved = True$ then
18         $I_{best} = 1$;
19         UpdateBestPheromone($F$,$best$);
20     else
21         $I_{best} = I_{best} + 1$;
22     end
23 end
24 return $best$;
```

In order to allow different behaviors, the presented algorithm uses seven parameters. These parameters can be adjusted to prioritize the algorithm speed or the quality of the returned solution.

I_{max} Number of iterations without an improvement.

N_{ants} Size of the nest, *i.e.*, the number of ants searching the graph at each iteration.

γ Determines how many solutions are selected for the improvements phase at each iteration. The selection of these solutions is purely elitist, that is, only the γ best solutions are selected.

α *and* β Specify the balance between the pheromone and edge cost when computing the probabilities for the biased roulette in the construction of a solution by an ant.

E_{rate} Specify how much pheromone must be removed from the edges of the graph at each iteration (Evaporation Rate).

φ Define how much pheromone must be removed from each edge when an ant crosses it.

In addiction to these parameters, the algorithm also receives the current instance data: n

and C that represents respectively the number of vertices and the matrix of edge weights for the current instance.

As mentioned before, the first phase of the proposed algorithm, represented by the function *doAntSolution*, corresponds to the construction of new solutions using the concepts of an ant colony system. These concepts are usually applied to the Traveling Salesman Problem. The ant colony system presented in this paper was based on the approaches presented by [Li et al., 2008] and [Chenga and Maob, 2007], in which a predetermined number of ants is placed on the graph in such a way that each ant starts in a different vertex. In the following steps, each ant creates a cycle using the instance data and the pheromone trails to choose the next vertex of the cycle.

However, in the Minimum Latency Problem this approach is not valid considering the restriction about the initial vertex be always v_0. Considering the importance of a random component in order to create a bigger diversity of solutions, a new random component was developed respecting the problem restrictions. In this way, the aleatory component was implemented in the routine that selects the next vertex of the cycle. That is, instead of analyzing just the instance data and the pheromone trails, a random factor is also considered.

To implement this random component, a biased roulette was implemented. To each vertex that was not previous included to the cycle a determined probability is assigned. The probability of each vertex can be obtained by the following equation:

$$Prob(v) = \frac{F[uv]^{\alpha} \times (\frac{1}{C[uv]})^{\beta}}{\sum\limits_{w \in CL} F[uw]^{\alpha} \times (\frac{1}{C[uw]})^{\beta}}. \tag{2}$$

Where v e u are, respectively, the vertex to which the probability is being calculated and the last vertex included in the solution. Moreover, $F[u, v]$ and $C[u, v]$ represent the amount of pheromone deposited in the edge between vertices u and v and its cost. The variable CL contains every vertex not included in the cycle. The values of α and β are parameters of the algorithm and represent, respectively, the balance between the pheromone data and the heuristic data (e.g. cost of the edges). This procedure received the name *CreateRoulette* in Algorithm 2.

A vertex is randomly chosen from the roulette and inserted at the end of the cycle. The use of the biased roulette has guaranteed an almost random component, that is, the vertices that will most contribute to the final quality of the solution has more probability to be chosen.

When a vertex w is chosen by the roulette method, it is added to the solution that are being constructed and removed from the candidates list CL. The process of construct the roulette, choose a vertex, remove it from the candidate list and add it to the solution is repeated until the list of candidates is empty, that is, all vertices have been added to the solution, which is now a Hamiltonian cycle.

Furthermore, when a vertex is inserted in the solution, the edge that connects it to the previous vertex is crossed by the ant. Every time that an ant crosses an edge, it removes a small amount of pheromone from that edge. This reduction makes that edge less attractive to other ants, guaranteeing a bigger diversification. This step is represented in the algorithm by the function named *UpdatePathPheromone*.

