# MEMORY OPTIMIZED PHEROMONE STRUCTURES FOR MAX-MIN ANT SYSTEM

*Marek Běhálek*[*][†], *Martin Šurkovský*[†], *Ondřej Meca*[*][†], *Stanislav Böhm*[*]

**Abstract:** Ant Colony Optimization is a meta-heuristic for solving hard combinatorial optimization problems. It is a constructive population based approach inspired by the social behavior of ants. In our research, we are focused on the parallel/distributed computing on massively parallel systems. More precisely, we want to adjust Max-Min Ant System (one of Ant Colony Optimization algorithms) for these systems. Traditionally, a matrix is used to store the pheromone information. If we want to solve large instances, this is a very memory consuming solution. In this paper, we propose a different approach. We do not use the matrix to store the pheromone information. Instead, ant trails that are normally incorporated into this matrix are stored during the computation and just some parts and only in time when they are really needed are assembled. Proposed solution was implemented in C++. The implemented solution was tested on large symmetric instances of Traveling Salesman Problem. In these experiments, we were able to compute results with a comparable quality and even faster than with the traditional approach while using only a portion of the original memory.

Key words: *Max-Min Ant System, memory optimized pheromone structures, distributed memory systems, C++*

*Received: October 6, 2014* **DOI:** 10.14311/NNW.2015.25.008
*Revised and accepted: April 8, 2015*

## 1. Introduction

Ant Colony Optimization (ACO) [7] is a class of meta-heuristic algorithms for solving hard combinatorial optimization problems. It is a constructive population based approach inspired by the social behavior of ants. It was successfully used to solve different problems. Traveling Salesman Problem (TSP) [7, Chapter 3] can be used as an example of such problem. TSP is one of the well-known NP-hard problems. It seems that there is no exact method to compute a solution of such problems in a reasonable amount of time. Still, computer scientists and

[*]Marek Běhálek – Corresponding Author, Ondřej Meca, Stanislav Böhm, IT4Innovations National Supercomputing Center, VŠB – Technical University of Ostrava, Studentská 6231/1B, Ostrava, Czech Republic, Phone: +420 597 325 879, E-mail: `marek.behalek@vsb.cz, ondrej.meca@vsb.cz, stanislav.bohm@vsb.cz`

[†]Marek Běhálek, Martin Šurkovský, Ondřej Meca, Department of Computer Science FEECS, VŠB – Technical University of Ostrava, 17.listopadu 15, Ostrava, Czech Republic, Email: `martin.surkovsky@vsb.cz`

programmers frequently encounter such problems and they are often addressed by using heuristic methods or approximation algorithms.

In our research, we are focused on the parallel/distributed computing on massively parallel systems. We want to use ACO to solve large problem instances on supercomputers. Using ACO, we can get good results in a reasonable time, still there are some obstacles. The quality of a solution strongly depends on many hard to predict factors. First, randomness is a crucial part of ACO. Hence, different runs can give different results. Second, the quality of a solution strongly depends on initial parameters. We do not want to invent a new ACO variant with different guidelines how to use it, but rather adjust one of existing algorithms. For various reasons we have chosen Max-Min Ant System (see Section 2).

There are various approaches how to implement ACO algorithms for parallel/distributed systems (a recent survey is in [2]). They address different issues like granularity or a topology of communication, but still the resulting implementations are usually tested and used only on relatively small systems with tens of processors. Moreover, they frequently use a shared memory. Our long term goal is to adjust ACO for distributed memory systems with thousands of processors.

As an initial step, we address an issue how to efficiently store the pheromone information. Traditionally, a matrix is used to store this data. This matrix serves as a shared medium where artificial ants share their data. As a result, every ant needs it to construct a solution in every iteration. Using the pheromone matrix is a very memory consuming solution, if we want to solve large instances. For example, if we use the largest problem from TSPLIB [11] (it is a library of sample instances including the symmetric TSP) with 85900 cities, then the pheromone matrix size will be (C++ type `double` is used for elements in the matrix): $85900 \times 85900 \times 8 Bytes \approx 59 GB$. This may be too much, considering the pheromone matrix is not the only information that we need to store. For example Anselm [10] (a supercomputer of National Supercomputing center IT4Inovations) has nodes with 64GB of a physical memory. Furthermore, it is shared by 16 computing cores. Also, we need to update the pheromone matrix after each iteration and the amount of data that we need to transmit between nodes can be an issue.

In this paper, we propose a different approach. We do not use the pheromone matrix. Instead, we store the data that are incorporated into this matrix during the computation and later assemble just some parts and only for moments, when they are really needed. Thus, we can emulate the original matrix using only a!portion of the original memory. In addition, because it is smaller, it is easier to distribute it among computing nodes.

The proposed solution is implemented in programming language C++. It was tested on symmetric instances of TSP form TSPLIB (see Section 5). In performed experiments, we were able to compute results with a similar quality in even better time as with the traditional approach while using only a portion of the original memory.

