Scaling Techniques for Parallel Ant Colony Optimization on Large Problem Instances

Joshua Peake
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
J.Peake@mmu.ac.uk

Martyn Amos
Department of Computer and Information Sciences
Northumbria University
Newcastle upon Tyne, United Kingdom
martyn.amos@northumbria.ac.uk

Paraskevas Yiapanis
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
P.Yiapanis@mmu.ac.uk

Huw Lloyd
Centre for Advanced Computational Science
Manchester Metropolitan University
Manchester, United Kingdom
Huw.Lloyd@mmu.ac.uk

ABSTRACT

Ant Colony Optimization (ACO) is a nature-inspired optimization metaheuristic which has been successfully applied to a wide range of different problems. However, a significant limiting factor in terms of its scalability is memory complexity; in many problems, the *pheromone matrix* which encodes trails left by ants grows quadratically with the instance size. For very large instances, this memory requirement is a limiting factor, making ACO an impractical technique. In this paper we propose a restricted variant of the pheromone matrix with linear memory complexity, which stores pheromone values only for members of a candidate set of next moves. We also evaluate two selection methods for moves outside the candidate set. Using a combination of these techniques we achieve, in a reasonable time, the best solution qualities recorded by ACO on the *Art TSP* Traveling Salesman Problem instances, and the first evaluation of a parallel implementation of $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System on instances of this scale ($\geq 10^5$ vertices). We find that, although ACO cannot yet achieve the solutions found by state-of-the-art genetic algorithms, we rapidly find approximate solutions within $1-2\%$ of the best known.

ACM Reference Format:
Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. 2019. Scaling Techniques for Parallel Ant Colony Optimization on Large Problem Instances. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3321707.3321832

1 INTRODUCTION

Ant Colony Optimization (ACO) [15] is a population-based optimization technique based on the foraging behaviour of ants [12, 13]. The technique represents ants as software agents that traverse a problem space and construct multiple solutions. Ants allocate "pheromone" to each component of a good solution, and this signal concentration is used by following ants to inform decisions. Over time, this process of positive feedback causes the ant population to converge to a high-quality solution.

Multiple ACO variants have been developed; these are often specifically designed to perform more efficiently on certain problems, or with certain hardware in mind. We focus on one of these, $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [30], due to its established good performance in terms of *parallelization*. $\mathcal{MM}$AS differs from the original ACO, known as Ant System [16], in two main ways. Firstly, maximum and minimum pheromone levels are enforced in order to limit the effects of a phenomenon known as *stagnation*. Secondly, the act of pheromone distribution is restricted to only the *best performing* ant, as opposed to Ant System and other variants, which allow all ants to distribute pheromone.

Parallelization of the ACO algorithm is a well-researched area, due to the inherently distributed nature of the technique. While early parallel ACO techniques largely made use of distributed systems [5, 8, 14, 26, 28, 34] , more recent work has investigated use of GPUs, with Nvidia's CUDA framework [6, 7, 27] and Intel's range of manycore CPUs, Xeon Phi [19, 25, 32, 33] receiving particular attention.

The Travelling Salesman Problem (TSP) was the first problem used to demonstrate ACO, and is still commonly used for benchmarking new techniques. While ACO is capable of finding good quality solutions for TSP instances of varying sizes, it has not been used on instances larger than a few tens of thousands of cities. This is due to its reliance on a pheromone matrix, the data structure containing pheromone levels for (in this example) each pair of cities. The size of this matrix grows quadratically with the instance size. Assuming that a pheromone level is stored as a 32-bit float, a TSP instance of size 10,000 requires a matrix occupying around 380MB, which can be easily handled by most modern hardware. However, for a 100,000 city TSP, approximately 37GB is required, which is much less practical. In order to allow ACO to effectively solve these large-scale instances, we need to make fundamental changes to the ACO data structure. Previous work in this area has

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321832

focused on adopting a population-based ACO approach [9, 17]. In this paper, we investigate an combination of alternative techniques which allow us to use ACO to effectively solve large-scale TSPs.

While reducing the size of the pheromone matrix is a significant step towards increasing the practicality of ACO for very large problems, the use of *candidate sets* is also crucial for reducing execution time [10]. These restrict the number of options available to an ant at any time step to a pre-determined number of nearest neighbours. This significantly reduces processing time without impacting on solution quality.

In this paper, we demonstrate the the effectiveness of combining candidate sets with a reduced pheromone matrix, by restricting the matrix to each city's group of *nearest neighbours* (as opposed to all pairs of cities). The fundamental underlying assumption is that high quality solutions to the TSP generally avoid long-range jumps between cities. This restriction allows our ACO method to solve, to near optimality, TSP instances that are significantly larger than those previously solved using this method, without compromising the basic principles of ACO. Our principal contributions are: (1) a scalable method for pheromone matrix representation with linear memory complexity, based on a candidate set approach, (2) two alternative fallback techniques for choosing edges outside of the candidate set, and (3) the first evaluation of $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System on large ($> 10^5$ city) TSP instances.