The second phase of the proposed algorithm is the local search method. The objective of this routine is try to improve a solution by analyzing several neighbor solutions, that is, solutions obtained by applying a movement in the original one(e.g. a swap of two vertices). To implement a local search phase, the proposed algorithm uses a RVND method. Different from the classical VND, proposed by [Mladenovic and Hansen, 1997], the order in which neighborhoods are explored is defined at random. When a neighborhood is fully explored, two distinct cases can occur:

i. A solution of better cost is found. In this case, the algorithm will change to this best solution, restarting the RVND procedure. (Original Solution Improved)

Algorithm 2: Function doAntSolution

```
Input: Matrix of Edge Weights $C$, Matrix of Pheromones $F$, Number of Vertices $n$, $\alpha$, $\beta$, $\varphi$
Output: A valid solution for the current instance
1  $Solution = \{v_0\}$;
2  $CL = \{v_1, ..., v_{n-1}\}$;
3  $u = v_0$;
4  while $CL \neq \emptyset$ do
5      $R$ =CreateRoulette($u$,$CL$,$C$,$F$,$\alpha$,$\beta$);
6      $v$ = SelectRandom($R$);
7      $Solution = Solution + \{v\}$;
8      $CL = CL \setminus \{v\}$;
9      UpdatePathPheromone($F$,$u$,$v$,$\varphi$);
10     $u = v$;
11 end
12 return $Solution$;
```

ii. No better solution found in this neighborhood. In this case, a new neighborhood will be explored. When all neighborhoods have been explored, the algorithm will terminate the search and return the best solution found, which in the worst case is the initial solution.

Algorithm 3 differs from the classical RVND because when a change of neighborhood or a reset occurs, the order in which the neighborhoods are explored is randomly set again.

Algorithm 3: Function RVND

```
Input: $n$, $C$, Solution $s$
Output: Solution $s$ improved, if possible
1  $opt$ =CreateOptimizationStructure($s$,$C$,$n$);
2  $Neighborhoods = \{N_1, N_2, N_3, N_4, N_5\}$;
3  while $Neighborhoods \neq \emptyset$ do
4      $N_x$ =RandomChoose($Neighborhoods$);
5      $s'$ = Explore($N_x$,$s$,$opt$);
6      if $f(s') < f(s)$ then
7          $s = s'$;
8          $opt$ = CreateOptimizationStructure($s$,$C$,$n$);
9          $Neighborhoods = \{N_1, N_2, N_3, N_4, N_5\}$;
10     else
11         $Neighborhoods = Neighborhoods \setminus N_x$;
12     end
13 end
14 return $s$;
```

As demonstrated in Equation 1, the computational cost to evaluate each solution leads to $O(n)$ complexity. In order to reduce this cost, Algorithm 3 uses a technique presented by [Silva et al., 2012]. This technique consists of a concatenation operator and a proper data structure. The procedure *CreateOptimizationStructure* called in the Algorithm 3 is used to create this structure. The application of the concatenation operator with this data structure leads to a move evaluation that can be done in $O(1)$ amortized operations.

The proposed RVND method uses five neighborhoods structures:

***SWAP*:** Its work principle is based in swap of pairs of vertices in the input solution. Every combination of pairs, except those that includes v_0, is analyzed.

2-*Optimal*: This structure removes two non-adjacent edges and reconnect the two resulting paths in a such way that the cycle is different from the original and the restrictions of a Hamiltonian Cycle is preserved.

***Reinsertion*:** Examines all the possible relocations of each vertex, but v_0, in different positions of the solution.

***Or-Opt2*:** Analyses all the possibilities created by the relocation of each pair of adjacent vertices in the solution, excluding the pairs that contain v_0, in different positions inside the solution.

***Or-Opt3*:** This structure is a variation of the previous one in which instead of pairs, three adjacent vertices are analyzed.

All neighborhoods have the same complexity: $O(n^2)$, considering the use of the optimization technique. Without this technique all the neighborhoods will have $O(n^3)$ complexity, since every possible solution need to be evaluated by Equation 1. This technique uses a data structure and a concatenation operator.