This article is structured as follows. In the following section, Max-Min Ant System and reasons why we use this ACO algorithm are introduced. Then our Disassembled Pheromone Matrix is described. Section 5 summarizes performed experiments. The last section contains a conclusion and open problems.

## 2. ACO algorithms

There are different ACO algorithms, but they all roughly follow these steps:

```
int iteration=0;
Init() //Set parameters, initialize pheromone trails
while (termination condition not met) {
  ConstructAntsSolutions()
  ApplyLocalSearch() // optional
  EvaporatePheromones()
  UpdatePheromones(Ant* chosenAnts)
  iteration++;
}
```

In most ACO algorithms, there is a fixed number of ants. Each of these ants constructs a tour between given cities. The tour construction process is performed in each iteration. Each ant is initially placed in a randomly chosen city. Then it chooses from unvisited cities the best target to travel to. This decision is based on a combination of heuristic information (computed using distances between cities in TSP) and the pheromone information. This step is repeated until all cities are visited. After all ants build their tours, then some chosen ants deposit their pheromone information. It guides ants in following iterations to potentially better tours. Usually the global best ant or the iteration best ant is chosen. This process repeats, until termination conditions are met.

Even if ACO algorithms usually follow this schema, they differ in some details. The first ACO algorithm is called Ant System and it was introduced by Dorigo in 1992. Since then, a lot of new ACO algorithms were introduced. In [7, Chapter 3], the following ACO algorithms that are able to solve TSP are mentioned: *Ant System, Elitist Ant System, Ant-Q, Ant Colony System* [6], *Max–Min Ant System* [13] *and Rank-based Ant System.* Furthermore, new algorithms are still frequently introduced.

For our task, we have chosen Max-Min Ant System (MMAS) [13]. It is one of the standard ACO algorithms. It incorporates main mechanisms and memory structures that are common to most ACO algorithms. Also, it is one of the most successful ACO algorithms. Following properties are interesting for our research.

- It does not modify the pheromone information during the tour construction (unlike Ant Colony System [6]). So, the tours may be easily constructed in parallel.
- In every iteration, only one ant deposits pheromones into the pheromone matrix (unlike for example Elitist Ant System).
- It defines minimum and maximum for pheromone values. So, we know boundaries for values in the pheromone matrix.

To improve computational results, diversification mechanisms based on a pheromone reinitialization are used. There are different strategies, how to implement this process. The strategy used in MMAS is: If more than 250 iterations have been executed without the reinitialization of pheromone trails and no improved solution has been found in last 25 iterations, we reinitialize the pheromone information. For

MMAS, we initialize pheromone trails between all cities to the current maximum, like at the beginning of the computation, but we keep the best solution founded so far.

When faced with large problems (especially for TSP), MMAS uses some local optimizations. Usually, it implements a candidate list to reduce a number of possible cities for the next step in the tour construction. For each city, the candidate list contains a given number of nearest neighbors sorted in increasing order. Ants choose cities from the candidate list until all immediate nearest neighbors are visited. Only then they are allowed to pick a city outside the list.

Finally, MMAS may be augmented with a local search procedure [1]. Usually, *2-opt*, *2.5-opt* or *3-opt* is used. Local optimizations greatly improve a solution's quality in ACO algorithms. They are almost mandatory, if we want to use these algorithms for large TSP instances.

## 2.1 Operations with the pheromone matrix in MMAS

As was mentioned before, the matrix is usually used to store the pheromone information. The number of rows and columns corresponds to the number of cities for TSP. We will name this matrix $\mathbf{T}$ and we will use $\tau_{x,y}$ to address an element on $x$-th row and $y$-th column. As was mentioned in the previous section, in MMAS, lower limit $\tau_{\min}$ and upper limit $\tau_{\max}$ are imposed on possible pheromone values on any arc. For MMAS with the local search, the following values are suggested in [13].

- $\tau_{\max} = 1/(\rho \cdot C^{\text{best}})$, where $C^{\text{best}}$ is the length of the tour of the best ant founded so far and $\rho$ is the parameter for evaporation.
- $\tau_{\min} = \tau_{\max}/(2 \cdot N)$, where $N$ is the number of cities.

While $\tau_{\min}$ and $\tau_{\max}$ depend on the best solution, they are updated whenever a new best solution is found. For MMAS with the local search, suggested $\rho$ is 0.2.

We do the following operations with the pheromone matrix in MMAS.