The rest of the paper is organized as follows: in Section 2 we describe the background to our method and related work, and in Section 3 we describe our new methods. We give the results of experimental investigations in Section 4, before concluding in Section 5 with an assessment of our method, and a consideration of how it may be more broadly applied.

2 BACKGROUND AND RELATED WORK

Previous work on improving the efficiency of ACO may be partitioned into three main areas of focus: (1) parallelization, (2) candidate sets, and (3) pheromone matrix reduction. Most existing work has concentrated on the first two areas; here, we focus on the third. However, we first give a brief overview of relevant aspects of ACO parallelization.

A fundamental component of any ACO algorithm is *selection* of the next solution component (e.g., the next edge to traverse) for each individual ant. This is done probabilistically, according to both pheromone concentrations and any local rules associated with the problem. Roulette Wheel selection is traditionally used by ants to choose their next edge, with each edge receiving a "slice" of the roulette wheel that is proportionate to its "weight", a parameter determined by a combination of pheromone level and distance. While edges with a higher weight have a higher chance of being selected, it is still possible for the ant to travel to any city that hasn't yet been visited. This technique is straightforward to implement sequentially, but is difficult to parallelize.

The Independent Roulette (I-Roulette) [6] technique was a significant development in parallel ACO on GPU, as it substituted the traditional Roulette Wheel (i.e. fitness proportionate) method of edge selection with a data-parallel approach. An alternative method, Double-Spin Roulette (DSRoulette) [11], aimed to preserve the exact proportionality of the original roulette (unlike I-Roulette, in which the proportional relationship between probability and edge weights is lost).

I-Roulette was later adapted to make use of the vectorization potential provided by Intel's Xeon Phi manycore co-processor, via its Vector Processing Unit (VPU) and IMCI vector instructions. This vectorized version of I-Roulette, known as *vRoulette-1* [19], enabled I-Roulette to be used on many-core SIMD (Single Instruction, Multiple Data) architectures such as Intel Xeon Phi. A vectorized version of DSRoulette, *vRoulette-2*, also performed better than the original implementation. Similarly vectorized I-Roulette implementations, UV-Roulette [33] and I-Roulette v2 [24], have been developed, as well as a vectorized implementation of the traditional Roulette Wheel approach [24].

We now consider the second technique for improving ACO efficiency. *Candidate sets* are widely used with ACO to reduce computation time by only allowing ants to select from a pre-determined number of their nearest neighbours. While vRoulette-1 made use of candidate sets, the focus was on improving solution quality by ensuring ants only visited nearby cities rather than on reducing execution time. The Vectorized Candidate Set Selection technique (VCSS) [25] focused on using candidate sets to improve execution time by introducing a Nearest Neighbour object to the ACO algorithm. Designed to take advantage of the AVX512 instruction set, which replaced IMCI in the Knight's Landing generation of Xeon Phi, VCSS operates two separate selection methods: a candidate set roulette (*CSRoulette*) and a fallback method. Both methods are very similar to vRoulette-1, with the only difference being that CSRoulette only selects from available nearest neighbour edges rather than selecting from every available edge. If no nearest neighbour edges are available the fallback method is used, which is identical to vRoulette-1. VCSS showed a significant speedup over vRoulette-1, and more details of the technique are given in Section 2.2.

In this paper, we also focus on the memory complexity of ACO, thus addressing the third highlighted opportunity for improvement. The baseline memory requirement for a pheromone matrix on a problem with n vertices is $O(n^2)$, which becomes prohibitive (on current hardware) for solving instances with $\sim 10^5$ vertices or larger. One previous attempt to overcome this restriction is Population-based ACO (P-ACO) [17], although this was motivated by a need to solve *dynamic* problems, rather than very large problems *per se*. P-ACO removes the pheromone matrix entirely, replacing it with a population of good tours that are deleted once they reach a certain "age". Rather than using pheromone in decision making, ants consult the population of good tours when selecting the next city to visit. P-ACO inspired the PartialACO technique [9], which instead replaced the pheromone matrix with *local memory* for each ant (storing the best tour found by that ant). PartialACO also represents a radical departure from the traditional ACO tour construction phase, by having each ant change only part of a good *previous* tour, rather than producing a new tour at every iteration. At the start of an iteration, an ant selects a starting city and a number of cities to retain from the local best tour. The PartialACO technique enabled the first recorded results for ACO on four of the well-known *Art TSPs*, six very large TSP instances ranging from 100,000 to 200,000 cities. PartialACO found tours that were within 7% of the best known, in times ranging from around 1 hour to around 7.4 hours. However, although this technique performs well on very large TSP

instances, we will demonstrate that it is still possible to achieve improved solution qualities whilst retaining the core features of the traditional ACO algorithm.

In the rest of this Section we give an overview of the $\mathcal{MM}$AS variant of ACO, which forms the basis of our work, and provide more details of the VCSS technique.