The data structure is used to store all the possible subsequences obtained from the original solution. To store all of these subsequences a matrix with n by n dimensions will be used. The creation of this matrix has $O(n^2)$ complexity and uses the concepts of dynamic programming. Before presenting the algorithm that creates this matrix concatenation operator must be defined.

The concatenation operator, represented by the symbol $\oplus$, calculates the cost of a subsequence obtained by the union of two other subsequences. Considering that in the MLP in order to calculate the cost of a subsequence it is necessary know not just the cost of each edge, but the order in which they appear, this operator needs other information about the subsequences that will be concatenated. So, for each subsequence S three tributes must be considered: $T(S)$, $C(S)$ and $W(S)$ that respectively represents the sum of all edge costs contained in S, the sum of the latency of each vertex contained in S (In this sum, the costs of the edges not contained in S are ignored) and how many vertices are contained in the subsequence S, excluding v_0.

When the subsequence S contains just one vertex v, the value of this three attributes is know, considering the absence of edges. With this in mind, $T(S)$ and $C(S)$ are always zero and $W(S)$ corresponds to one when $v \neq v_0$ and zero in the other case.

Let be two subsequences of vertices $S_1 = (x_i, x_{i+1}, ..., x_j)$ and $S_2 = (x_k, x_{k+1}, ..., x_l)$. Let be S a subsequence obtained by the concatenation of S_1 and S_2. So, $S = S_1 \oplus S_2 = (x_i, x_{i+1}, ..., x_j, x_k, x_{k+1}, ..., x_l)$. Note that it is necessary add another edge to connect these two subsequences. This new edge must be added between the vertices x_j and x_k, that are, respectively, the end and the beginning of these subsequences. So, let $(x_j x_k)$ be this new edge and $C(x_j x_k)$ its cost. In order to get the cost of the resulting subsequence S, the other attributes of S must be calculated by the following equations:

$$T(S) = T(S_1) + C(x_j x_k) + T(S_2). \tag{3}$$

$$C(S) = C(S_1) + W(S_2) \times (T(S_1) + C(x_j x_k)) + C(S_2). \tag{4}$$

$$W(S) = W(S_1) + W(S_2). \tag{5}$$

This concatenation operator has permitted the application of dynamic programming, considering that in each iteration two previously calculated subsequences are concatenated creating a bigger one. Furthermore, this routine was used in the Algorithm 3 every time that the RVND method is about to process a different solution. Thus, this routine is executed exactly one time for each solution. In doing so, the complexity of the RVND method was not increased.

Algorithm 4 defines a matrix of dimensions n x n, where each cell contains the three previous mentioned attributes: T, C e W. Each position $[l, c]$ of this matrix store one subsequence of the solution S, started at lth vertex of S and finished at cth vertex of this same solution. It must be highlighted that the position $[0, n-1]$ contains the total cost of the solution S. Let *opt* be that matrix.

At each iteration of Algorithm 4, the size of the subsequences that are being calculated is increased. It starts with subsequences of size one containing only one vertex and can be initialized according to previous defined rules. The algorithm ends with one subsequence of size n containing all solution vertices. The following algorithm has $O(n^2)$ complexity.