1. *Initialization and reinitialization* – at the beginning and after the reinitialization all values in the pheromone matrix are set to $\tau_{\max}$.
2. *Pheromone evaporation* – all values in the pheromone matrix are multiplied by $1 - \rho$.
3. *Update pheromones* – in MMAS, only one ant deposits its' trail. The pheromone deposit $\tau$ is computed as $\tau = 1/C$, where $C$ is the length the trail. If the ant went from $x$ to $y$ in iteration $i$, then $\tau^{i}_{x,y} = \tau^{i-1}_{x,y} + \tau$. Moreover, the same pheromone deposit is also added to the arc form $y$ to $x$. So, the resulting pheromone matrix $\mathbf{T}$ is symmetric.

## 3. Disassembled pheromone matrix

If we look on the pheromone matrix in MMAS and how it changes in time, we can observe that a lot of its cells contain the same value.

Let $\tau_{\text{global}}$ (a global pheromone deposit) in iteration $i$ be:

$$\tau_{\text{global}} = \begin{cases} \tau_{\max} \cdot (1-\rho)^i & \text{if } \tau_{\max} \cdot (1-\rho)^i > \tau_{\min} \\ \tau_{\min} & \text{otherwise} \end{cases}.$$

Hence, $i$ is the number of iterations from the last restart or if there was no restart, from the beginning of the computation. In other words, $\tau_{\text{global}}$ is the pheromone deposit in pheromone matrix cells, where no ant deposited its pheromones.

In the first experiment, three tests were executed with three different inputs. For every test, 1000 iterations were computed. We counted the number of cells that were equal to $\tau_{\text{global}}$. For input[1] *eil51* it was in average 97.85% of entries, for *rat783* 99.78% and for *pr2392* 99.93%. These results can be expected, if we realize that the number of unique ant trails is limited.

At this point, very inspiring for us was paper [8]. In the paper, a new ACO algorithm named Population Based ACO (PACO) is introduced. The author's motivation is different than ours. They used the new ACO algorithm to solve dynamic optimization problems where the input instance changes during the computation [9]. In PACO, the pheromone matrix is still used, but no pheromone evaporation is done. There is a queue where ant trails are stored and there is a limit for the queue size. Whenever we want to deposit an ant trail into the pheromone matrix, we add this ant trail into the queue and deposit its pheromone information into the pheromone matrix. If the queue exceeds the given limit, the oldest ant trail is removed. Also, its pheromone deposit is subtracted from the pheromone matrix. From our point of view, the interesting result is that PACO gives comparable results even if the number of ants in the queue is small (in the paper experiments, the length five is used).

In our work, we tried to use a similar approach, but we want to use MMAS and get results as close as possible to an original solution, where the pheromone matrix is straightforwardly represented as an array.

First, the issue how to store ant trails was addressed. ACO algorithms naturally produce the trail like a sequence of cities. This is inappropriate for our task. We need to get a target city from a given starting point quickly. That is why ant trails are represented like an array, where at a position $x$ is a city index where the ant went from city $x$. Moreover, if an ant traversed from $x$ to $y$, it also boosts the pheromone information from $y$ to $x$. Hence, trails in the backward direction are also stored.

For example, if the trail is: $0, 1, 3, 2, 0$ then it is represented as two arrays: $[1, 3, 0, 2]$ and $[2, 0, 3, 1]$. Note, that 0 at the index 2 in the first array defines that the ant went from city 2 to city 0.

Also, the ant pheromone deposit is computed and stored. It is implemented as class `AntTrail`.

```
class AntTrail {
  // ...
  double pheromoneDeposit;

  long int* forwardPath;
```

[1]These inputs were taken from TSPLIB. The number in the name corresponds to the number of cities.

```
  long int* backwardPath;
  // ...
};
```

In the following subsections, three variants of Disassembled Pheromone Matrices are introduced (*Fixed Size DPM, Dynamic Size DPM, Improved DPM*). They differ in a precision of emulating the original pheromone matrix. Later, a precise solution (Sparse Pheromone Matrix) is introduced.

The abstract class `Pheromones` was introduced and all our solutions extend this class. It has virtual methods that correspond to basic operations for the pheromone matrix (they were described in Section 2.1).

```
class Pheromones {
  // ...
  virtual void initPheromoneTrails()=0;
  virtual double get(long int i, long int j) =0;
  virtual void evaporate(void) =0;
  virtual void updatePheromone(Ant &ant) =0;
  // ...
};
```

Thus during experiments, the algorithm was the same. We just changed the type of the instance that emulates the pheromone matrix.