2.1 $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System

The ACO algorithm for the Travelling Salesman Problem may be divided into two main phases: (1) tour construction, and (2) pheromone update. During the tour construction phase, each of the m ants randomly selects a starting city, and moves across the graph to gradually build a tour. At each iteration, an ant uses both pheromone concentration and Euclidean distance between cities to make a probabilistic selection of the next city to visit. The probability of ant k at city i choosing to move to city j is given by:

$$p_{i,j}^k = \begin{cases} \frac{[\tau_{i,j}]^\alpha [\eta_{i,j}]^\beta}{\sum_{i \in N_i^k} [\tau_{i,j}]^\alpha [\eta_{i,j}]^\beta} & i \in N_i^k \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Here, $\eta_{i,j} = 1/d_{i,j}$ where $d_{i,j}$ is the length of edge (i, j), $\tau_{i,j}$ is the pheromone value for edge (i, j) and N_i^k is the *feasible region* for i. The feasible region (the list of cities that ant k is able to visit) is derived from a *tabu list* structure containing a list of cities *already* visited by the ant (ants may not revisit cities on the tabu list).

Once an ant has visited each city once, it returns to the starting city. The ant then begins the *pheromone update* stage. The $\mathcal{MM}$AS update phase differs from other ACO variants in two ways: (1) only the *global-best* or *iteration-best* ant deposits pheromone, rather than every ant; and (2) pheromone is *clamped* between a minimum and maximum bound (hence the name of the method) in order to reduce the possibility of *stagnation*. The first difference has significant implications for our own work, as restricting pheromone deposition to a single ant makes the technique amenable to parallelization (since there is no need for multiple write access to the pheromone matrix). In general, the amount of pheromone deposited is proportional to the quality of the solution: in the case of the TSP, the amount is inversely proportional to the tour length, since shorter tours are better. The pheromone is deposited according to:

$$\tau_{i,j} = \tau_{i,j} + \Delta\tau_{i,j} \forall (i, j) \in L \quad (2)$$

where L is the set of edges in the complete graph and $\Delta\tau_{i,j}$ is the amount of pheromone deposited on edge (i, j), given by

$$\Delta\tau_{i,j} = \begin{cases} 1/C & \text{if edge}(i, j) \in T \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where T is the set of edges in the iteration-best or best-so-far tour, and C is the total length of this tour. Once pheromone has been distributed, the next step is *pheromone evaporation*, during which the global pheromone is reduced by a constant fraction, allowing sub-optimal solutions to be "forgotten" over time. The pheromone is evaporated using the rule

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} \forall (i, j) \in L \quad (4)$$

where $\rho \in [0, 1]$ controls the evaporation rate.

In $\mathcal{MM}$AS pheromone values in are "clamped" between two limits, τ_{min} and τ_{max}, which are defined by

$$\tau\text{max} = \frac{1}{pC_{\text{best}}}; \tau\text{min} = \tau\text{max} \frac{2(1 - a)}{a(n_{\text{neighbours}} + 1)} \quad (5)$$

where $n_{\text{neighbours}}$ is the number of nearest neighbours and $a = \exp(\log(0.05)/n)$.

2.2 Vectorized Candidate Set Selection

As previously noted, while the traditional *Roulette Wheel* selection method performs well for serial implementations of ACO, it is a difficult technique to parallelize. Although several parallel alternatives exist, our selection method is based on Independent Roulette (I-Roulette) [6]. Here, the weight of each edge available to an ant is multiplied by a uniform random deviate between 0 and 1, and the edge with the highest product of weight and random number is selected. While higher-weighted edges are more likely to be selected than lower-weighted edges, the selection probabilities are not directly proportional to weight, and I-Roulette selection is greedier than the standard roulette wheel [20].

The I-Roulette technique was later vectorized as vRoulette-1 [19] which maintains the fundamentals of I-Roulette, but makes use of vectorization provided by the Xeon Phi. Weights and random numbers are loaded into 16-wide vectors and multiplied simultaneously using the IMCI instruction set. A *highest weights* vector is maintained, storing the highest weight from each lane of the vector. Once every weight has been multiplied, the highest weight vector is reduced, with the result of this being the highest value throughout the entire process. The city associated with this value becomes the next to be visited.

vRoulette-1 was improved further with the development of the Vectorized Candidate Set Selection (VCSS) [25] technique. The significant difference between the two is the use of a new candidate set structure in VCSS. An array of nearest neighbour objects, each of which contains the index of one or more nearest neighbours, is associated with each city. When an ant moves between cities, the nearest neighbour object array of the current city is loaded directly into a modified vRoulette-1 which performs the same actions as the standard method, but which operates only on nearest neighbours rather than on every possible vertex. If no nearest neighbour cities are available, the standard vRoulette-1 is performed as a fallback. To the best of our knowledge, VCSS is the best-performing parallel implementation of ACO, and we therefore use it as the basis of the work presented here.