Algorithm 4: Function CreateOptimizationStructure

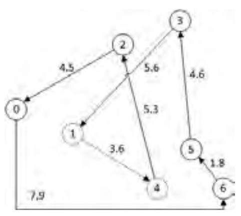
```
   Input: $n$, $C$, Solution $s$
   Output: Matrix with dimensions $n$ by $n$
1  for $i = 0$ to $n - 1$ do
2      $opt[i, i].T = 0$;
3      $opt[i, i].C = 0$;
4      if $i = 0$ then $opt[i, i].W = 0$ ;
5      else $opt[i, i].W = 1$ ;
6  end
7  for $size = 2$ to $n$ do
8      for $i = 0$ to $n - size$ do
9          $j = i + (size - 1)$;
10         $opt[i, j] = opt[i, j - 1] \oplus opt[j, j]$;
11     end
12 end
13 for $size = 2$ to $n$ do
14     for $i = size - 1$ to $n - 1$ do
15         $j = i - (size - 1)$;
16         $opt[i, j] = opt[i, i] \oplus opt[i - 1, j]$;
17     end
18 end
19 return $opt$;
```

Figure 2 presents an example of move evaluation done in $O(1)$ amortized operations. In the example, the matrix *opt* was calculated previously for the solution S.'It is worth noticing the complexity of this move evaluation technique, due to the fact that all possible moves realized by the presented neighborhoods can be evaluated in five concatenation operations, at most Silva et al. [2012]. This figure contains two solutions $S = v_0, v_6, v_5, v_3, v_1, v_4, v_2$, with cost $f(S) = 104.1$, and $S' = v_0, v_1, v_4, v_6, v_5, v_3, v_2$, with cost $f(S') = 57.6$. The solution S' was obtained from S through neighborhood $Or - Opt2$, considering the vertices v_1 and v_4. In doing so, the solution S' was formed concatenating segments of the solution S, which cost was already evaluated and is stored in *opt*.

Once the two main phases of the algorithm for the Minimum Latency Problem were presented, the pheromone manipulation routines must be formally defined.

As mentioned before, the motivation for using pheromones is to indicate which edges, when included in a solution, contributed to its quality. However, to increase the solutions diversification, it is necessary the existence of factors that reduce the level of this chemical marks, avoiding that one edge has a much higher pheromone level than others.

When the algorithm is started, the pheromone levels must be the same in every edges



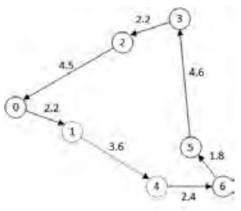
(a) Solution S

	0	1	2	3	4	5	6
0	**0**	7.9	17.6	31.9	51.8	75.3	104.1
1	0	0	1.8	8.2	20.2	35.8	56.7
2	1.8	1.8	0	4.6	14.8	28.6	47.7
3	11.0	11.0	4.6	0	5.6	14.8	29.3
4	27.8	27.8	15.8	5.6	0	**3.6**	12.5
5	42.2	42.2	26.6	12.8	3.6	0	5.3
6	68.7	68.7	47.8	28.7	14.2	5.3	0

(b) *opt* Matrix: Attribute C

	0	1	2	3	4	5	6
0	**0**	7.9	9.7	14.3	19.9	23.5	28.8
1	7.9	0	1.8	6.4	12	15.6	20.9
2	9.7	1.8	0	4.6	10.2	13.8	19.1
3	14.3	6.4	4.6	0	5.6	9.2	14.5
4	19.9	12	10.2	5.6	0	**3.6**	8.9
5	23.5	15.6	13.8	9.2	3.6	0	5.3
6	28.8	20.9	19.1	14.5	8.9	5.3	0

(c) *opt* Matrix: Attribute T



(d) Solution S'

Move: *Or-Opt2* $\boldsymbol{v_1, v_4}$
$S' = v_0, \boldsymbol{v_1}, \boldsymbol{v_4}, v_6, v_5, v_3, v_2$
$S_1 = (v_0) \oplus (v_1, v_4)$
$S_2 = (S_1) \oplus (v_6, v_5, v_3)$
$S_3 = (S_2) \oplus (v_2)$
$f(S') = S_3.C$
$f(S') = 57.6$

(e) Cost of S'

$S_1 = (v_0) \oplus (v_1, v_4)$
v_1 is 4th element of S
v_4 is 5th element of S
The subsequence (v_1, v_4) is located at *opt*$[4, 5]$

$S_1.W = 1 + 2 = 3$
$S_1.T = \mathbf{0} + C(v_0 v_1) + \mathbf{3.6} = 5.8$
$S_1.C = \mathbf{0} + 2(0 + C(v_0 v_1)) + \mathbf{3.6} = 8.0$

(f) Cost of S_1

Figure 2: Example of move evaluation: Neighborhood *Or-Opt2* and Vertices v_1 and v_4

of the graph, so, at this moment the heuristic behavior of the algorithm will prevail. To initialize the pheromone level, the algorithm evaluates the cost of solution obtained by the nearest neighbor heuristic. Let C be this cost. The level of pheromone in every edge of the graph will be initialize with $1/C$. In Algorithm 1, this method is called *InitializePheromone*. This function returns a matrix that stores the level of pheromone of each edge of the graph. Also, this initial value is stored.