## 3.1 Fixed size DPM

First, we tried to implement the same pattern like the authors of PACO. We used a queue with a fixed size – `ants`. It is implemented as an array of pointers to ant trails. In attribute `globalDeposit`, $\tau_{\text{global}}$ is stored. Virtual methods (pheromone matrix operations) are implemented as follows:

- `initPheromoneTrails` – it simply clears ant trails and `globalDeposit` is set to $\tau_{\max}$.
- `evaporate` – `globalDeposit` and pheromone deposits of ant trails are multiplied by $1 - \rho$.
- `updatePheromone` – a new ant's trail is stored in the queue and the oldest one is removed, if the queue current size exceeds the limit.
- `get` – this function returns the pheromone deposit on an arc between cities $x$ and $y$.

```
double FixedSizeDPM::get(long int x, long int y) {
  double result = globalDeposit;

  for (int i = 0; i < current_queue_size; i++) {
    AntTrail trail = ants[i];
    if (trail.contains(x,y) || trail.contains(y,x)) {
          result += trail.pheromoneDeposit;
    }
  }

  if (result < trail_min) result = trail_min;
  if (result > trail_max) result = trail_max;

  return result;
}
```

Methods `initPheromoneTrails` and `evaporate` are very fast, even faster than if we use the matrix (see Section 5). The method `updatePheromone` is comparable with its matrix equivalent. But the bottleneck in speed is method `get` and this method is used frequently during the tour construction. To improve its performance, we have to look at the tour construction algorithm. As was mentioned in Section 2, the elementary step is that an ant searches for its next step from the current location. The pheromone information for arcs connecting this city with other cities or its nearest neighbors can be precomputed.

This can be done by a slight modification of method `get`. In this modification, an array with the size equal to the number of cities is used instead of `result`. Then similar operations like before are performed, but for all immediate neighbors. To compute this data, the array of ants needs to be processed, but just once. Then this buffer is used during the computation. We need to go through the list of immediate neighbors twice (the initialization and the checking of $\tau_{\max}$ and $\tau_{\min}$), but MMAS algorithm goes through the list of immediate neighbors as well to choose a next step. So the resulting implementation of the tour construction is slower, but only by a small constant factor.

## 3.2 Dynamic size DPM

*Fixed Size DPM* introduces a new parameter to MMAS – the length of the queue. Like other ACO parameters, it is hard to predict what the optimal value is. Moreover, MMAS after 250 iterations boosts only the global best solution. At this point, there will be only multiple copies of the same trail in the list. So, our second implementation extends the previous one, but uses a list with a dynamic size. Then ant trails are removed, if their pheromone deposits become "*insignificant*". Moreover, when we recognize that some ant trail is already present in the list, we just increase its pheromone deposit and we do not add the second copy to the list.

In our experiments, we remove trails when their current deposit is smaller than 10%, 1%, 0.01% or 0.00001% of their original deposit.

## 3.3 Improved DPM

A possible improvement without an additional memory, is to remove only selected arcs in a trail.

Assume, that we have two ant trails: $1, 2, 3, 4, 1$ and $1, 2, 4, 3, 1$. At some point in time, none of those ants alone is able to boosts the pheromone information above $\tau_{\min}$ after the evaporation. But if they trails are combined together (arcs: (1,2), (3,4)), then it is above $\tau_{\min}$. If the pheromone matrix is used, then the deposit of such arcs is in fact removed and $\tau_{\min}$ is used in the matrix instead. These evaporated arcs do not affect the solution in further steps. This is not true for our previous solutions. The pheromone deposit is reduced to $\tau_{\min}$, if it is bellow $\tau_{\min}$, but these evaporated arcs are still there and they are used.

It is relatively easy to detect the described situation and it can be solved by removing evaporated arcs from stored ant trails. During the process, it is possible to count the number of removed arcs and remove only ant trails with no arcs. But from a practical point of view, this is not a good solution. Some short arcs are used by almost all ants. Pheromones on these arcs never evaporate, so consecutively, trails that use them are never removed.

That is the reason, why the same approximation as before is used. Ant trails are removed, if their pheromone deposits are smaller than 10%, 1%, 0.01% or 0.00001% of their original deposit.

### 3.4 $\tau_{\max}$ reduction

Even if we remove ant trails based on evaporated arcs, the emulation is not precise. Similarly to the lower bound, there is a problem with the upper bound. If the computed pheromone information on some arc exceeds $\tau_{\max}$, then contributing arcs from stored ant trails should be reduced to give together $\tau_{\max}$. To implement such reduction, we need to be able to modify the pheromone deposit for each of arcs in stored ant trails. A possible solution can be an additional array, where we store something like pheromone reduction coefficients. The better solution that precisely emulates the pheromone matrix is introduced in Section 4. So, for Dynamic Size DPM and Improved DPM, we have solved just one, the most frequent situation when the pheromone deposit exceeds $\tau_{\max}$. That does not require additional memory.

After some number of iterations, only so far the best ant's trail is updated. Its update is bigger than its evaporation. Hence, this pheromone deposit grows in time. If it exceeds $\tau_{\max}$ during the computation, it is reduced to $\tau_{\max}$. We will refer to this extension as $\tau_{\max}$ reduction.

## 4. Sparse pheromone matrix

The last pheromone structure implements a similar idea like a sparse matrix. We want to store only cells where values are affected by some ant. While previous pheromone structures only imprecisely emulate the pheromone matrix, Sparse Pheromone Matrix (SPM) precisely represents all values from the original pheromone matrix. The basic idea is, that we store only values that are different from $\tau_{\text{global}}$. From previous experiments, we assume that there is only limited number of such cells.