3 RESTRICTED PHEROMONE MATRIX

Removing or significantly adapting the pheromone matrix is an important and necessary step towards establishing ACO as an effective solution for very large problems. Previous work on P-ACO [17] and PartialACO [9] focused on removing the pheromone matrix entirely, relying instead on a population of solutions. The key contribution of the current paper is the creation of a new, candidate-set based memory structure, the *Restricted Pheromone Matrix*, to reduce the memory complexity of ACO from quadratic to linear in instance size, thus allowing large problem instances to be solved in a reasonable time. This data structure stores only the weights

Table 1: Data requirements for Pheromone Matrix and Restricted Pheromone Matrix on various TSP sizes, with a nearest neighbour list size of 32.

Instance Size	Pheromone Matrix	Restricted Matrix
100	39 KB	12.5 KB
1000	3.8 MB	125 KB
10,000	381.5 MB	1.22 MB
100,000	37.3 GB	12.2 MB

between the current vertex and its nearest neighbours, as well as other vertices stored in the nearest neighbour structure for efficient vectorization. If n_{NN} is the number of nearest neighbours, n is the number of vertices and v is the vector size available on our hardware, the restricted pheromone matrix requires $n \times n_{NN} \times v$ real numbers, compared to n^2 for the full pheromone matrix. This significantly reduces the memory requirements of ACO, especially on very large instances, as demonstrated in Table 1. For a 100,000 city TSP instance, the restricted matrix occupies only 0.26% of the space required by the standard pheromone matrix.

In order to accelerate the calculation, we also store a *distance matrix* for vertices represented in the pheromone matrix. The *edgeDist* matrix stores the distances between each vertex used for the weight calculation, and requires the same amount of memory as the restricted pheromone matrix. For a constant vector width and nearest-neighbour list size, the memory complexity of the proposed algorithm is therefore $O(n)$.

3.1 Tour Construction

The tour construction phase is parallelized using OpenMP, with each ant being allocated to an available thread. No synchronization is required, as ants write only to local memory during a tour, and global memory is only written to once per iteration, when all ants have completed their tours. Each ant selects a starting vertex randomly, and then repeatedly calls the edge selection function. The first stage of the edge selection is similar to VCSS, with the only difference being that weights are directly loaded (rather than having to check a nearest neighbour data structure to look up indices in the matrices), since the pheromone and distance matrices only contain nearest neighbours. The process of applying the nearest neighbour mask to obtain a vector of valid weights is shown in Figure 1.

Once this vector of valid weights has been filled, the tabu mask is then applied in order to filter out any cities that have already been visited. The weights are then multiplied by a vector of random numbers between 0 and 1. The randomized weights are then compared with the running maximum weights vector on a lane-by-lane basis, with larger values in the current weights vector replacing values in their line in the maximum weights vector. This process is repeated until all the vectors of weights in the nearest neighbour list have been considered. We then perform a reduction on the maximum weights vector to find the highest overall weight. The index associated with this weight is then used as the index of the next visited city. At this point, it is possible that no city is selected, if all the cities in the nearest neighbour list are tabu; in this case,

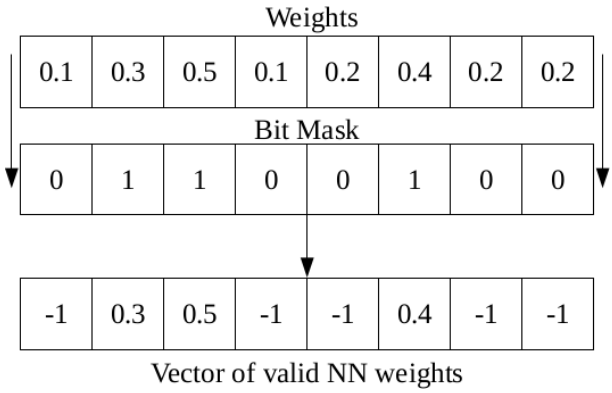

Figure 1: Applying the NN mask to filter out non-NN weights

one of the two "fallback" methods described in Sections 3.2 and 3.3 is used to select the next city.

The process continues until every city has been visited, at which point the ant returns to the starting city. Further details about the VCSS technique can be found in [25]; while VCSS falls back to v-Roulette1 when no nearest neighbour vertices are available, here we propose two alternative fallback methods.

3.2 Heuristic Fallback

In standard ACO, the highest-weighted vertex is usually chosen when all nearest neighbours are tabu. When using the restricted pheromone matrix, however, no pheromone is available for vertices outside the nearest neighbour list. The first fallback algorithm we propose is to select the *nearest vertex not yet visited*. Since the pre-computed distance matrix also extends only to the nearest neighbour list, the distances must be directly computed from the vertex coordinates. To avoid having to perform a square root calculation, we therefore look for the vertex with the lowest squared distance to the current vertex.

3.3 Pheromone Map Fallback

The Pheromone Map Fallback method aims to faithfully reproduce the $\mathcal{MM}$AS algorithm by ensuring that all edges make use of a varying level of pheromone (not just the nearest neighbour edges), but without compromising on memory requirements. We make use of a C++ map object (an associative array), which stores data in key-value pairs. This stores a pheromone value for every edge that forms part of a best ant's tour and which is not a nearest neighbour edge (a hash map has previously been used to replace the pheromone matrix [3]).