Once the pheromone level of each edge is initialized, the algorithm begins the creation of feasible solutions. At each iteration, when the set of solutions is completed, the algorithm executes a method that evaporates a small amount of pheromone of each edge.

In Algorithm 1, this process is called *evaporatePheromone* and receives two parameters: F and E_{rate}, representing, respectively, the matrix that contains the level of pheromones and the evaporation rate. The evaporation method updates the level of pheromone of each edge $(v_i v_j)$ according to the following equation:

$$F[v_i v_j] = (1 - E_{rate}) \times F[v_i v_j] + (E_{rate} \times initialPheromoneValue). \tag{6}$$

The second method that deals with pheromones is responsible for removing a small amount of this chemical mark every time an ant crosses an edge. The motivation for that is to avoid other ants of using this edge in their paths, increasing the solutions diversity. This function is called in Algorithm 2 as *UpdatePathPheromone*. The parameter φ specify how much pheromone will be removed of each crossed edge. Thus, every time an ant crosses an edge $(v_i v_j)$ of the graph, the level of pheromone must be updated by the following equation:

$$F[v_i v_j] = (1 - \varphi) \times F[v_i v_j] + (\varphi \times initialPheromoneValue). \tag{7}$$

The last pheromone manipulation method, called *UpdateBestPheromone* in Algorithm 1, increases the level of this mark in the edges included in the best solution found at each iteration if, and only if, an improvement has been accomplished. This increment must be greater than the amount removed by evaporation and by any ant to ensure that this signal is going to be considered in next iterations. Considering that, the level of pheromone of each edge $(v_i v_j)$ contained in this solution which cost S_{cost}, is correct by this equation (e corresponds to the base of the natural logarithm and is generally used in ant colony metaheuristics):

$$F[v_i v_j] = F[v_i v_j] + \mathrm{e}/S_{cost}. \tag{8}$$

3. Computational Results

This section presents the best configuration found for the algorithm parameters. This configuration was determined in an empirical form, that is, multiple combinations were tested for each instance and the one who has presented the greatest balance between quality of solutions and spent time was chosen. So, the employed configuration was: $I_{max} = 2$, $N_{ants} = 50$, $\gamma = 3$, $E_{rate} = 0.25$, $\alpha = 0.9$, $\beta = 1.5$ and $\varphi = 0.25$.

The proposed algorithm was implemented in C# and was executed on an Intel^{TM} i7 2.4 GHz with 8 GB of RAM memory, running Windows 10. Only one thread was used in the tests. Furthermore, for each instance the algorithm was executed 20 times.

Considering the similarities between the MLP and the Traveling Salesman Problem, the set of classical instances for the TSP can be used as input to this problem. These instances were selected from the TSPLib [Reinelt, 1991].

There are two different ways to evaluate solutions. The first, based on the classical problem definition, considers that the latency of the vertex v_0 is always 0, that is, the cost of the solution do not include the last edge, between the last and the initial vertex of the solution. So, the solution corresponds to Hamiltonian Path. The second way of evaluating a solution includes the last edge. So, the latency of v_0 is included twice in the cost of the solution, at the beginning, with cost 0, and in the end of the cycle, with cost equals to the last vertex latency plus the cost of the return edge. The solution then is a Hamiltonian Cycle. In the comparisons presented in this paper, all solutions were evaluated using the first way.