Let $a^j$ is the pheromone deposit of a chosen ant from iteration $j$. Then $a^j_{x,y}$ is the pheromone deposit on the arc between cities $x$ and $y$. It is defined as

$$a^j_{x,y} = \begin{cases} a^j & \text{if the ant went from } x \text{ to } y\text{, or from } y \text{ to } x; \\ 0 & \text{otherwise.} \end{cases}$$

Let $i$ denote a number of iterations from a last restart or from the beginning. The pheromone deposit $\tau^i_{x,y}$ between cities $x$ and $y$ is computed as

$$\tau^i_{x,y} \quad = \quad \tau_{\max} \cdot (1-\rho)^i + \sum_{j=1}^{i} a^j_{x,y} \cdot (1-\rho)^{i-j}.$$

Now, it is sufficient to compute $\sum_{j=0}^{i} a_{x,y}^{j} \cdot (1-\rho)^{i-j}$ for every arc to precisely reconstruct the pheromone matrix. It is important to mention, that most of arcs are never visited. The value for these arcs is zero and thus no value is stored. To store nonzero values in SPM, a hash map is used for each city. We have used a hash map in order to search values in logarithmic time.

To get the precise emulation of the original pheromone matrix, we need to enforce $\tau_{\max}$ and $\tau_{\min}$ boundaries. The pheromone deposits grow only when some ant deposits its pheromone trail. During the update, we can simply check $\tau_{\max}$ constraint and if the current value exceeds $\tau_{\max}$, it is reduced to $\tau_{\max}$ (similarly as in Section 3.4). For $\tau_{\min}$, we check every computed $\tau_{x,y}^{i}$ and if it is bellow $\tau_{\min}$, then $\tau_{\min}$ is used instead. Moreover, all values that contributed to this computation, can be safely removed while they evaporated bellow $\tau_{\min}$ (as in Sectoin 3.3).

In addition, $(1-\rho)^{i}$ is not computed in every iteration. The value from the previous iteration is used and it is multiplied by $(1-\rho)$.

## 5. Experiments

The implemented solutions were tested on large *symmetric* instances of TSP from TSPLIB. The second frequently used matrix is a distance matrix. It contains distances between cities. Its size is similar to the size of the pheromone matrix. But for example for asymmetric TSP, it is a part of the problem description and different techniques must be used to address the issue with its size. Potentially, we can divide it and distribute it among computing nodes, but this enforce us to use some kind of a data parallelism. For example in [5], they used such technique to solve some platform dependent issues while implementing ACO on GPUs. For large distributed memory systems, we have not found a variant that implements a real data parallelism. Such task may be very challenging. When we use symmetric instances, we can store only cities positions and compute distances when they are needed. Still, data parallelism of ACO for distributed memory systems is an interesting task for a future research.

Our implementation is written in C++ using object oriented programming and it closely follows guidelines from the book [7]. We were also guided by the software package *ACOTSP* [12]. It is freely available GNU General Public License. In fact, we have used some parts (mainly functions implementing the local search) in our code. We want to parallelize it using Kaira [3] (a tool that we are developing) in the future. The implemented source codes are freely available at Kaira's homepage [4].

There is a randomness integrated into every ACO algorithm. Thus, results can be different even if the setting is the same. Unless stated otherwise, experiments were executed ten times and presented results are average values obtained during these executions. The initial MMAS parameters were set to their default values as suggested in [7] (number of ants = 25, number of nearest neighbors = 20, $\alpha = 1$, $\beta = 2$, $\rho = 0.2$, *3-opt* for local search).

In our first experiment, the average and maximum number of trails for Dynamic Size DPM and Improved DPM were measured. For SPM, the average and maximum number of stored cells were measured. As the input, *pr2392* from TSPLIB was used and 2000 iterations were computed. The results of the first experiments summarize Tab. I.

| Matrix Type | Number of trails | |
|---|---|---|
| (when a trail become insignificant) | Average | Maximum |
| Improved DPM (0.00001%) | 17.49 | 64 |
| Improved DPM (0.01%) | 11.33 | 38 |
| Improved DPM (1%) | 6.00 | 21 |
| Improved DPM (10%) | 3.71 | 11 |
| Dynamic Size DPM (0.00001%) | 23.45 | 63 |
| Dynamic Size DPM (0.01%) | 14.27 | 38 |
| Dynamic Size DPM (1%) | 6.08 | 21 |
| Dynamic Size DPM (10%) | 3.94 | 11 |
| | Number of cells | |
| | Average | Maximum |
| Sparse Pheromone Matrix | 3031 | 5440 |

**Tab. I** *The average number of trails and cells for pr2392.*

Based on the first experiment, we can observe that the number of trails remains very small and it does not depend on the number of iterations. The amount of used memory depends on the number of stored trails. Thus, the needed memory grows linearly with the number of cities, while for the matrix, this grow is quadratic.