The key for map entries objects is a hash value that uniquely identifies one edge, and the value is the weight of that edge. The hash value is calculated with the simple formula of $(A \times N) + B$, where A is the current vertex, B is the next vertex, and N is the overall number of vertices (A and B are swapped if B is a higher index than A).

Since this fallback is used only when all nearest neighbours are tabu, we may assume that if an edge is found in the map then it has an associated pheromone value, otherwise the pheromone value is taken as τ_{min}. Each vertex is iterated over, and the hash value

corresponding to the edge is looked up in the map. The edge weight is computed using the pheromone and Euclidean distance, and compared with the current highest weight, becoming the highest weight if it is greater. After iterating over all vertices, the vertex associated with the overall highest weight is visited next.

3.4 Pheromone Distribution

The pheromone distribution phase of the algorithm differs depending on the fallback method that is used in the tour construction phase. If the Heuristic fallback is used, pheromone levels on edges between nearest neighbours are adjusted. Edges traversed by the best ant in the current iteration have their pheromone levels increased by an amount determined by the pheromone deposit formula given in Section 2.1. However, as pheromone is not stored for non-nearest-neighbour values, no pheromone is deposited on those edges. While pheromone value is stored for certain non-nearest neighbour vertices that are in the NN object of NN values, these weights are never actively used, so their pheromone is not updated. Pheromone reduction, as well as clamping between maximum and minimum values, takes place after the pheromone has been deposited.

The Pheromone Map fallback pheromone distribution phase includes the steps taken when using the Heuristic fallback, but includes an additional step. If pheromone is to be distributed on an edge where at least one vertex is a non-nearest-neighbour value, a new entry is created in the pheromone map. If the hash already exists in the map, the associated pheromone value is increased, but if it does not exist, a new map entry is created with the hash as the key. As with the Restricted Pheromone Matrix, the map is iterated over, and every value in the map is evaporated and clamped.

3.5 Local Search

Variants of the local search [2] technique have been successfully paired with ACO implementations on multiple occasions [9, 15, 22]. Local search is used with ACO to improve completed tours by finding the local optimum with respect to some neighbourhood (2-opt, 2.5-opt or 3-opt). The 3-opt operator removes three edges in a tour, and evaluates the seven possible ways of reconnecting the tour. If any of these seven possibilities lead to a shorter tour distance, the original three edges are replaced with the new optimum configuration, and this process is repeated until no further improvement is found. Here, we use the 3-opt local search code from ACOTSP [29], and apply this operator to all tours created in an iteration. The local search phase is parallelized across the threads owned by the ants; each ant performs local search on its own thread at the end of tour construction.

4 EXPERIMENTAL RESULTS

In this Section, we present the results of experiments to evaluate the two proposed methods, and compare the results of the better-performing of the two with the published results for PartialACO and P-ACO, which are the only other ACO methods in the literature which have been applied to large-scale TSP instances. We compare our results on solution quality with published results using P-ACO and PartialACO and, although this is not a direct comparison since the original runs used different hardware, these published results represent the best solutions found to date using ACO on these large instances. Experiments on the Heuristic and Pheromone Map fallbacks were run on a machine with an Intel® Xeon E5-2640 v2 processor with 20 cores of 2 threads each (for a total of 40 threads), and a clock speed of 2.4 GHz. The code was compiled using the GNU C++ compiler (g++), with *O2* optimization enabled.

Table 2: Solution quality and mean execution time results for Heuristic (HF) and Pheromone Map (PMF) fallbacks over 10 runs each of 1000 iterations on the `mona-lisa100k` instance. Solution quality is measured as the percentage difference of tour length from best known.

	Solution Quality (%)				
Method	Min	Median	Mean	Max	t/hrs
HF	1.684	1.704	1.698	1.712	1.07
PMF	1.689	1.7	1.7	1.709	5.15

4.1 ACO Parameters and Problem Instances

For each experiment, we use 40 ants. Conveniently, this number is equal to both the number of threads we have available, and the generally recommended number of ants [21]. We use the $\mathcal{MM}$AS parameter values of $\alpha = 1, \beta = 2, \rho = 0.02$. Each ant has a Nearest Neighbour list of size 32, in line with the recommended list size in [31]. Each run of our algorithm consists of 1000 iterations.

The problem instances used in our experiments are taken from the well-known Art TSP collection [1] of Traveling Salesman Problem instances. We used all six of the instances, which are shown in Figure 2. We compare our results with the best-known tour for each of these instances. All of the best known solutions were found using a genetic algorithm with Edge-Assembly Crossover (EAX) [18].

4.2 Fallback Comparison

Our first experiment was performed to determine which of our two fallback methods performs best, and to evaluate whether or not the use of the heuristic fallback (which disregards the pheromone on edges outside the candidate set) has a detrimental effect on *solution quality*.