In the following tables, the columns *Best* and *Average* represent the smaller cost and average cost among all solutions returned for each instance. The column *Time* means the average time in seconds used by the algorithm to return a solution, considering the stop condition showed in Section 2. Beside those, the column *GAP* presents the average gap between the best results found in the other papers and the best result of this paper for each instance. This way, this value can be obtained by applying the following expression $GAP = (S_2 - S1)/S_1$, where S_1 represents the best previous result and S_2, the best result obtained by this paper.

The results presented by [Salehipour et al., 2011] contains two different instance sets. The results for the first set, consisting of 10 instances selected from the TSPLib, are presented in Table 1. The second set was constructed by [Salehipour et al., 2011] and contains 140 instances of problem sizes ranging in 10, 20, 50, 100, 200, 500, 1000. Table 2 contains the results for these instances, grouped by the instance size. In this way, each table line contains the average time and the average gap of the 20 instances with size n. The gap for each instance corresponds to the gap between the nearest neighbor solution and the best result found[a]It should be highlighted that the tests presented by [Salehipour et al., 2011] were executed in a Pentium 4 with 2.4 GHz with 512 MB of RAM memory.

[a]The algorithm was executed 20 times for instances up to 200 vertices and only once above this size.

	Salehipour *et al*		Ant Colony Algorithm			
Instance	Best	Avg Time	Best	Average	Avg Time	Cost GAP
st70	19553.00	2.14	20030.68	20411.97	0.57	2.38%
rat99	56994.00	13.98	**56989.57**	**57809.65**	**1.88**	**-0.01%**
kroD100	976830.00	4.69	**951731.45**	**984153.58**	**1.57**	**-2.64%**
lin105	585823.00	11.25	587015.15	594785.60	1.82	0.20%
pr107	1983475.00	39.44	1984540.47	2004629.31	1.74	0.05%
rat195	213371.00	104.38	221239.43	226040.78	10.73	3.56%
pr226	7226554.00	228.78	**7117374.39**	**7185368.02**	**12.41**	**-1.53%**
lin318	5876537.00	401.06	**5670374.32**	**5857527.31**	**45.05**	**-3.64%**
pr439	18567170.00	508.84	**18128690.01**	**18627114.28**	**145.95**	**-2.42%**
att532	18448435.00	5134.36	**5761449.00**	**5898476.90**	**228.91**	**-220.20%**
Average						-2.21%

Table 1: Results for TSPLIB instances selected by [Salehipour et al., 2011].

Instance		Salehipour *et al* - NoFix-Full		Ant Colony Algorithm	
Name	Size	GAP (UB)	Time	GAP (UB)	Time
TRP-S10	10	-2.44%	0.00	**-2.56%**	**0.08**
TRP-S20	20	-9.86%	0.04	-9.54%	0.10
TRP-S50	50	-9.67%	3.21	**-10.40%**	**0.28**
TRP-S100	100	-11.56%	101.27	**-11.91%**	**1.40**
TRP-S200	200	-11.33%	3741.05	**-11.92%**	**9.25**
TRP-S500	500	-8.11%	91470.78	**-9.57%**	**205.91**
TRP-S1000	1000	-	-	-9.04%	2075.65
Average		-8.83%	15886.06	-9.31%	36.17

Table 2: Results for random instances created by [Salehipour et al., 2011].

A comparison between the results of this paper and the obtained by [Silva et al., 2012] can be found in Table 3. These results are the best known in the literature.