In the next experiment, we focus on a quality of computed solutions. Three instances from TSPLIB with an increasing size were chosen and 2000 iterations were performed in every configuration. A length of the best path and a time needed to compute these iterations were measured. Tab. II summarizes obtained results.

As we can see, all implemented structures become faster than the traditional matrix solution for larger instances. The tour construction is slower, the pheromone update is very similar, but evaporating and the matrix initialization are much faster in our approaches. Their complexity (considering the number of cities as the input length) is in $\Theta(1)$ or $\Theta(n)$ for the initialization of SPM, while for the matrix it is in $\Theta(n^2)$. This operation makes the difference for larger instances. Also as can be expect, the solutions that use more ant trails are slower.

Interesting is also the quality of founded solutions. They all ended up with a similar solution quality (considering the overall improvement). Versions emulating the pheromone matrix less precisely were sometimes even better than the traditional matrix variant. But for some stronger conclusions we performed too few experiments. Some local optimizations may be for example driving the convergence for the less precise solutions or the reason is the suboptimal initial setting.

Based on these experiments, we can conclude that removing ant trails when their pheromone deposit is smaller than 0.01% of their original deposit is a good strategy, while it produced reasonable results in a good time. But the optimal value for this parameter can be different especially for instances with different sizes. If we want to incorporate a problem size into this parameter, we can use for example $\tau_{\min}$.

The pheromone deposit of an ant is computed as $\tau = 1/C$, where $C$ is the trail length. Trails are removed if their current deposit is 0.01% of their original deposit. Thus, its value is $0.0001 \cdot 1/C$. Based on definitions from Section 2.1 can be observed:

| Name<br>Optimal solution | **rat783**<br>8806 | | **pcb1173**<br>56892 | | **pr2392**<br>378032 | |
|---|---|---|---|---|---|---|
| | Length | Time (s) | Length | Time (s) | Length | Time (s) |
| **Pheromone matrix** | **8811** | **36.13** | **56927** | **63.00** | **379122** | **155.68** |
| Improved DPM (0.00001%) | 8810 | 42.34 | 56915 | 67.07 | 378981 | 143.34 |
| Improved DPM (0.01%) | 8808 | 40.20 | 56922 | 61.82 | 379093 | 139.30 |
| Improved DPM (1%) | 8809 | 39.01 | 56947 | 61.32 | 378974 | 137.66 |
| Improved DPM (10%) | 8809 | 37.42 | 56934 | 59.20 | 378938 | 130.10 |
| Dynamic Size DPM (0.00001%) | 8808 | 42.52 | 56927 | 66.32 | 379203 | 145.93 |
| Dynamic Size DPM (0.01%) | 8808 | 41.10 | 56899 | 63.78 | 378842 | 133.56 |
| Dynamic Size DPM (1%) | 8809 | 38.55 | 56907 | 62.04 | 378942 | 135.63 |
| Dynamic Size DPM (10%) | 8807 | 37.77 | 56945 | 60.48 | 379000 | 129.18 |
| Fixed Size (50) | 8807 | 46.61 | 56949 | 72,55 | 379507 | 157.09 |
| Fixed Size (25) | 8807 | 42.24 | 56925 | 66.29 | 378966 | 136.57 |
| Fixed Size (10) | 8810 | 38.97 | 56966 | 60.65 | 379032 | 135.37 |
| Fixed Size (5) | 8810 | 37.30 | 56949 | 57.20 | 379184 | 130.54 |
| Sparse Pheromone Matrix | 8807 | 41.82 | 56934 | 62.48 | 379178 | 140.17 |
| *Following experiments use $\tau_{max}$ reduction.* | | | | | | |
| Improved DPM (0.00001%) | 8807 | 43.16 | 56931 | 66.42 | 379127 | 148.57 |
| Improved DPM (0.01%) | 8809 | 40.76 | 56919 | 63.64 | 378595 | 140.90 |
| Improved DPM (1%) | 8809 | 38.83 | 56928 | 61.77 | 378869 | 138.96 |
| Improved DPM (10%) | 8810 | 38.26 | 56919 | 60.55 | 378795 | 135.25 |
| Dynamic Size DPM (0.00001%) | 8810 | 41.75 | 56897 | 67.90 | 379035 | 143.35 |
| Dynamic Size DPM (0.01%) | 8807 | 44.55 | 56911 | 64.56 | 378785 | 135.67 |
| Dynamic Size DPM (1%) | 8811 | 38.79 | 56932 | 61.15 | 379172 | 136.66 |
| Dynamic Size DPM (10%) | 8806 | 38.32 | 56973 | 59.92 | 378791 | 132.94 |