We carried out 10 runs of 1000 iterations with each fallback method, using the `mona-lisa100k` instance. The results are given in Table 2. We find that the solution qualities for both fallback methods are consistent with each other, and within each ensemble of runs; in all cases the tours found are around 1.7% longer than the best known. A *Wilcoxon signed-rank* test on the two sets of solution qualities gives a p value of 0.959, indicating that our data cannot support the conclusion that one fallback produces a better solution quality on average. However, the Heuristic fallback constructs tours in significantly shorter time, with the runs taking on average around an hour, compared to around 5 hours for the Pheromone Map fallback. The extra overhead in querying the pheromone map dominates the time to solution in this case.

Figure 3 shows the mean memory consumption of the pheromone map as a function of iteration. Although this grows steadily, the

Figure 2: Best known tours for the Art TSP instances: `mona-lisa100k` (top left), `vangogh120k` (top right), `venus140k` (middle left), `pareja160k` (middle right), `courbet180k` (bottom left) and `earring200k` (bottom right).

map consumes a relatively small part of the overall memory budget for pheromone data (less than 1 MB out of a total of 13 MB).

4.3 Solution Quality

While we see no significant difference between the tour lengths for either fallback method, the difference in execution time makes the Heuristic fallback a much more practical method for evaluating our restricted pheromone matrix on the five larger Art TSP instances.

For each instance we performed ten runs of 1000 iterations. We compare our solutions with those found by PartialACO and P-ACO [9], where these exist. Our technique produces solutions that are approximately 1-2% over the shortest recorded tours for these instances, which is a significantly smaller difference than P-ACO and PartialACO (see Figure 4 for a comparison). It is difficult to directly compare solution *times* due to hardware differences, and the fact that the PartialACO technique does not create full tours for each iteration, but, for completeness, a comparison of execution times is given in Table 3. We can at least say that these are broadly

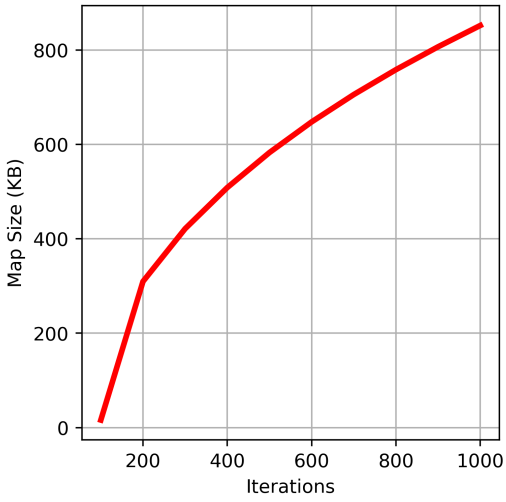

Figure 3: Pheromone map size over time

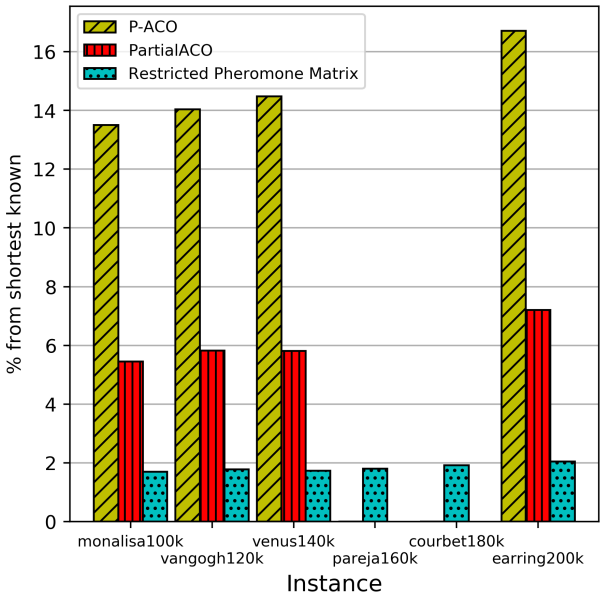

Figure 4: Plot of the solution quality difference between P-ACO, PartialACO (results taken from [9] and Restricted Pheromone Matrix (our experiments) against shortest known tour. No P-ACO or PartialACO results are available for `pareja160k` and `courbet180k`.

comparable times to solution, in both cases using recent commodity hardware.

Figure 5 plots solution quality over time for each of the instances. Although small gains are still being made when our runs are terminated, in all cases the solutions are well converged.

Table 3: Execution times for PartialACO and Restricted Pheromone Matrix.

Instance	Execution Time (Hours)	
	PartialACO	Restricted Matrix
mona-lisa100k	1.07	1.36
vangogh-120k	1.45	1.92
venus140k	2.09	2.63
pareja160k	N/A	3.45
courbet180k	N/A	4.5
earring200k	5.06	6

4.4 Discussion

We have demonstrated the feasibility of scaling up ACO to solve large ($> 10^5$ city) instances of TSP, and shown that ACO can produce tours within 2% of the best known on a selection of well-known large instances. Our code runs in times of order an hour on commodity hardware, compared to the supercomputing resources required to find the best-known tours using genetic algorithms[18]. We note that our solution qualities degrade only slightly between the `mona-lisa100k` and `earring200k` instances, with only a minimal difference of $\sim 0.2\%$ (compared to the almost 2% degradation seen using PartialACO). This consistency of solution qualities suggests that our technique could potentially be used to obtain good quality tours for problem instances that are even larger than the Art TSPs.