	Silva *et al*		Ant Colony Algorithm				
Instance	Best	Avg Time	Best	Average	Avg Time	Cost GAP	Time GAP
st70	19215.00	1.51	20030.68	20411.97	0.57	4.07%	-62.34%
rat99	54984.00	9.70	56989.57	57809.65	1.88	3.52%	-80.67%
kroD100	949594.00	6.90	951731.45	984153.58	1.57	0.22%	-77.22%
lin105	585823.00	6.19	587015.15	594785.60	1.82	0.20%	-70.56%
pr107	1980767.00	8.13	1984540.47	2004629.31	1.74	0.19%	-78.65%
rat195	210191.00	75.56	221239.43	226040.78	10.73	4.99%	-85.79%
pr226	7100308.00	59.05	7117374.39	7185368.02	12.41	0.24%	-78.98%
lin318	5560679.00	220.59	5670374.32	5857527.31	45.05	1.93%	-79.58%
pr439	17688561.00	553.74	18128690.01	18627114.28	145.95	2.43%	-73.64%
att532	5581240.00	1792.61	5761449.00	5898476.90	228.91	3.13%	-87.23%
Average						1.90%	-83.52%

Table 3: Results for TSPLIB instances selected by [Silva et al., 2012].

4. Conclusions

This paper presented a new approach to the Minimum Latency Problem that uses the idea of an ant colony with a local search method. In all presented tests, it is noticed that the application of this new algorithm reduces the spent time to get a satisfactory solution.

During this work, it became clear that the application of an ant colony system have conducted to a greater diversity of initial solutions, but it is also clear that the isolated application of this technique do not present a satisfactory solution.

As future work, two interesting modifications can be done. The first is the parallelization of this algorithm, considering the clearly independence of each ant in solution creation. The second consists in the implementation of other local search neighborhoods with a perturbation phase.

References

Abeledo, H., Fukasawa, R., Pessoa, A., and Uchoa, E. (2010a). The time dependent traveling salesman problem: Polyhedra and algorithm. Technical Report RPEP Volume 10, No 15, RPEP, Universidade Federal Fluminense, Brasil.

Abeledo, H., Fukasawa, R., Pessoa, A., and Uchoa, E. (2010b). The time dependent traveling salesman problem: polyhedra and branch-cut-and-price algorithm. *Proceedings of the 9th International Symposium on Experimental Algorithms*, page 202–203.

Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P., and Sudan, M. (1994). On the minimum latency problem. *Proc 26thSymp Theory of Computing STOC*, page 9.

Chenga, C. and Maob, C. (2007). A modified ant colony system for solving the travelling salesman problem with time windows. *Mathematical and Computer Modelling*, 46:1225–1235.

Dorigo, M., Caro, G. D., and Gambardella, L. M. (1999). Ant Algorithms for Discrete Optimization. *Artificial Life*, 5(2):137–172.

Dorigo, M. and Di Caro, G. (1999). Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477.

Fischetti, M., Laporte, G., and Martello, S. (1993). The Delivery Man Problem and Cumulative Matroids. *Operations Research*, 41(6):1055–1064.

Kindervater, G. and Savelsbergh, M. (1997). Vehicle routing: handling edge exchanges. In Aarts, E. and Lenstra, J., editors, *Local Search in Combinatorial Optimization*, page 337–360. Wiley.

Li, B., Wang, L., and Song, W. (2008). Ant colony optimization for the traveling salesman problem based on ants with memory. *Fourth International Conference on Natural Computation*.

Mladenovic, N. and Hansen, P. (1997). Variable neighborhood search. *Computers Operations Research*, 24:1097–1100.

Reinelt, G. (1991). TSPLIB—A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384.

Salehipour, A., Sörensen, K., Goos, P., and Bräysy, O. (2011). Efficient grasp + vnd and grasp + vns metaheuristics for the traveling repairman problem. *4OR: A Quarterly Journal of Operations Research*, 9:189–209.

Silva, M., Subramanian, A., Vidal, T., and Ochi, L. (2012). A simple and effective metaheuristic for the minimum latency problem. *European Journal of Operational Research*, 221:513–520.

Tsitsiklis, J. N. (1992). Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3):263–282.

Vidal, T. (2013). *Approches générales de résolution pour les problèmes multi-attributs de tournées de véhicules et confection d'horaires*. Ph.d., Université de Montréal.

Vidal, T., Crainic, T., Gendreau, M., and Prins, C. (2011). A unifying view on timing problems and algorithms. Technical report, CIRRELT.