**Tab. II** *Results given by implemented pheromone structures and their performance.*

$$\tau_{\max} \cdot \rho = \frac{1}{C^{\text{best}}}$$

$$\tau_{\max} = 2 \cdot N \cdot \tau_{\min}$$

Considering $C$ in equations above is approximately the same as $C^{\text{best}}$ then

$$0.0001 \cdot 1/C = \frac{1}{10000 \cdot C} \approx \frac{\rho \cdot \tau_{\max}}{10000} = \frac{\rho \cdot 2 \cdot N}{10000} \cdot \tau_{\min}$$

For tested instances, the trails were removed when their deposit was approximately from 3% of $\tau_{\min}$ (for *rat783*) to 9% of $\tau_{\min}$ (for *pr2392*). Further on, the limit 5% of $\tau_{\min}$ is used for trails removing. Also, the $\tau_{\max}$ reduction improved the quality of results. So, it is used.

In our last experiments, two large instances from TSPLIB were solved. Not only results are presented, but also the progress of computations. While it is a computationally demanding task, just one run for each test was performed. In the first experiment, input usa13509 was used for selected pheromone structures. in the experiment, imprecise solutions are compared to the original matrix. During the experiment, 10 000 iterations were computed. Fig. 5 summarizes the results. All implementations ended up with similar paths lengths. The progress during the computation was only a slightly different. The computation took 32 646s for Dynamic Size DPM, 34 708s for Improved DPM, and 41 403s for the matrix.

Finally, the fastest solution – Dynamic Size DPM was used to compute 3 000 iterations for the largest input from TSPLIB – pla85900. It took approximately 30 hours and Fig. 5 captures the progress of the computation.

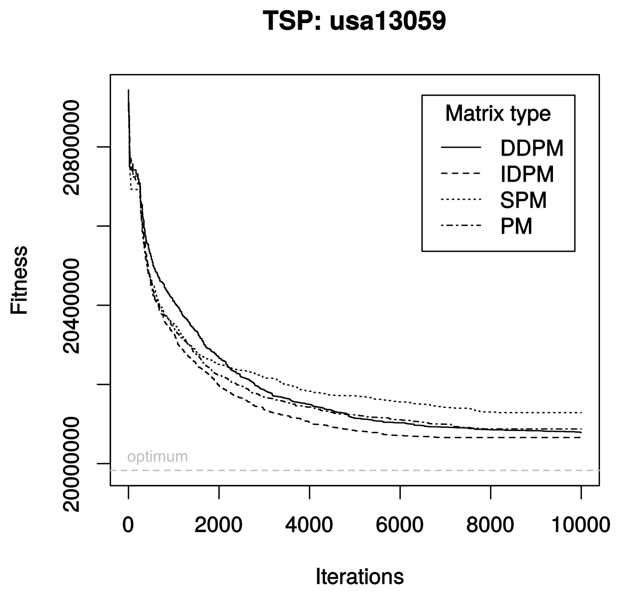

**Fig. 1** *The progress of computations for implemented pheromone structures for input usa13509.*

The last two experiments, the solutions were far from the optimum. We believe that the reason is initial settings of parameters. They were based on experiments on relatively small instances [13] and they seem inappropriate for large instances. This can also be an interesting problem for further research.

## 6. Conclusion and open problems

In this paper, we explored the idea, if we really need to store the pheromone information in a matrix. Using the matrix is the most common solution for ACO algorithms. For larger instances, such matrix can be huge. Moreover, only a limited number of cells contain relevant data. The paper presents a different option. For MMAS, only ant trails and some additional information are stored. They are used

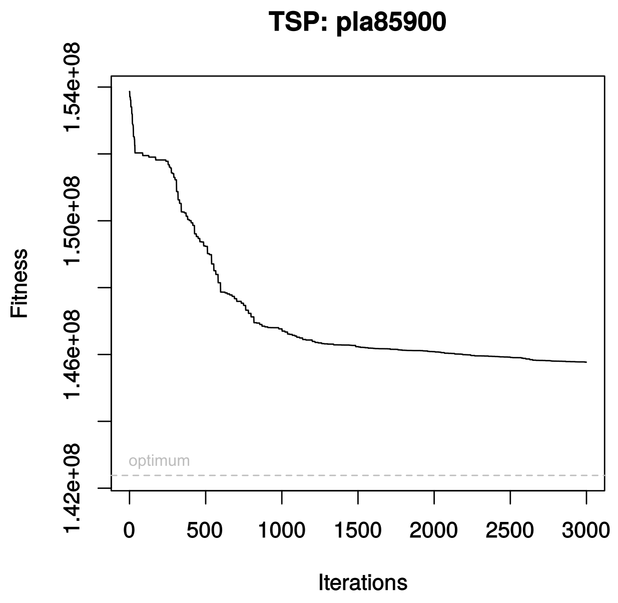

**Fig. 2** *The progress of computations for the largest problem from TSPLIB – pla85900.*

to compose only necessary values of the pheromone matrix and only when they are really needed during the computation.