While it is perhaps intuitively obvious that the Pheromone Map fallback should produce better quality solutions (due to the availability of more accurate edge weight information through the use of pheromone), we find that ignoring pheromone on edges outside the candidate set has little impact. We should note that, overall, the fallback rate is very low, with fewer than 5% of tour construction selections being made using either fallback method. While pheromone is an integral part of ACO, our experiments suggest that it is less important when the cities being traveled between are significantly far apart. Quantifying the effect of pheromone at varying distances in the nearest-neighbour list is an area for future work. Given the negligible difference in solution quality, the much faster execution time of the Heuristic fallback makes it a far more practical technique than the Pheromone Map fallback.

5 CONCLUSIONS

In this paper we presented a Restricted Pheromone Matrix method which allows ACO to be used to solve large instances of the TSP, by reducing the memory complexity from quadratic to linear. We also presented two selection techniques for cities outside the nearest neighbour list. By combining the Restricted Pheromone Matrix with the Heuristic Fallback technique, we found tours that are within 2% of the best known solutions for the Art TSP instances, a substantial improvement on previous attempts using ACO. Importantly, our implementation closely follows the original $\mathcal{MM}$AS algorithm, and represents the first evaluation of this algorithm on large instances of the TSP.

While the substantial reduction in memory size allows us to solve much larger instances than previously possible, the time complexity of ACO remains a limiting factor. Though the execution time is greatly reduced through the use of parallel and vector methods such as the *VCSS* selection technique, substantial changes to the core ACO algorithm would be required to reduce this complexity. However, neither of our fallback techniques currently uses the vector instructions employed by, for example, I-Roulette and VCSS, and a significant speedup could be obtained by vectorizing the fallback algorithms. Future work will focus on this.

Finally, we note that many problems to which ACO has been successfully applied share with the TSP the properties of quadratic memory complexity and the use of candidate sets to accelerate the solution. Examples include the Quadratic Assignment Problem [21], Resource-constrained project scheduling problems [23], and vehicle routing problems [4]. The methods presented in this paper could be also be applied in these cases, where the solution of large instances is limited by memory.

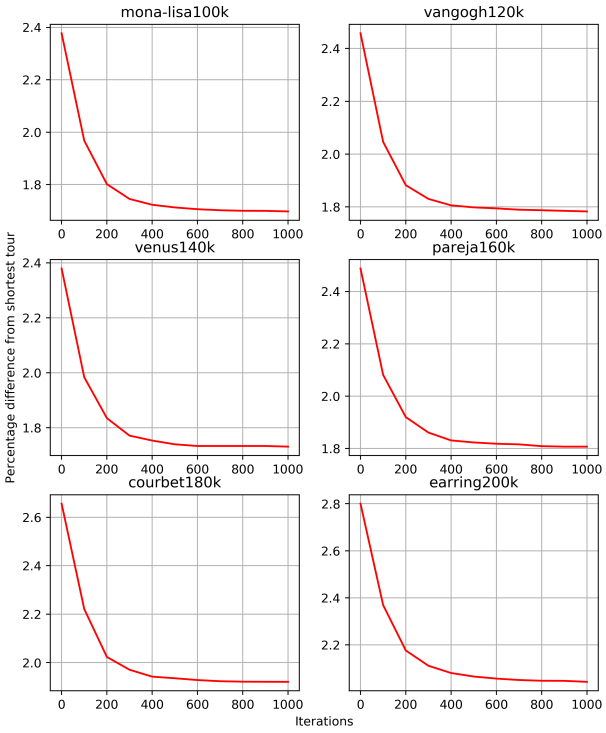

Figure 5: Solution quality versus iteration for the Art TSP instances using the heuristic fallback.

ACKNOWLEDGEMENTS

We thank René Doursat for useful discussions.

REFERENCES

[1] TSP Art Instances. http://www.math.uwaterloo.ca/tsp/data/art/. Accessed: 2019-01-30.

[2] Emile Aarts and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[3] Enrique Alba and Francisco Chicano. ACOhg: Dealing with Huge Graphs. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 10–17, New York, NY, USA, 2007. ACM.

[4] John E. Bell and Patrick R. McMullen. Ant Colony Optimization Techniques for the Vehicle Routing Problem. *Advanced Engineering Informatics*, 18(1):41 – 48, 2004.

[5] Bernd Bullnheimer, Gabriele Kotsis, and Christine Strauß. Parallelization Strategies for the Ant System. In *High Performance Algorithms and Software in Nonlinear Optimization*, pages 87–100. Springer, 1998.

[6] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing Data Parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[7] José M. Cecilia, José M. García, Manuel Ujaldón, Andy Nisbet, and Martyn Amos. Parallelization Strategies for Ant Colony Optimization on GPUs. In *Proceedings of the 25th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2011)*, pages 334–341, 2011.

[8] Ling Chen and Chunfang Zhang. Adaptive Parallel Ant Colony Algorithm. In *International Conference on Natural Computation*, pages 1239–1249. Springer, 2005.

[9] Darren M. Chitty. Applying ACO to Large Scale TSP Instances. In *UK Workshop on Computational Intelligence*, pages 104–118. Springer, 2017.

[10] Laurence Dawson and Iain Stewart. Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 216–225. Springer, 2013.

[11] Laurence Dawson and Iain Stewart. Improving Ant Colony Optimization Performance on the GPU using CUDA. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1901–1908. IEEE, 2013.

[12] Jean-Louis Deneubourg and Simon Goss. Collective Patterns and Decision-making. *Ethology Ecology & Evolution*, 1(4):295–311, 1989.

[13] Jean-Louis Deneubourg, Jacques M. Pasteels, and Jean-Claude Verhaeghe. Probabilistic Behaviour in Ants: a Strategy of Errors? *Journal of Theoretical Biology*, 105(2):259–271, 1983.

[14] Karl F. Doerner, Richard F. Hartl, Siegfried Benkner, and Maria Lucka. Parallel Cooperative Savings-based Ant Colony Optimization — Multiple Search and Decomposition Approaches. *Parallel Processing Letters*, 16(03):351–369, 2006.

[15] Marco Dorigo and Gianna Di Caro. Ant Colony Optimization: a New Meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 2, 1999.

[16] Marco Dorigo and Luca Maria Gambardella. Ant Colonies for the Travelling Salesman Problem. *BioSystems*, 43(2):73–81, 1997.

[17] Michael Guntsch and Martin Middendorf. A population based approach for aco. In *Workshops on Applications of Evolutionary Computation*, pages 72–81. Springer, 2002.

[18] Kazuma Honda, Yuichi Nagata, and Isao Ono. A Parallel Genetic Algorithm with Edge Assembly Crossover for 100,000-City Scale TSPs. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1278–1285. IEEE, 2013.

[19] Huw Lloyd and Martyn Amos. A Highly Parallelized and Vectorized Implementation of Max-Min Ant System on Intel® Xeon Phi™. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6. IEEE, 2016.

[20] Huw Lloyd and Martyn Amos. Analysis of Independent Roulette Selection in Parallel Ant Colony Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 19–26, New York, New York, USA, 2017. ACM Press.

[21] Manuel López-Ibáñez, Thomas Stützle, and Marco Dorigo. *Ant Colony Optimization: A Component-Wise Overview*, pages 1–37. Springer International Publishing, Cham, 2016.

[22] Michalis Mavrovouniotis, Felipe M. Müller, and Shengxiang Yang. Ant Colony Optimization With Local Search for Dynamic Traveling Salesman Problems. *IEEE Transactions on Cybernetics*, 47(7):1743–1756, 2017.

[23] Daniel Merkle, Martin Middendorf, and Hartmut Schmeck. Ant Colony Optimization for Resource-constrained Project Scheduling. *IEEE Transactions on Evolutionary Computation*, 6(4):333–346, 2002.

[24] Victoriano Montesinos and José M. García. Vectorization Strategies for Ant Colony Optimization on Intel Architectures. *Parallel Computing is Everywhere*, 32:400, 2018.

[25] Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. Vectorized Candidate Set Selection for Parallel Ant Colony Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1300–1306. ACM, 2018.

[26] Marcus Randall and Andrew Lewis. A Parallel Implementation of Ant Colony Optimization. *Journal of Parallel and Distributed Computing*, 62(9):1421–1432, 2002.

[27] Rafał Skinderowicz. The GPU-based Parallel Ant Colony System. *Journal of Parallel and Distributed Computing*, 98:48–60, 2016.

[28] Thomas Stützle. Parallelization Strategies for Ant Colony Optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 722–731. Springer, 1998.

[29] Thomas Stützle. ACOTSP, 2004. Available from http://www.aco-metaheuristic.org/aco-code, 2004.

[30] Thomas Stützle and Holger H. Hoos. MAX–MIN Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[31] Thomas Stützle and Marco Dorigo. Ant Colony Optimization. 2004.

[32] Felipe Tirado, Ricardo J. Barrientos, Paulo González, and Marco Mora. Efficient Exploitation of the Xeon Phi Architecture for the Ant Colony Optimization (ACO) Metaheuristic. *The Journal of Supercomputing*, 73(11):5053–5070, 2017.

[33] Felipe Tirado, Angelica Urrutia, and Ricardo J. Barrientos. Using a Coprocessor to Solve the Ant Colony Optimization Algorithm. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6. IEEE, 2015.

[34] Colin Twomey, Thomas Stützle, Marco Dorigo, Max Manfrin, and Mauro Birattari. An Analysis of Communication Policies for Homogeneous Multi-Colony ACO Algorithms. *Information Sciences*, 180(12):2390–2404, 2010.