We do not argue about theoretical aspects, we simply follow the guidelines for MMMAS from [7] and various options how to implement this idea are presented. Results of practical experiments show that presented structures can successfully emulate the original pheromone matrix and they give reasonable results while using a considerably less memory. For problem instances with thousands of cities, only tens of ant trails need to be stored to get similar results. Moreover, our solution becomes even faster with an increasing problem's size.

Based on our experiments, we can conclude that Dynamic Size DPM with $\tau_{\max}$ reduction and 0.5% of $\tau_{\min}$ as a limit and Sparse Pheromone Matrix are the best candidates to store the pheromone information. For Dynamic Size DPM it is easy to determine (or limit) the needed memory. It produces reasonable results and it was the fastest variant. Sparse Pheromone Matrix is the only variant precisely emulating the original matrix. Moreover, it is even faster for large instances. The needed memory depends on an input instance and may be different for different executions.

Our motivation for this task was the adjustment of MMAS for massively parallel systems with the distributed memory. For the parallelization in the distributed memory environment, a huge pheromone matrix presents an issue, because it is not easy to distribute and update such a large structure. Also an interesting topic for future research can be a fact, that during the experiments our disassembled solutions gave sometimes even better results than the original pheromone matrix.

## Acknowledgement

This paper has been elaborated in the framework of the project New creative teams in priorities of scientific research, reg. no. CZ.1.07/2.3.00/30.0055, supported by Operational Programme Education for Competitiveness and co-financed by the European Social Fund and the state budget of the Czech Republic. The work is also supported by the European Regional Development Fund in the IT4Innovations Center of Excellence project (CZ.1.05/1.1.00/02.0070) and by the Technology Agency of the Czech Republic under the Project TE01020197.

## References

[1] AARTS E., LENSTRA J. *Local Search in Combinatorial Optimization.* New Jersey: Princeton University Press, 2003.

[2] ALBA E., LUQUE G., NESMACHNOW S. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research.* 2013, 20(1), pp. 1–48, doi: 10.1111/j.1475-3995.2012.00862.x.

[3] BÖHM S., BĚHÁLEK M., MECA O., ŠURKOVSKÝ M. Kaira: Development environment for MPI applications. In: *Proceedings of the 35th International Conference on Application and Theory of Petri Nets and Concurrency*, Tunis, Tunisia. Springer International Publishing, 2014, pp. 385–394, doi: 10.1007/978-3-319-07734-5_22.

[4] BÖHM S., BĚHÁLEK M., MECA O., ŠURKOVSKÝ M. Kaira 1.2 [software]. 2014-06-21 [accessed 2014-10-07]. Available from: `http://verif.cs.vsb.cz/kaira/`

[5] CECILIA J.M., GARCÍA J.M., NISBET A., AMOS M., UJALDÓN M. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing.* 2013, 73(1), pp. 42–51, doi: 10.1016/j.jpdc.2012.01.002.

[6] DORIGO M., GAMBARDELLA L. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation.* 1997, 1(1), pp. 53–66, doi: 10.1109/4235.585892.

[7] DORIGO M., STÜTZLE T. *Ant Colony Optimization.* Bradford Company, MA: MIT Press, USA, 2004.

[8] GUNTSCH M., MIDDEMDORF M. A population based approach for ACO. In: *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN*, Kinsale, Ireland. Berlin-Heidelberg: Springer, 2002, pp. 72–81, doi: 10.1007/3-540-46004-7_8.

[9] GUNTSCH M., MIDDEMDORF M. Applying population based ACO to dynamic optimization problems. In: *Proceedings of the 3rd International Workshop Ant Algorithms (ANTS 2002)*, Brussels, Belgium. Berlin-Heidelberg: Springer, 2002, pp. 111–122, doi: 10.1007/3-540-45724-0_10.

[10] IT4Inovations. Anselm Cluster Documentation. In: *IT4Inovations#Docs* [online]. IT4Inovations, Ostrava [viewed 2014-10-07]. Available from: `https://docs.it4i.cz/anselm-cluster-documentation/compute-nodes`

[11] REINELT G. Symmetric traveling salesman problem. In *TSPLIB* [online]. Universität Heidelberg, Germany [viewed 2014-10-07]. Available from: `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

[12] STÜTZLE T. ACOTSP 1.03 [software]. 2004-03-01 [accessed 2014-10-07]. Available from: `http://www.aco-metaheuristic.org/aco-code/public-software.html`

[13] STÜTZLE T., HOOS H. Max–min ant system. *Future Generation Computer Systems.* 2000, 16(8), pp. 889–914, doi: 10.1016/S0167-739X(00)00043-1.